UFES - Universidade Federal do Espírito Santo

Engenharia de Software

Notas de Aula

Ricardo de Almeida Falbo

falbo@inf.ufes.br

2014

Sumário

PARTE II – Gerência de Software

Capítulo 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 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 do 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 e acabamento. 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 é a área da Ciência da Computação voltada à especificação, desenvolvimento e manutenção de sistemas de software, com aplicação de tecnologias e práticas de gerência de projetos e outras disciplinas, visando organização, produtividade e qualidade no processo 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. Para cada uma dessas partes, uma solução deve ser elaborada. Solucionados os subproblemas 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 automatizar, pelo menos parcialmente, o trabalho.

Neste cenário, raramente é possível conduzir o desenvolvimento de um produto 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.

1.2 – Processo de Software

Uma vez que a qualidade do processo de software é o caminho para se obter a qualidade dos produtos de software, é importante estabelecer um processo de software de qualidade. Mas o que é um processo de software?

Um processo de software pode ser visto como o conjunto de atividades, métodos e práticas que guiam os profissionais 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.

De maneira geral, os processos de software são decompostos em processos menores (ditos subprocessos), tais como processo de desenvolvimento, processo de garantia da qualidade, processo de gerência de projetos etc. Esses processos, por sua vez, são compostos de atividades, que também podem ser decompostas. Para cada atividade de um processo é importante saber quais as suas subatividades, as atividades que devem precedê-las (pré-atividades), os artefatos de entrada (insumos) e de saída (produtos), os recursos necessários (humanos, hardware, software etc.) e os procedimentos (métodos, técnicas, roteiros, modelos de documento etc.) a serem utilizados na sua realização. A Figura 1.1 mostra os elementos que compõem um processo de forma esquemática.

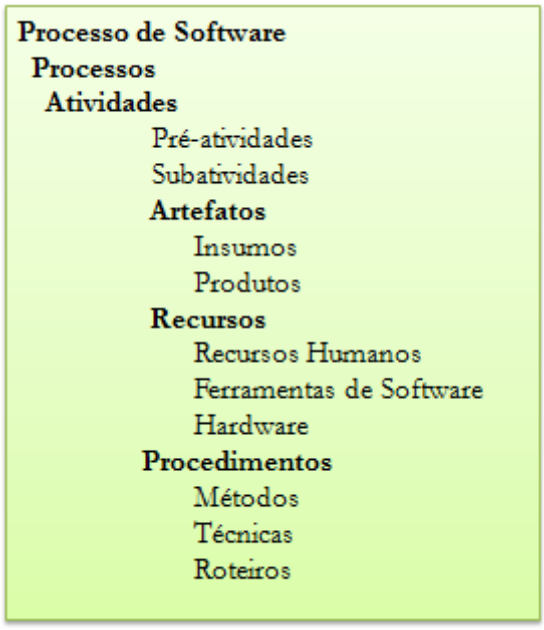


Figura 1.1 – Elementos que compõem um Processo de Software

O processo de desenvolvimento de software, como o próprio nome indica, é a espinha dorsal do desenvolvimento de software e envolve as atividades que contribuem diretamente para o desenvolvimento do produto de software a ser entregue ao cliente. São exemplos de

atividades do processo de desenvolvimento: especificação e análise de requisitos, projeto e implementação.

Paralelamente ao processo de software, diversos processos de apoio e de gerência são realizados. O processo de gerência de projetos envolve atividades relacionadas ao planejamento e ao acompanhamento gerencial do projeto, tais como realização de estimativas, elaboração de cronogramas, análise dos riscos do projeto etc. Os processos de apoio, por sua vez, visam apoiar (provendo informações ou serviços) as atividades dos demais processos (incluindo, além dos processos de desenvolvimento e de gerência de projetos, os próprios processos de apoio). Dentre os processos de apoio, há os processos de medição, gerência de configuração de software, garantia da qualidade, verificação e validação. As atividades dos processos de gerência de projetos e de apoio não estão ligadas diretamente à construção do produto final (o software a ser entregue para o cliente, incluindo toda a documentação necessária) e, 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 1.2 mostra a relação entre os vários tipos de processos.

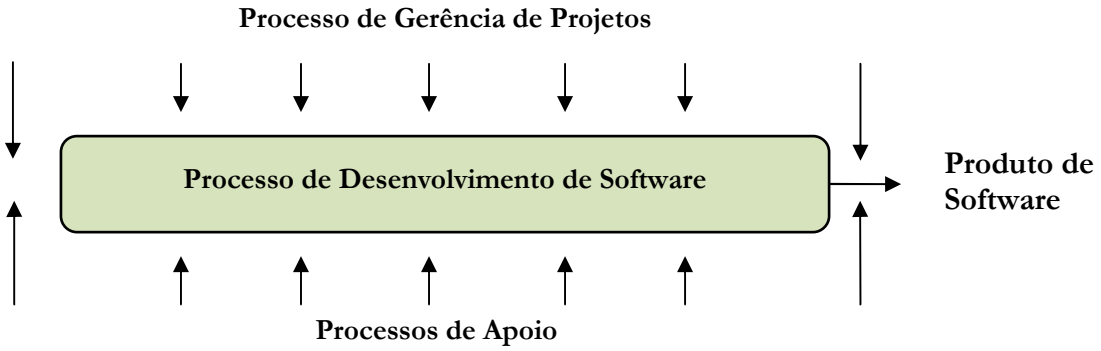


Figura 1.2 – Decomposição do Processo de Software

1.3 – A Organização deste Texto

Neste texto, procura-se oferecer uma visão geral da Engenharia de Software, discutindo seus principais processos e atividades e como realizá-los. Este texto está dividido em três partes: na Parte I são abordados os processos de desenvolvimento e manutenção de software; na Parte II são abordados o processo de gerência de projetos e os processos de apoio. Essas partes estão organizadas nos seguintes capítulos:

PARTE I: Desenvolvimento e Manutenção de Software

- Capítulo 2 – *Visão Geral do Processo de Desenvolvimento*: procura dar uma visão geral das atividades que compõem o processo de desenvolvimento de software, bem como apresenta os principais modelos de ciclo de vida que organizam essas atividades.
- Capítulo 3 – *Requisitos de Software*: aborda requisitos e tipos de requisitos e procura dar uma visão geral da Engenharia de Requisitos, discutindo alguns dos

métodos e técnicas utilizados nesta fase, com destaque para a Modelagem de Casos de Uso e a Modelagem de Entidades e Relacionamentos.

- Capítulo 4 – *Projeto de Software*: aborda os conceitos básicos de projeto de sistemas de software, tratando da arquitetura do sistema a ser desenvolvido e do projeto de seus componentes. Também são discutidos o projeto de dados e o projeto de interface com o usuário.
- Capítulo 5 – *Implementação e Testes de Software*: são enfocadas as atividades de implementação e testes, sendo esta última tratada em diferentes níveis, a saber: teste de unidade, teste de integração e teste de sistema.
- Capítulo 6 – *Entrega e Manutenção*: discute as questões relacionadas à entrega do sistema para o cliente, tais como o treinamento e a documentação de entrega, e o processo de manutenção de software.

PARTE II: Gerência de Software

- Capítulo 7 – *Gerência da Qualidade*: são abordadas técnicas para revisão de software, bem como processos importantes para a garantia da qualidade, a saber: documentação, gerência de configuração de software e medição de software .
- Capítulo 8 – *Gerência de Projetos*: são abordadas as principais atividades do processo de gerência de projetos, a saber: planejamento, acompanhamento e encerramento de projetos.
- Capítulo 9 – *Tópicos Avançados em Engenharia de Software*: discute alguns aspectos adicionais da Engenharia de Software, a saber: normas e modelos de qualidade do processo de software, processos padrão, processos ágeis e apoio automatizado ao processo de software.

O desenvolvimento de um produto de software é norteado pela escolha de um 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, discutimos as atividades do processo de desenvolvimento de software à luz do paradigma estruturado.

PARTE I

Desenvolvimento e Manutenção de Software

Capítulo 2 – Visão Geral do Processo de Desenvolvimento

O processo de desenvolvimento de software engloba as atividades que contribuem diretamente para o desenvolvimento do produto de software a ser entregue ao cliente, incluindo a sua documentação. De maneira geral, o processo de desenvolvimento de software envolve as seguintes atividades: Análise e Especificação de Requisitos, Projeto, Implementação, Testes, Entrega e Implantação do Sistema.

Na Análise e Especificação de Requisitos, o foco está no levantamento, compreensão e especificação dos requisitos que o produto de software deve ser capaz de satisfazer. 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 para o sistema. Uma vez capturados os requisitos do sistema, estes devem ser modelados, avaliados e documentados. Uma parte vital desta fase é a construção de modelos descrevendo *o quê* o software tem de fazer (e não *como* fazê-lo), ditos modelos conceituais.

A fase de Projeto é responsável por incorporar requisitos tecnológicos aos requisitos essenciais do sistema e, portanto, requer que a plataforma de implementação seja conhecida. Basicamente, envolve duas grandes etapas: projeto da arquitetura do sistema e o 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. Essa 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 maiores de detalhamento, até que possam ser codificados e testados.

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.

A fase de 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. A seguir, os diversos componentes devem ser integrados sucessivamente até se obter o sistema. Finalmente, o sistema como um todo deve ser testado.

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 da fase de Implantação e Entrega é disponibilizar o software para o cliente, garantindo que o mesmo satisfaz os requisitos estabelecidos. Isto requer a instalação do software e a condução de testes de aceitação. Quando o software tiver demonstrado prover as capacidades requeridas, ele pode ser aceito e a operação iniciada.

Encerrado o processo de desenvolvimento, com a entrega do sistema ao cliente, diz-se que o sistema está em operação, quando o software é utilizado pelos usuários no ambiente de produção. Contudo, indubitavelmente, o software sofrerá mudanças após ter sido entregue para o cliente. 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, a manutenção pode requerer a definição de um

novo processo, onde cada uma das fases do processo de desenvolvimento é reaplicada no contexto de um software existente ao invés de um novo.

Ainda que as atividades do processo de desenvolvimento guardem certa relação de precedência, como os parágrafos anteriores sugerem, não é verdade que cada uma das atividades tenha de ter sido necessariamente concluída para que a próxima possa ser iniciada. Muito pelo contrário, as atividades do processo de desenvolvimento podem ser organizadas de maneiras bastante diferentes. Para capturar algumas formas de se estruturar as atividades do processo de desenvolvimento, são definidos modelos de processo.

Um modelo de processo (ou modelo de ciclo de vida) pode ser visto como uma representação abstrata de um esqueleto de processo, incluindo tipicamente algumas atividades principais e a ordem de precedência entre elas. De maneira geral, um modelo de processo descreve uma filosofia de organização de atividades, estruturando as atividades do processo em fases e definindo como essas fases estão relacionadas. Entretanto, ele não descreve um curso de ações preciso, recursos, procedimentos e restrições. Ou seja, ele é um importante ponto de partida para definir como o projeto será conduzido, mas a sua adoção não é o suficiente para guiar e controlar um projeto de software na prática.

Os modelos de ciclo de vida, de maneira geral, contemplam apenas as fases do processo de desenvolvimento (Análise e Especificação de Requisitos, Projeto, Implementação, Testes e Entrega e Implantação). A escolha de um modelo de processo é fortemente dependente das características do projeto, dentre elas: tipo de software a ser desenvolvido (p.ex., sistema de informação, sistema de tempo real etc.), paradigma de desenvolvimento (estruturado, orientado a objetos etc.), tamanho e complexidade do sistema, estabilidade dos requisitos e características da equipe. Assim, é importante conhecer alguns modelos e em que situações são aplicáveis. Os principais modelos de processo podem ser agrupados em três categorias principais: modelos sequenciais, modelos incrementais e modelos evolutivos.

2.1 - Modelos de Processo Sequenciais

Como o nome indica, os modelos sequenciais organizam o processo em uma sequência linear de atividades. O principal modelo desta categoria é o modelo em cascata, a partir do qual diversos outros modelos foram propostos, inclusive a maioria dos modelos incrementais e evolutivos.

2.1.1 – O Modelo em Cascata

Também chamado de modelo de ciclo de vida clássico, o modelo em cascata organiza as atividades do processo de desenvolvimento de forma sequencial, como mostra a Figura 2.1. Cada fase envolve a elaboração de um ou mais documentos, que devem ser aprovados antes de se iniciar a fase seguinte. Assim, uma fase só deve ser iniciada após a conclusão daquela que a precede. Uma vez que, na prática, essas fases se sobrepõem de alguma forma, geralmente, permite-se um retorno à fase anterior para a correção de erros encontrados. A entrega do sistema completo ocorre em um único marco, ao final da fase de Entrega e Implantação.

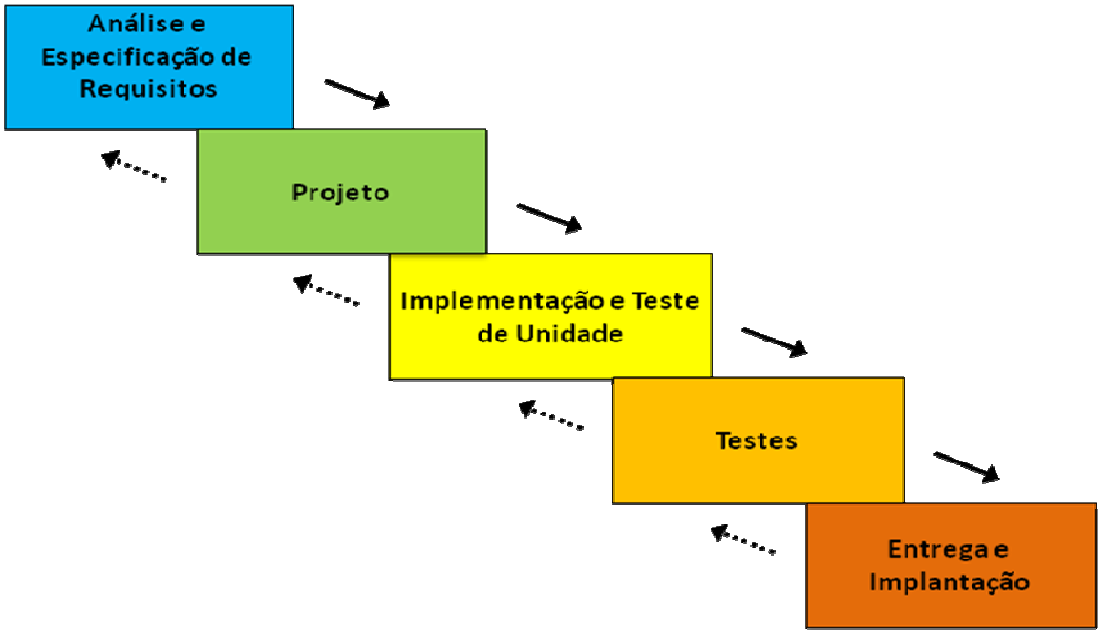


Figura 2.1 - O Modelo em Cascata.

O uso de revisões ao fim de cada fase permite o envolvimento do usuário. Além disso, cada fase serve como uma base aprovada e documentada para o passo seguinte, facilitando bastante a gerência do projeto.

O modelo em cascata é o modelo de ciclo de vida mais antigo e mais amplamente usado. Entretanto, críticas têm levado ao questionamento de sua eficiência. Dentre os problemas algumas vezes encontrados na sua aplicação, destacam-se (PRESSMAN, 2011):

- Projetos reais muitas vezes não seguem o fluxo sequencial que o modelo propõe.
- Os requisitos devem ser estabelecidos de maneira completa, correta e clara logo no início de um projeto. A aplicação deve, portanto, ser entendida pelo desenvolvedor desde o início do projeto. Entretanto, frequentemente, é difícil para o usuário colocar todos os requisitos explicitamente. O modelo em cascata requer isto e tem dificuldade de acomodar a incerteza natural que existe no início de muitos projetos.
- O usuário precisa ser paciente. Uma versão operacional do software não estará disponível até o final do projeto.
- A introdução de certos membros da equipe, tais como projetistas e programadores, é frequentemente adiada desnecessariamente. A natureza linear do ciclo de vida clássico leva a estados de bloqueio nos quais alguns membros da equipe do projeto precisam esperar que outros membros da equipe completem tarefas dependentes.

Cada um desses problemas é real. Entretanto, o modelo de ciclo de vida clássico tem um lugar definitivo e importante na Engenharia de Software. Muitos outros modelos mais complexos são, na realidade, variações do modelo cascata, incorporando laços de realimentação (PFLEEGER, 2004). Embora tenha fraquezas, ele é significativamente melhor do que uma abordagem casual para o desenvolvimento de software. De fato, para problemas pequenos e bem definidos, onde os desenvolvedores conhecem bem o domínio do problema e os requisitos podem ser claramente estabelecidos, esse modelo é indicado, uma vez que é fácil de ser gerenciado.

2.1.2 – O Modelo em V

O modelo em V é uma variação do modelo em cascata que procura enfatizar a estreita relação entre as atividades de teste (teste de unidade, teste de integração, teste de sistema e teste de aceitação) e as demais fases do processo de desenvolvimento (PFLEEGER, 2004), como mostra a Figura 2.2.

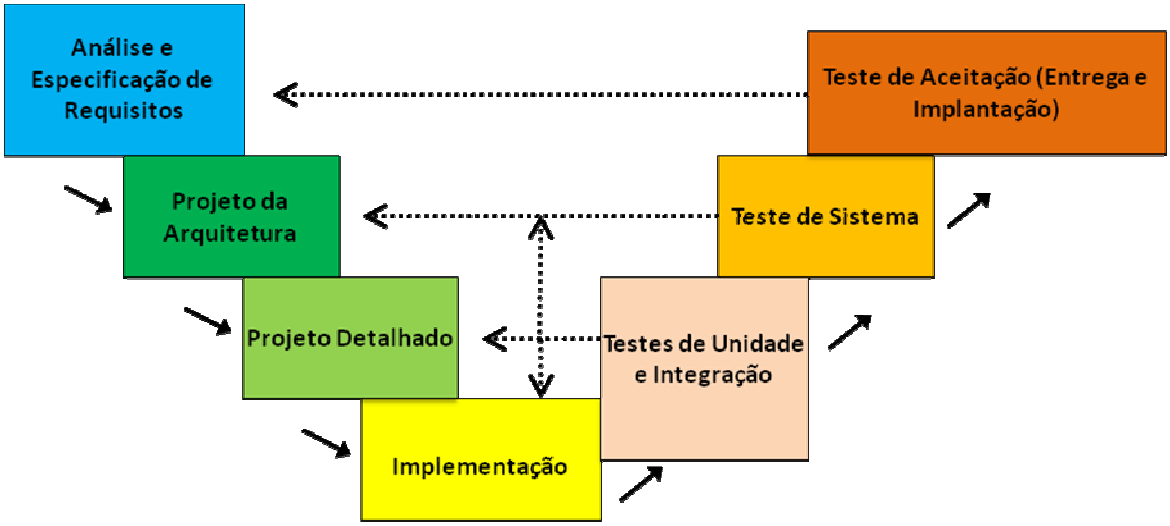


Figura 2.2 - O Modelo em V.

O modelo em V sugere que os testes de unidade são utilizados basicamente para verificar a implementação e o projeto detalhado. Uma vez que os testes de integração estão focados na integração das unidades que compõem o software, eles também são usados para avaliar o projeto detalhado. Assim, testes de unidade e integração devem garantir que todos os aspectos do projeto do sistema foram implementados corretamente no código. Quando os testes de integração atingem o nível do sistema como um todo (teste de sistema), o projeto da arquitetura passa a ser o foco. Neste momento, busca-se verificar se o sistema atende aos requisitos definidos na especificação. Finalmente, os testes de aceitação, conduzidos tipicamente pelos usuários e clientes, buscam validar os requisitos, confirmando que os requisitos corretos foram implementados no sistema (teste de validação).

A conexão entre os lados direito e esquerdo do modelo em V implica que, caso sejam encontrados problemas em uma atividade de teste, a correspondente fase do lado esquerdo e suas fases subsequentes podem ter de ser executadas novamente para corrigir ou melhorar esses problemas.

Os modelos sequenciais pressupõem que o sistema é entregue completo, após a realização de todas as atividades do desenvolvimento. Entretanto, nos dias de hoje, os clientes não estão mais dispostos a esperar o tempo necessário para tal, sobretudo, quando se trata de grandes sistemas (PFLEEGER, 2004). Dependendo do porte do sistema, podem se passar anos até que o sistema fique pronto, sendo inviável esperar. Assim, outros modelos foram propostos visando a, dentre outros, reduzir o tempo de desenvolvimento. A entrega por partes, possibilitando ao usuário dispor de algumas funcionalidades do sistema enquanto outras estão sendo ainda desenvolvidas, é um dos principais mecanismos utilizados por esses modelos, como discutido a seguir.

2.2 - Modelos de Processo Incrementais

Há muitas situações em que os requisitos são razoavelmente bem definidos, mas o tamanho do sistema a ser desenvolvido impossibilita a adoção de um modelo sequencial, sobretudo pela necessidade de disponibilizar rapidamente uma versão para o usuário. Nesses casos, um modelo incremental é indicado (PRESSMAN, 2011).

No desenvolvimento incremental, o sistema é dividido em subsistemas ou módulos, tomando por base a funcionalidade. Os incrementos (ou versões) são definidos, começando com um pequeno subsistema funcional que, a cada ciclo, é acrescido de novas funcionalidades. Além de acrescentar novas funcionalidades, nos novos ciclos as funcionalidades providas anteriormente podem ser modificadas para melhor satisfazer às necessidades dos clientes / usuários.

2.2.1 – O Modelo de Processo Incremental

O modelo incremental pode ser visto como uma filosofia básica que comporta diversas variações. O princípio fundamental é que, a cada ciclo ou iteração, uma versão operacional do sistema é produzida e entregue para uso ou avaliação detalhada do cliente. Para tal, requisitos têm de ser minimamente levantados e há de se constatar que o sistema é modular, de modo que se possa planejar o desenvolvimento em incrementos. O primeiro incremento tipicamente contém funcionalidades centrais, tratando dos requisitos básicos. Outras características são tratadas em ciclos subsequentes.

Dependendo do tempo estabelecido para a liberação dos incrementos, algumas atividades podem ser feitas para o sistema como um todo ou não. A Figura 2.4 mostra as principais possibilidades.

Na parte (a) da Figura 2.4, inicialmente, durante a fase de planejamento, um levantamento preliminar dos requisitos é realizado. Com esse esboço dos requisitos, planejam-se os incrementos e se desenvolve a primeira versão do sistema. Concluído o primeiro ciclo, todas as atividades são novamente realizadas para o segundo ciclo, inclusive o planejamento, quando a atribuição de requisitos aos incrementos pode ser revista. Este procedimento é repetido sucessivamente, até que se chegue ao produto final.

Outras duas variações do modelo incremental bastante utilizadas são apresentadas nas partes (b) e (c) da Figura 2.4. Na parte (b) dessa figura, os requisitos são especificados para o sistema como um todo e as iterações ocorrem a partir da fase de análise. Na Figura 2.4 (c), o sistema tem seus requisitos especificados e analisados, a arquitetura do sistema é definida e apenas o projeto detalhado, a implementação e os testes são realizados em vários ciclos. Uma vez que nessas duas variações os requisitos são especificados para o sistema como um todo, sua adoção requer que os requisitos sejam estáveis e bem definidos.

O modelo incremental é particularmente útil quando não há pessoal suficiente para realizar o desenvolvimento dentro dos prazos estabelecidos ou para lidar com riscos técnicos, tal como a adoção de uma nova tecnologia (PRESSMAN, 2011).

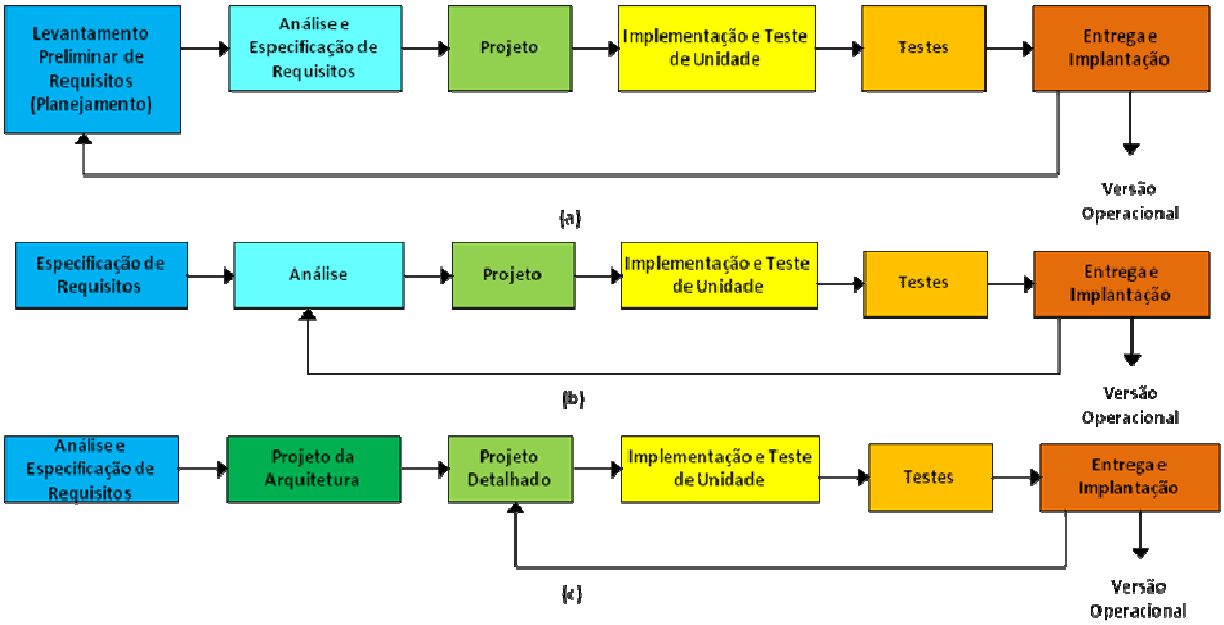


Figura 2.4 – Variações do Modelo Incremental.

Dentre as vantagens do modelo incremental, podem ser citadas (CHRISTENSEN; THAYER, 2002):

- Menor custo e menos tempo são necessários para se entregar a primeira versão;
- Os riscos associados ao desenvolvimento de um incremento são menores, devido ao seu tamanho reduzido;
- O número de mudanças nos requisitos pode diminuir devido ao curto tempo de desenvolvimento de um incremento.

Como desvantagens, podemos citar (CHRISTENSEN; THAYER, 2002):

- Se os requisitos não são tão estáveis ou completos quanto se esperava, alguns incrementos podem ter de ser bastante alterados;
- A gerência do projeto é mais complexa, sobretudo quando a divisão em subsistemas inicialmente feita não se mostrar boa.

2.2.2 – O Modelo RAD

O modelo RAD (*Rapid Application Development*), ou modelo de desenvolvimento rápido de aplicações, é um tipo de modelo incremental que prima por um ciclo de desenvolvimento curto (tipicamente de até 90 dias) (PRESSMAN, 2011). Assim, como no modelo incremental, o sistema é subdividido em subsistemas e incrementos são realizados. A diferença marcante é que os incrementos são desenvolvidos em paralelo por equipes distintas e apenas uma única entrega é feita, como mostra a Figura 2.5.

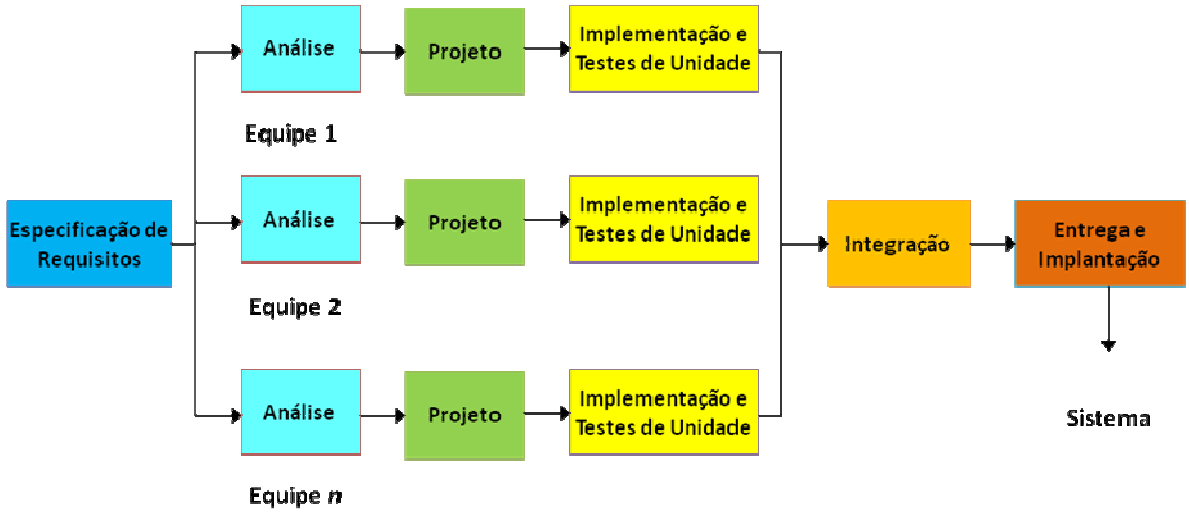


Figura 2.5 – O Modelo RAD.

Assim, como o modelo incremental, o modelo RAD permite variações. Em todos os casos, no entanto, os requisitos têm de ser bem definidos, o escopo do projeto tem de ser restrito e o sistema modular. Se o projeto for grande, por exemplo, o número de equipes crescerá demais e a atividade de integração tornar-se-á por demais complexa. Obviamente, para adotar esse modelo, uma organização tem de ter recursos humanos suficientes para acomodar as várias equipes.

2.3 - Modelos de Processo Evolutivos

Sistemas de software, como quaisquer sistemas complexos, evoluem ao longo do tempo. Seus requisitos, muitas vezes, são difíceis de serem estabelecidos ou mudam com frequência ao longo do desenvolvimento (PRESSMAN, 2011). Assim, é importante ter como opção modelos de ciclo de vida que lidem com incertezas e acomodem melhor as contínuas mudanças. Alguns modelos incrementais, dado que preconizam um desenvolvimento iterativo, podem ser aplicados a esses casos, mas a grande maioria deles toma por pressuposto que os requisitos são bem definidos. Modelos evolucionários ou evolutivos buscam preencher essa lacuna.

Enquanto modelos incrementais têm por base a entrega de versões operacionais desde o primeiro ciclo, os modelos evolutivos não têm essa preocupação. Muito pelo contrário: na maioria das vezes, os primeiros ciclos produzem protótipos ou até mesmo apenas modelos. À medida que o desenvolvimento avança e os requisitos vão ficando mais claros e estáveis, protótipos vão dando lugar a versões operacionais, até que o sistema completo seja construído. Assim, quando o problema não é bem definido e ele não pode ser totalmente especificado no início do desenvolvimento, é melhor optar por um modelo evolutivo. A avaliação ou o uso do protótipo / sistema pode aumentar o conhecimento sobre o produto e melhorar o entendimento que se tem acerca dos requisitos. Entretanto, a gerência do projeto e a gerência de configuração precisam ser bem estabelecidas e conduzidas.

2.3.1 – O Modelo em Espiral

O modelo espiral, mostrado na Figura 2.6, é um dos modelos evolutivos mais difundidos.

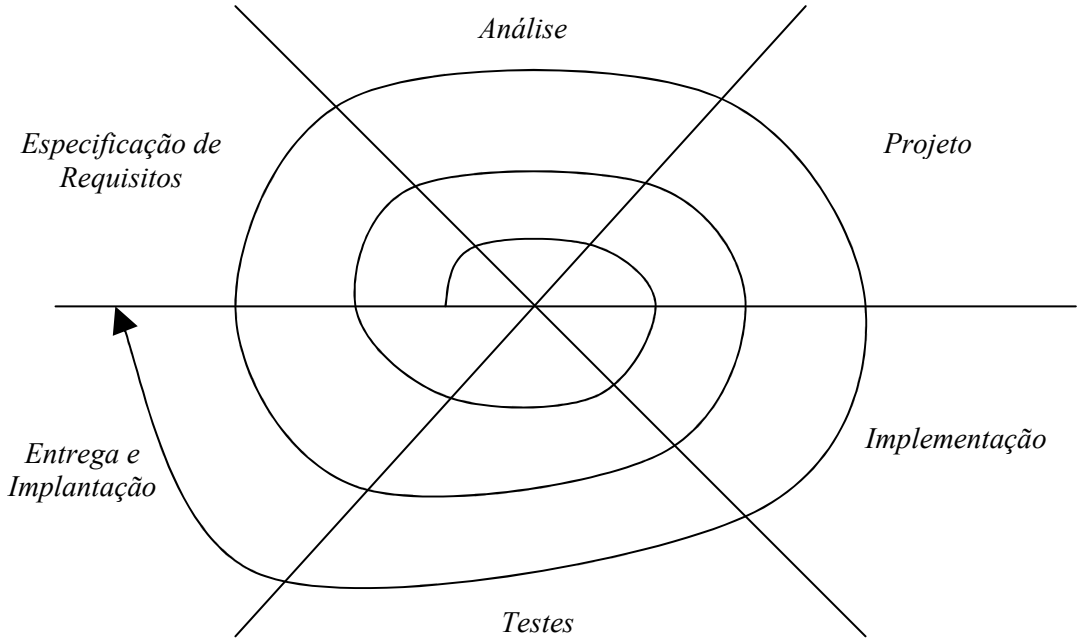


Figura 2.6 – O Modelo Espiral.

Ao se adotar o modelo espiral, o sistema é desenvolvido em ciclos, sendo que nos primeiros ciclos nem sempre todas as atividades são realizadas. Por exemplo, o produto resultante do primeiro ciclo pode ser uma especificação do produto ou um estudo de viabilidade. As passadas subsequentes ao longo da espiral podem ser usadas para desenvolver protótipos, chegando progressivamente a versões operacionais do software, até se obter o produto completo. Até mesmo ciclos de manutenção podem ser acomodados nesta filosofia, fazendo com que o modelo espiral contemple todo o ciclo de vida do software (PRESSMAN, 2011).

É importante ressaltar que, a cada ciclo, o planejamento deve ser revisto com base no feedback do cliente, ajustando, inclusive, o número de iterações planejadas. De fato, este é o maior problema do ciclo de vida espiral, ou de maneira geral, dos modelos evolucionários: a gerência de projetos. Pode ser difícil convencer clientes, especialmente em situações envolvendo contrato, que a abordagem evolutiva é gerenciável (PRESSMAN, 2011).

2.4 - Prototipação

Muitas vezes, clientes têm em mente um conjunto geral de objetivos para um sistema de software, mas não são capazes de identificar claramente as funcionalidades ou informações (requisitos) que o sistema terá de prover ou tratar. Modelos podem ser úteis para ajudar a levantar e validar requisitos, mas pode ocorrer dos clientes e usuários só terem uma verdadeira dimensão do que está sendo construído se forem colocados diante do sistema. Nestes casos, o uso da prototipação é fundamental. A prototipação é uma técnica para ajudar

engenheiros de software e clientes a entender o que está sendo construído quando os requisitos não estão claros. Ainda que tenha sido citada anteriormente no contexto do modelo espiral, ela pode ser aplicada no contexto de qualquer modelo de processo. A Figura 2.7, por exemplo, ilustra um modelo em cascata com prototipação (PFLEEGER, 2004).

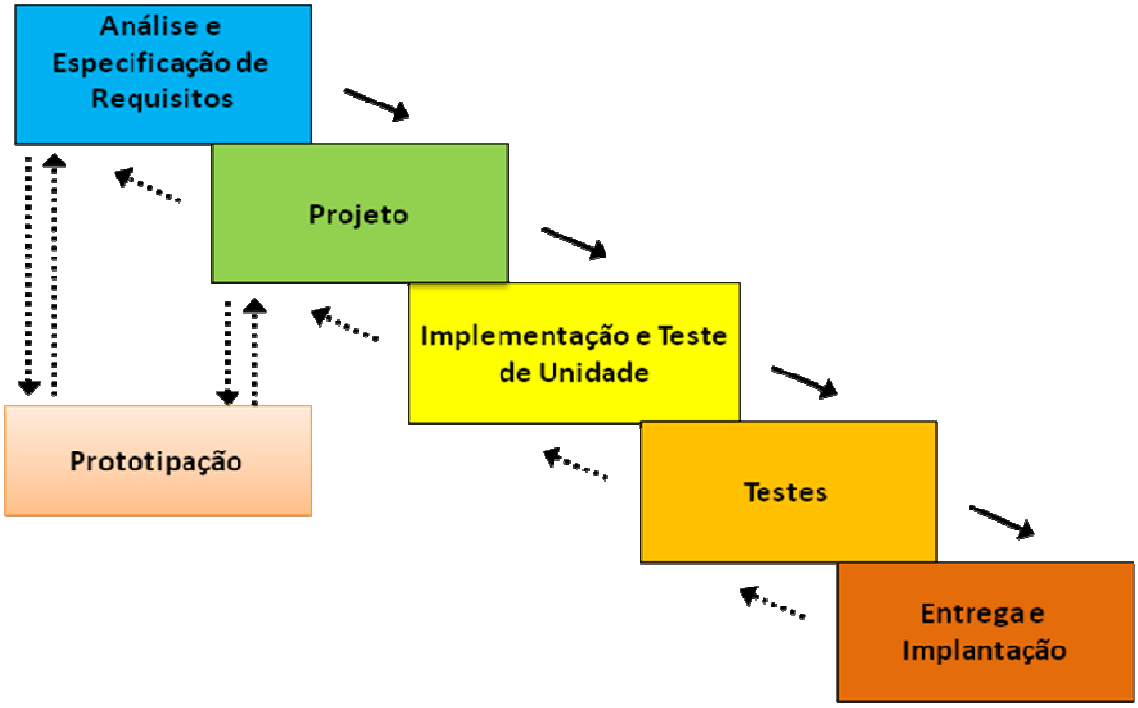


Figura 2.7 - O Modelo em Cascata com Prototipação.

2.5 – O Processo Unificado

O Modelo do Processo Unificado, muitas vezes referenciado como RUP (*Rational Unified Process*), é um modelo de processo bastante elaborado, que procura incorporar elementos de vários dos modelos de processo anteriormente apresentados, em uma tentativa de incorporar as melhores práticas de desenvolvimento de software, dentre elas a prototipação e a entrega incremental (SOMMERVILLE, 2011).

Ainda que apresentado neste texto isoladamente em uma seção, o modelo de processo do RUP é um modelo evolutivo, como ilustra a Figura 2.8, na medida em que preconiza o desenvolvimento em ciclos, de modo a permitir uma melhor compreensão dos requisitos. De fato, a opção por apresentá-lo em uma seção separada é que o RUP não é apenas um modelo de processo de desenvolvimento. Ele é muito mais do que isso. Ele é uma abordagem completa para o desenvolvimento de software, incluindo, além de um modelo de processo de software, a definição detalhada de responsabilidades (papéis), atividades, artefatos e fluxos de trabalho, dentre outros.

O modelo de processo do RUP tem como diferencial principal a sua organização em duas dimensões, como ilustra a Figura 2.9. Na dimensão horizontal, é representada a estrutura do modelo em relação ao tempo, a qual é organizada em fases e iterações. Na dimensão vertical, são mostradas as atividades (chamadas de disciplinas no RUP).

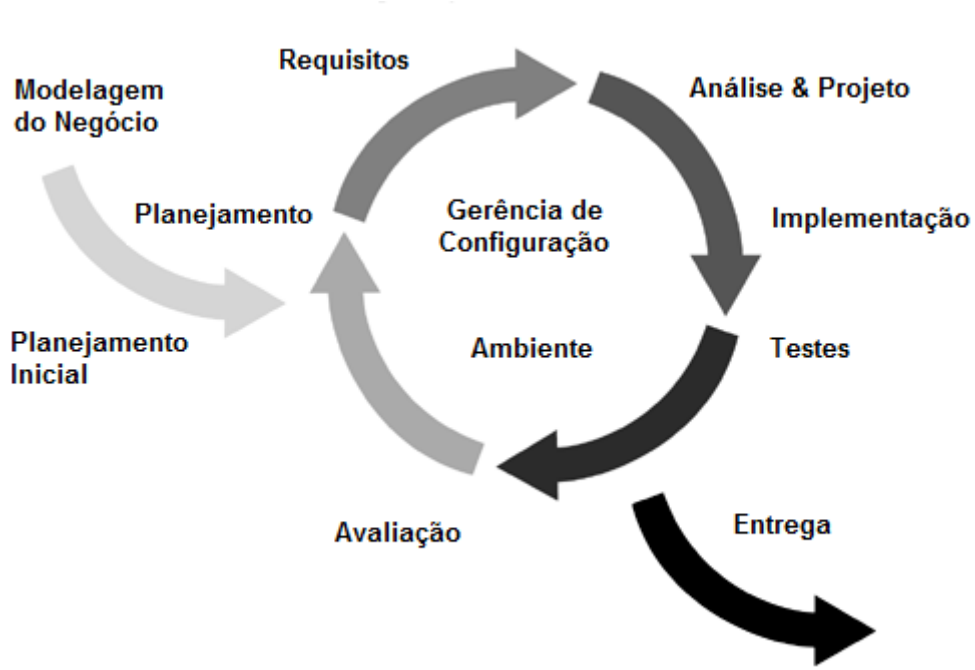


Figura 2.8 – Desenvolvimento Iterativo no RUP (adaptado de (KROLL; KRUCHTEN, 2003)).

No que se refere às fases, o RUP considera quatro fases no desenvolvimento de software:

- Concepção: visa estabelecer um bom entendimento do escopo do projeto, obtendo um entendimento de alto nível dos requisitos a serem tratados (KROLL; KRUCHTEN, 2003). Nesta fase o foco está na comunicação com o cliente para a identificação de requisitos e nas atividades de planejamento. No que se refere às atividades do processo de desenvolvimento, o foco é o levantamento de requisitos, ainda que atividades de modelagem conceitual (análise) e para a elaboração de um esboço bastante inicial da arquitetura do sistema (projeto) possam ser realizadas. Protótipos podem ser construídos para apoiar a comunicação com o cliente.
- Elaboração: os objetivos desta fase são analisar o domínio do problema, estabelecer a arquitetura do sistema, refinar o plano do projeto e identificar seus maiores riscos (KRUCHTEN, 2003). Assim, em termos do processo de desenvolvimento, o foco são as atividades de análise e projeto.
- Construção: envolve o projeto detalhado de componentes, sua implementação e testes. Nesta fase, os componentes do sistema são desenvolvidos, integrados e testados.
- Transição: como o próprio nome indica, o propósito desta fase é fazer a transição do sistema do ambiente de desenvolvimento para o ambiente de produção. São feitos testes de sistema e de aceitação e a entrega do sistema aos seus usuários.

Cada fase, por sua vez, pode envolver um número arbitrário de iterações, dependendo das características do projeto. Além disso, todo o conjunto de fases também pode ser realizado de maneira incremental, em ciclos.

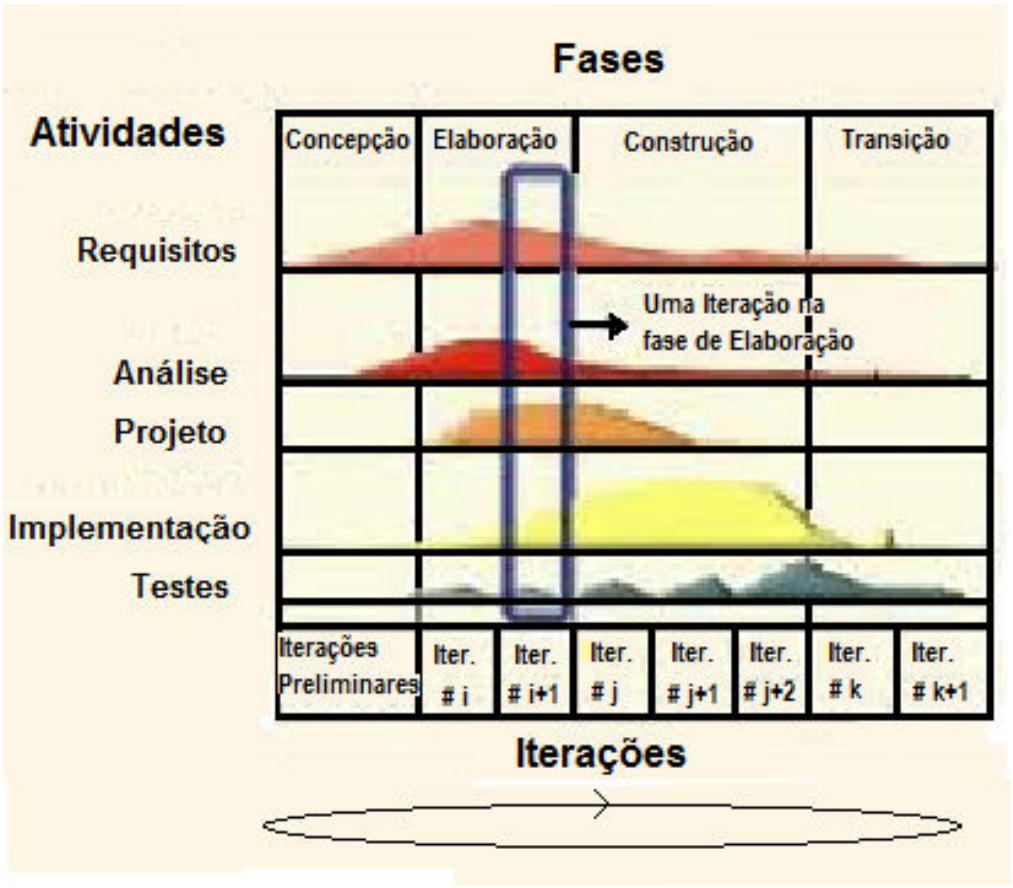


Figura 2.9 - O Modelo de Processo do RUP.

As atividades do processo de desenvolvimento são distribuídas ao longo de uma iteração, em função do foco da fase correspondente. Por exemplo, em uma iteração que ocorra no final da fase de elaboração, tipicamente são realizadas atividades de especificação de requisitos, análise, projeto, implementação e testes. Já em uma iteração típica da fase de concepção, essencialmente são realizadas atividades de levantamento de requisitos, com algum trabalho de modelagem (análise). Eventualmente, se um protótipo for desenvolvido, pode haver algum trabalho de implementação e testes.

Leitura Complementar

Os livros de Pressman (2011), Sommerville (2011) e Pfleeger (2004) são excelentes referência na área de Engenharia de Software e contemplam os aspectos tratados neste capítulo e muito mais. Modelos de processo são discutidos em (PRESSMAN, 2011) no Capítulo 2 – Modelos de Processo; em (SOMMERVILLE, 2011) no Capítulo 2 – Processos de Software; e em (PFLEEGER, 2004) no Capítulo 2 – Modelagem do Processo e Ciclo de Vida.

Referências do Capítulo

CHRISTENSEN, M.J., THAYER, R.H., *The Project Manager's Guide to Software Engineering Best Practices*, Wiley-IEEE Computer Society Press, 2002.

KROLL, P., KRUCHTEN, P., *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Addison Wesley, 2003.

KRUCHTEN, P., *The Rational Unified Process: An Introduction*, 3rd Edition, Addison Wesley, 2003.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, 2ª Edição, São Paulo: Prentice Hall, 2004.

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

SOMMERVILLE, I., *Engenharia de Software*, 9ª Edição. São Paulo: Pearson Prentice Hall, 2011.

Capítulo 3 – Requisitos de Software

Em um desenvolvimento de software, a primeira coisa a ser feita é capturar os requisitos que o sistema a ser desenvolvido tem de tratar. Um entendimento dos requisitos do software é essencial para o sucesso de um projeto de desenvolvimento de software. Os requisitos devem ser inicialmente levantados e descritos de maneira sucinta para permitir definir o escopo do sistema. Depois, os requisitos devem ser refinados em detalhes, as funções e o desempenho do software devem ser especificados e as interfaces e restrições que o software deve atender devem ser estabelecidas. Modelos dos dados e do comportamento do sistema devem ser elaborados e os principais artefatos produzidos devem ser submetidos à avaliação da qualidade. Os requisitos são o objeto de estudo deste capítulo.

3.1 –Requisitos e Tipos de Requisitos

Uma das principais medidas do sucesso de um sistema de software é o grau no qual ele atende aos requisitos para os quais foi construído. Os requisitos de um sistema de software incluem descrições das funções que o sistema deve prover e das restrições que devem ser satisfeitas. Em outras palavras, os requisitos definem o que o sistema deve fazer e as circunstâncias sob as quais deve operar (SOMMERVILLE, 2011).

Requisitos são, normalmente, classificados em requisitos funcionais e não funcionais. Requisitos funcionais, como o próprio nome indica, apontam as funções que o sistema deve prover. Já os requisitos não funcionais descrevem restrições sobre as funções oferecidas, tais como restrições de tempo, de uso de recursos etc. De maneira geral, requisitos não funcionais de produtos de software referem-se a atributos de qualidade que o sistema deve apresentar, tais como confiabilidade, usabilidade, eficiência, portabilidade, manutenibilidade e segurança. Vale a pena mencionar que alguns requisitos não funcionais dizem respeito ao sistema como um todo e não a uma funcionalidade específica. Por exemplo, pode ser um requisito de todo um sistema de software ser de fácil manutenção.

Além das classes de requisitos funcionais e não funcionais, é importante considerar também requisitos de domínio. Requisitos de domínio (ou regras de negócio) são provenientes do domínio de aplicação do sistema e refletem características e restrições desse domínio. Eles são derivados do negócio que o sistema se propõe a apoiar e podem restringir requisitos funcionais existentes ou estabelecer como cálculos específicos devem ser realizados, refletindo fundamentos do domínio de aplicação (SOMMERVILLE, 2011). Por exemplo, em um sistema de matrícula de uma universidade, uma importante regra de negócio diz que um aluno só pode se matricular em uma turma de uma disciplina se ele tiver cumprido seus pré-requisitos.

3.2 – Engenharia de Requisitos

Dada a importância de requisitos para o sucesso de um projeto, eles devem ser cuidadosamente identificados, analisados, documentados e avaliados. Além disso, alterações em requisitos devem ser gerenciadas para garantir que estão sendo adequadamente tratadas.

Ao conjunto de atividades relacionadas aos requisitos, dá se o nome de Engenharia de Requisitos. De maneira geral, o processo de Engenharia de Requisitos envolve as seguintes atividades (KOTONYA; SOMMERVILLE, 1998), como ilustra a Figura 3.1:

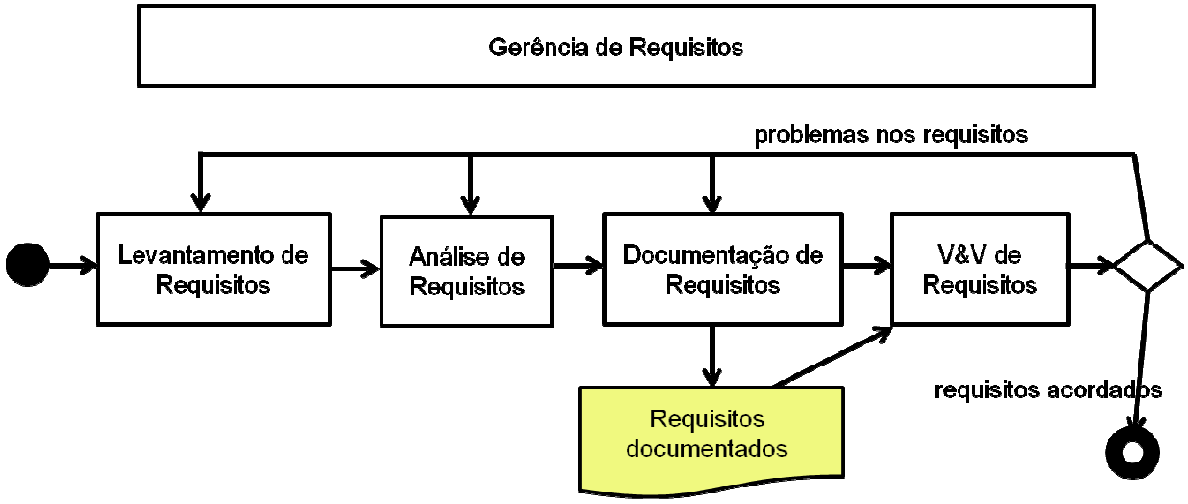


Figura 3.1 - O Processo da Engenharia de Requisitos

- Levantamento de Requisitos: Nesta fase, os usuários, clientes e especialistas de domínio são identificados e trabalham junto com os engenheiros de requisitos para entender a organização, o domínio da aplicação, os processos de negócio a serem apoiados, 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 de melhoria, restrições existentes e problemas a serem resolvidos devem ser levantados.
- Análise de Requisitos: visa estabelecer um conjunto acordado de requisitos consistentes e sem ambiguidades, que possa ser usado como base para as atividades subsequentes do processo de software. Para tal, diversos tipos de modelos são construídos. Assim, a análise de requisitos é essencialmente uma atividade de modelagem. A análise de requisitos pode incluir, ainda, negociação entre usuários para resolver conflitos detectados.
- Documentação de Requisitos: é a atividade de representar os resultados da Engenharia de Requisitos em um documento (ou conjunto de documentos), contendo os requisitos do software e os modelos que os especificam.
- Verificação e Validação de Requisitos: A verificação de requisitos avalia se os requisitos estão sendo tratados 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 os requisitos do sistema estão corretos, ou seja, se os requisitos e modelos documentados atendem às reais necessidades de usuários e clientes.
- Gerência de Requisitos: se preocupa em gerenciar as mudanças nos requisitos já acordados, manter uma trilha dessas mudanças e gerenciar os relacionamentos entre os requisitos e as dependências entre requisitos e outros artefatos produzidos no processo de software, de forma a garantir que mudanças nos requisitos sejam feitas de maneira controlada e documentada.

Em relação ao processo de software, das cinco atividades do processo de Engenharia de Requisitos anteriormente listadas, apenas as três primeiras fazem parte do processo de desenvolvimento (Levantamento, Análise e Documentação de Requisitos). As duas últimas são, na realidade, atividades de gerência envolvendo requisitos. A atividade de Verificação e Validação de Requisitos é uma importante atividade do processo de Gerência da Qualidade e um instrumento fundamental para a garantia do projeto como um todo, tendo em vista que os requisitos são a base para o sucesso de um projeto de software. A atividade de Gerência de Requisitos pode ser vista como a Gerência de Configuração aplicada a requisitos. A Figura 3.2 procura ilustrar o posicionamento das atividades da Engenharia de Requisitos no processo de software.

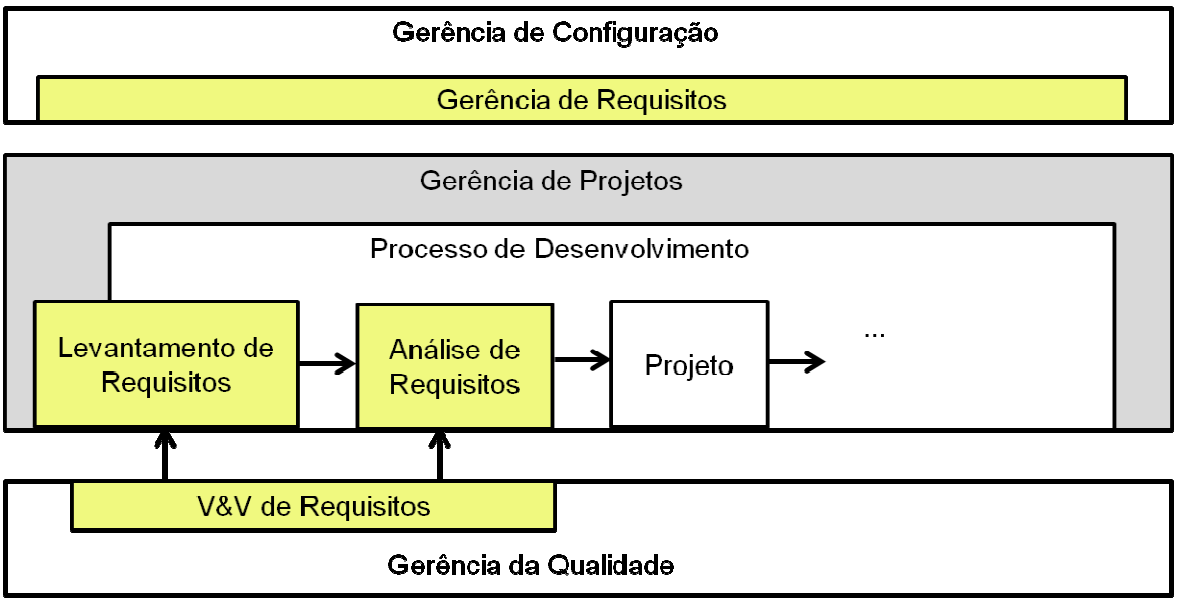


Figura 3.2 - As Atividades da Engenharia de Requisitos no Processo de Software.

Uma vez que o foco desta primeira parte das notas de aula é o processo de desenvolvimento, as atividades de levantamento e análise de requisitos são estudadas com maiores detalhes. As demais atividades são discutidas na Parte II destas notas de aula.

3.3 – Levantamento de Requisitos

O levantamento de requisitos é uma atividade de descoberta de informações. Para poder capturar os requisitos de um sistema, é necessário compreender, dentre outros, o domínio de aplicação, os processos de negócio a serem apoiados pelo sistema, os problemas que se pretende resolver e necessidades e restrições dos envolvidos. Para se ganhar este entendimento, diversas fontes podem ser pesquisadas, dentre elas material bibliográfico, manuais de funcionamento da organização, documentos da organização e as pessoas envolvidas, dentre elas especialistas do domínio, clientes e usuários.

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. São vários os problemas enfrentados nesta fase, tais como: dificuldades para se estabelecer o escopo do sistema (o que deve ser tratado no desenvolvimento e o que não será tratado); problemas de comunicação entre pessoas e falta de entendimento acerca do assuntos

que cercam o desenvolvimento do sistema; volatilidade dos requisitos (requisitos mudando muito rapidamente); e falta de envolvimento dos principais interessados (especialistas de domínio, clientes e usuários). Assim, para tentar minimizar essas dificuldades, estabelecer uma relação de confiança com clientes e usuários e melhor aproveitar os esforços, 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.

Uma característica fundamental da atividade de levantamento de requisitos é o seu enfoque em uma visão do cliente / usuário. Em outras palavras, está-se olhando para o sistema a ser desenvolvido por uma perspectiva externa. Ainda não se está procurando definir a estrutura interna do sistema, mas sim procurando saber que funcionalidades o sistema deve oferecer ao usuário e que restrições o sistema devem satisfazer ou garantir.

Os resultados do levantamento de requisitos devem ser documentados em um documento de requisitos. Normalmente, este documento tem um formato pré-definido pela organização, contendo, dentre outros, listas de requisitos funcionais, não funcionais e regras de negócio, descritos em linguagem natural.

3.4 – Análise de Requisitos

A análise de requisitos (muitas vezes chamada análise de sistemas), por outro lado, enfoca o que o sistema tem de ter para tratar adequadamente os requisitos levantados. Assim sendo, a análise de requisitos 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. Um bom exemplo de modelos são os mapas. Como ilustra a Figura 3.3, há mapas, por exemplo, que focalizam a estrutura política de um estado, enquanto outros podem enfocar aspectos relacionados a turismo neste mesmo estado. No exemplo da Figura 3.3, a entidade sendo representada é a mesma: o estado do Espírito Santo, contudo, as perspectivas de atenção são diferentes.

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

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

Mapa Político

Mapa Turístico

Figura 3.3 - Modelos como Abstrações da Realidade.

Um modelo enfatiza um conjunto de características da realidade, que corresponde à perspectiva do modelo. No desenvolvimento de sistemas, há duas perspectivas principais:

- Estrutural: tem por objetivo descrever as informações que o sistema deve representar e gerenciar;
- Comportamental: visa especificar as ações (funcionalidades / serviços) que o sistema deve prover, bem como o comportamento de certas entidades do modelo estrutural em relação a essas ações.

Além da perspectiva que um modelo enfatiza, modelos de sistemas podem ser construídos em diferentes níveis de abstração (foco no problema ou na solução). Geralmente, no desenvolvimento de sistemas, são considerados três níveis:

- *Modelo conceitual*: considera características do sistema independentes do ambiente computacional (hardware e software) no qual o sistema será implementado. Modelos conceituais são construídos na atividade de análise de requisitos.
- *Modelo lógico*: trata características dependentes de um determinado *tipo* de plataforma computacional. Essas características são, contudo, independentes de produtos específicos. Tais modelos são típicos da fase de projeto.
- *Modelo físico*: leva em consideração características dependentes de uma plataforma computacional específica, isto é, uma linguagem e um compilador específicos, 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 detalhado quanto na fase de implementação.

Conforme apontado anteriormente, nas primeiras etapas do processo de desenvolvimento (levantamento de requisitos e análise), o engenheiro de software representa o sistema através de modelos conceituais. Em etapas posteriores (projeto e implementação), as características lógicas e físicas são representadas em outros 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, os quais são classificados em classes. 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, discutimos as atividades do processo de desenvolvimento de software à luz do paradigma estruturado. Assim sendo, a atividade de análise de requisitos é conduzida tomando por base esse paradigma.

3.5 – Métodos de Análise de Requisitos Segundo o Paradigma Estruturado

Para a realização das atividades de análise, métodos e técnicas podem ser empregados, visando guiar o trabalho. Há diversos métodos e técnicas de análise estruturada. Dentre os métodos, destacam-se: Análise Estruturada de Sistemas (GANE; SARSON, 1983; DE MARCO, 1983), Análise Estruturada Moderna (YOURDON, 1990) e Análise Essencial de Sistemas (MCMENAMIN; PALMER, 1991).

De maneira geral, todos esses métodos têm em comum a orientação a processos e a aplicação da técnica de Modelagem de Fluxos de Dados. Esta técnica foi proposta no contexto dos métodos precursores de Análise Estruturada (GANE; SARSON, 1983; DE MARCO, 1983). Uma característica marcante da aplicação da técnica de Modelagem de Fluxos de Dados na Análise Estruturada era a abordagem *top-down* para a decomposição funcional de sistemas: partir da visão geral do sistema e ir descendo, em uma visão hierárquica, a níveis de detalhes cada vez maiores.

Posteriormente, outras técnicas, como a Modelagem de Entidades e Relacionamentos (CHEN, 1990) e a Modelagem de Estados, foram incorporadas à análise estruturada, dando origem à Análise Estruturada Moderna (YOURDON, 1990).

Por fim, a Análise Essencial de Sistemas (MCMENAMIN; PALMER, 1991) é uma evolução da Análise Estruturada Moderna, que passou a endereçar algumas das principais dificuldades que os analistas enfrentavam com a aplicação da Análise Estruturada Moderna: a dificuldade de capturar requisitos nos estágios iniciais do desenvolvimento, a capacidade de distinguir entre requisitos verdadeiros e falsos, e uma abordagem mais eficiente para dividir o sistema em partes independentes, de modo a facilitar o processo de análise.

O método da Análise Essencial de Sistemas preconiza que, de uma forma geral, um sistema deve ser modelado através de três dimensões (POMPILHO, 1995):

- *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.

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 (POMPILHO 1995). De fato, com essa abordagem, a análise essencial passava a abranger também o levantamento de requisitos, uma vez que a Análise Estruturada Moderna focava somente na análise de requisitos. A identificação de eventos passou a ser usada também para a decomposição funcional dos sistemas.

Atualmente, praticamente nenhum desses métodos é aplicado integralmente na prática. Contudo, as técnicas por eles empregadas são ainda bastante usadas, muitas vezes combinadas com outras técnicas mais modernas de especificação de sistemas, como a Modelagem de Casos de Uso. Assim, neste texto são abordadas algumas das principais técnicas de modelagem de sistemas aplicadas no desenvolvimento de sistemas segundo o paradigma estruturado, a saber: Modelagem de Entidades e Relacionamentos, Modelagem de Casos de Uso, Modelagem de Fluxos de Dados e Modelagem de Estados.

3.6 - Modelagem de Entidades e Relacionamentos

A Modelagem de Entidades e Relacionamentos 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 (CHEN, 1990; SETZER, 1987).

Tipicamente, um Modelo de Entidades e Relacionamentos (MER) é composto por um conjunto de Diagramas de Entidades e Relacionamentos e um Dicionário de Dados.

3.6.1 – 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, **tipos de entidades**, isto é, grupos de entidades que têm características semelhantes. No modelo ER, tipos (ou conjuntos) de entidades são representados graficamente por retângulos, como mostra a Figura 3.4.

Figura 3.4 – Representação Gráfica de Conjuntos de Entidades.

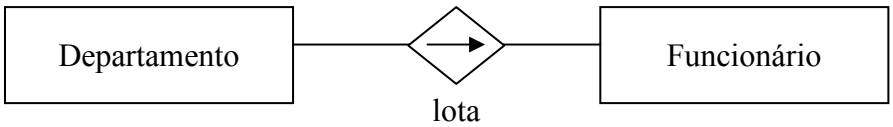
Um conjunto de entidades representa todos os elementos do mundo real referidos pelo conjunto. Por exemplo, em um sistema de uma biblioteca, o conjunto de entidades Livro representa todos os livros de uma biblioteca.

Para estabelecermos uma padronização, neste texto, usaremos nomes de tipos de entidades sempre no singular e escritos com a primeira letra em maiúsculo. No entanto, isto não representa efetivamente uma regra. Além disso, usaremos o termo entidade para referenciar tanto entidades quanto tipos 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 tipo de entidade 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 tipo Funcionário) está lotado (um relacionamento) no departamento de Vendas (uma entidade do tipo Departamento). Um relacionamento binário, como o citado no exemplo anterior, é uma representação abstrata da associação entre duas entidades.

Da mesma forma que as entidades, estamos mais interessados em modelar **tipos de relacionamentos**. Um tipo (ou 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 3.5. 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 existe entre conjuntos de entidades. Opcionalmente, usaremos o termo instância (tanto de entidades quanto de relacionamentos) para referenciar um elemento do tipo (de entidades e de relacionamentos, respectivamente).



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

Figura 3.5 – 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 3.6.

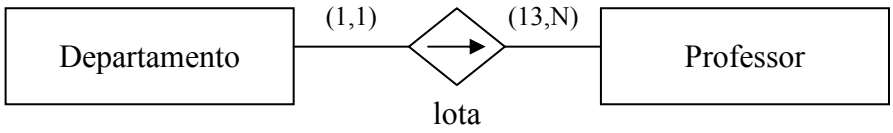


Figura 3.6 – Representação de cardinalidades em relacionamentos.

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. Por outro lado, ao se criar um novo departamento, ter-se-á de informar pelo menos 13 professores que nele serão lotados.

É importante frisar que, entre duas entidades, podem existir vários tipos de relacionamentos diferentes, como mostra o exemplo da Figura 3.7.

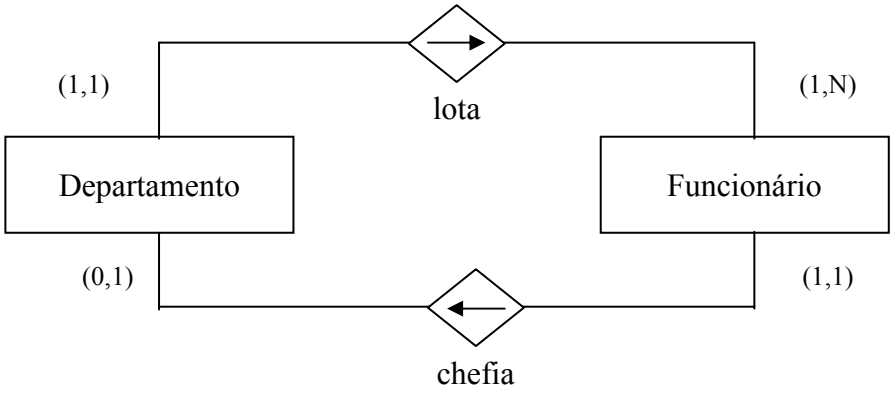


Figura 5.7 – Dois relacionamentos diferentes entre as mesmas entidades.

Além disso, uma entidade pode participar de relacionamentos com quaisquer outras entidades do modelo, inclusive com ela mesma (auto-relacionamento), como mostra a Figura 3.8.

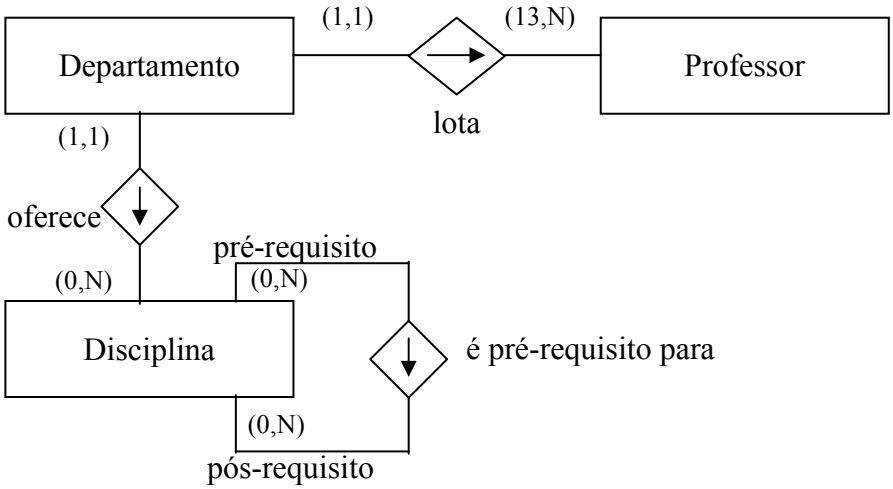


Figura 3.8 – Exemplo de auto-relacionamento.

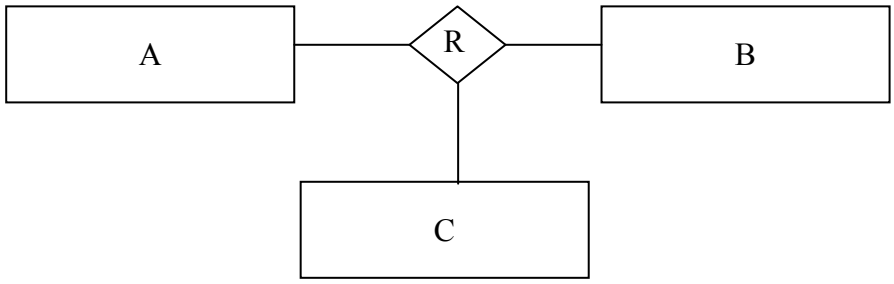
No caso de auto-relacionamentos, é útil distinguir qual a atuação de cada elemento do conjunto de entidades no relacionamento e, portanto, é importante atribuir *papéis*. Assim no caso do auto-relacionamento *é pré-requisito para* da Figura 3.8, estamos dizendo que:

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 bem menos corriqueiros. Um relacionamento ternário, por exemplo, só se caracteriza pelo terno, como mostra a Figura 3.9. 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 3.10. 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 3.9 – Relacionamento Ternário.

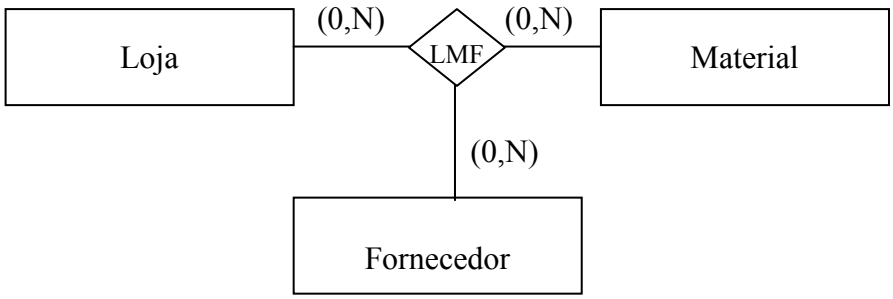


Figura 3.10 – Exemplo de Relacionamento Ternário.

Alguns relacionamentos são tão importantes que assumem o *status* de entidades. No modelo ER, esses relacionamentos são chamados de **entidades associativas** (ou agregados). Assim, uma entidade associativa é uma abstração através da qual um tipo de relacionamentos entre duas entidades é tratado como um tipo de entidades em um nível mais alto. Essa "nova entidade", a entidade associativa, pode, então, relacionar-se com outras entidades do modelo, como mostra a Figura 3.11.

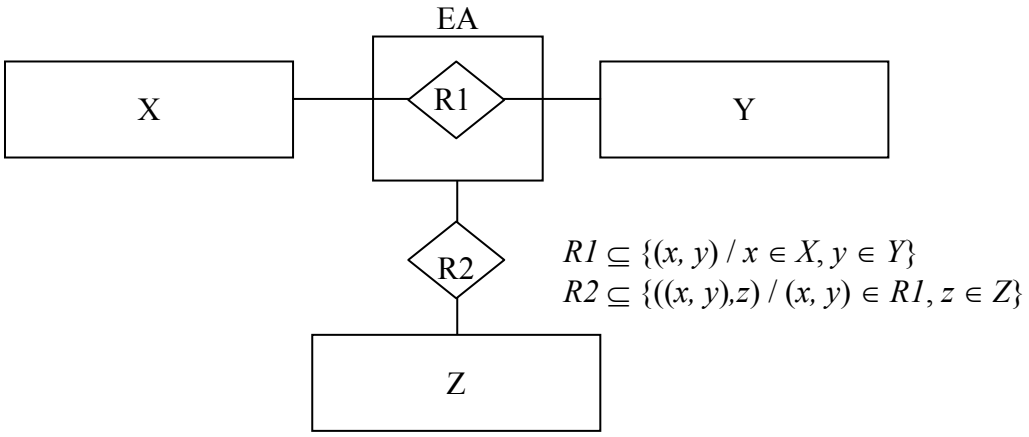


Figura 3.11 – Entidade Associativa

A Figura 3.12 mostra um exemplo de entidade associativa. Nesse exemplo, o tipo de relacionamento Consulta assume o status de um tipo de entidade. Uma consulta pode ser paga, então, por um Plano de Saúde.

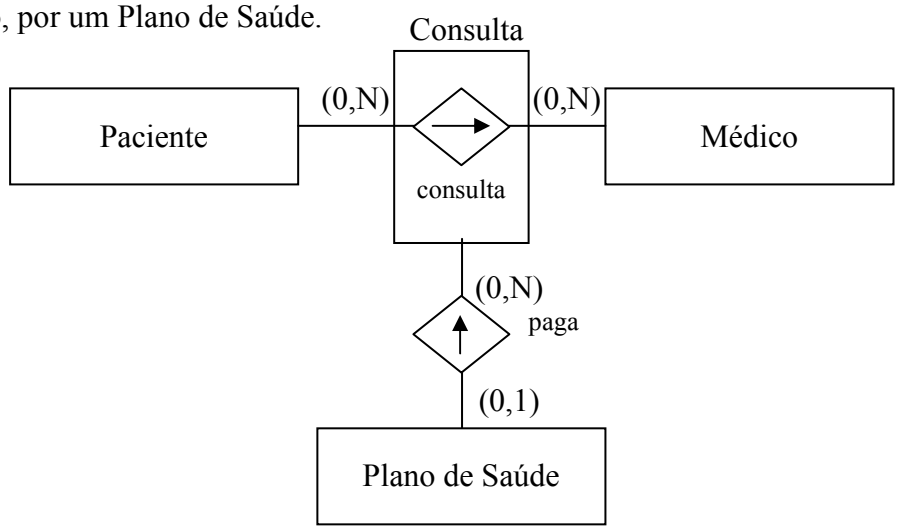


Figura 3.12 – Exemplo de Entidade Associativa.

É importante observar que entidades associativas, muitas vezes, capturam eventos que ocorrem no mundo real e que precisam ser registrados no sistema. Assim, uma opção até mais clara é representar esse evento como um tipo de entidades, relacionado aos demais, como ilustra a Figura 3.13.

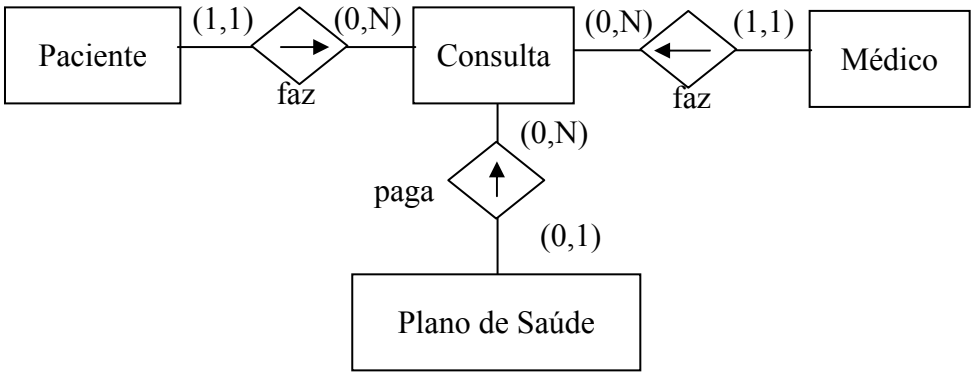


Figura 3.13 – Evento representado como um Tipo de Entidade.

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 3.14 mostra um exemplo de particionamento.

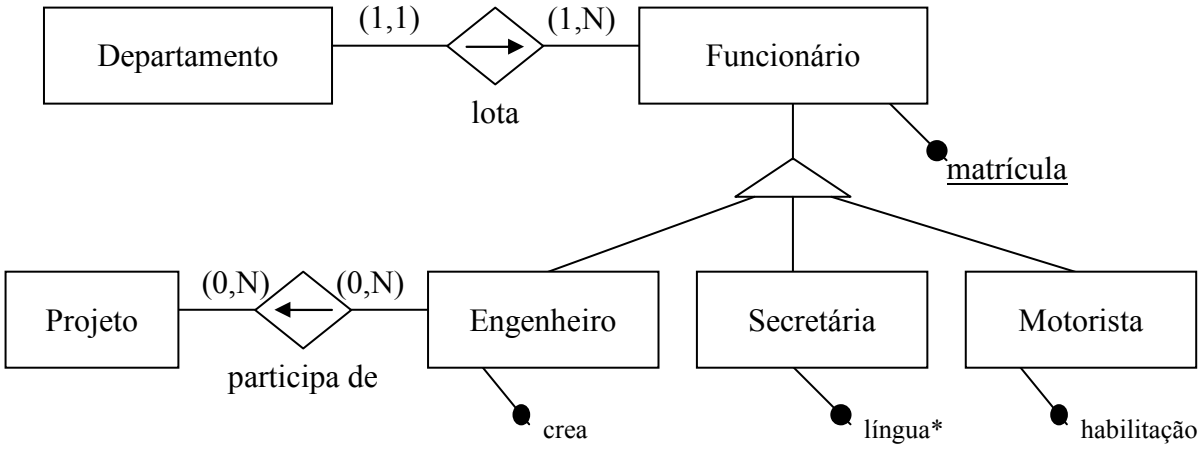


Figura 3.14 – Particionamento de um Conjunto de Entidades.

Atributos

Os atributos são utilizados para descrever características ou propriedades relevantes de entidades e de relacionamentos. 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. P.ex., 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 uma entidade do conjunto é dito ser determinante. Atributos determinantes são sublinhados, como forma de destaque.

A Figura 3.15 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.

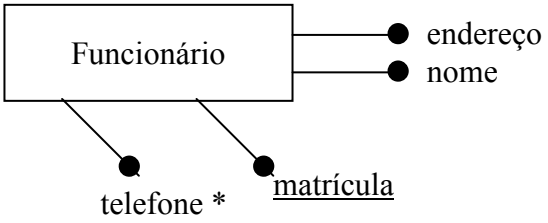


Figura 3.15 – Representação gráfica para atributos.

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 3.16.

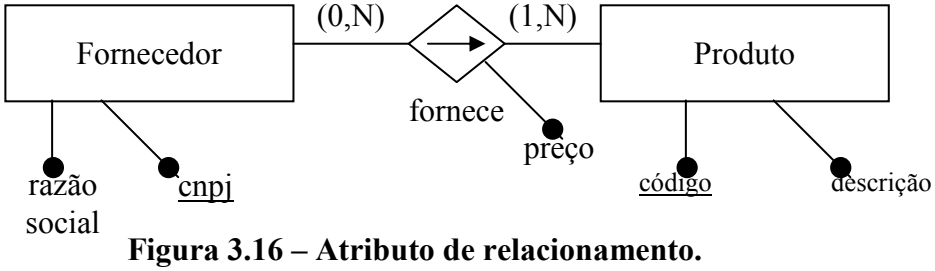


Figura 3.16 – Atributo de relacionamento.

Há um teste que pode ser aplicado para se deduzir se um atributo é de um dos dois tipos de entidades ou se é do relacionamento. Seja o exemplo da Figura 3.17.

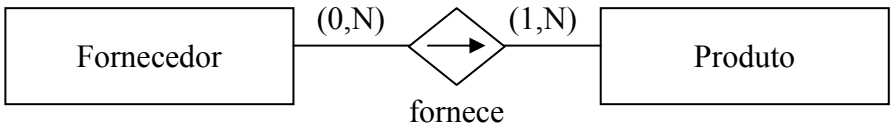


Figura 3.17 – Relacionamento com atributos.

1. Fixa-se um produto, p.ex. Impressora XPY, e variam-se os fornecedores desse produto. Evidentemente o valor do atributo pode mudar. Por exemplo, a Casa do Analista vende a Impressora XPY 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 a impressora XPY por R$ 350 e um notebook AWS 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 tipos de entidades, como mostra a Figura 3.18.

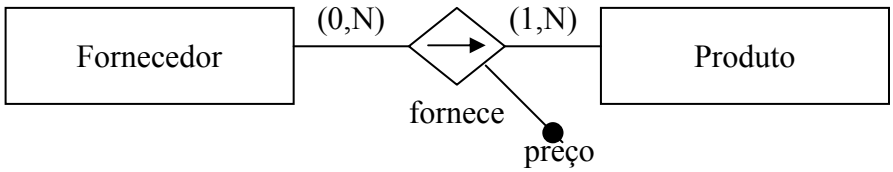


Figura 3.18 – Atributo do Relacionamento.

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 o sistema;
- 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 3.19.

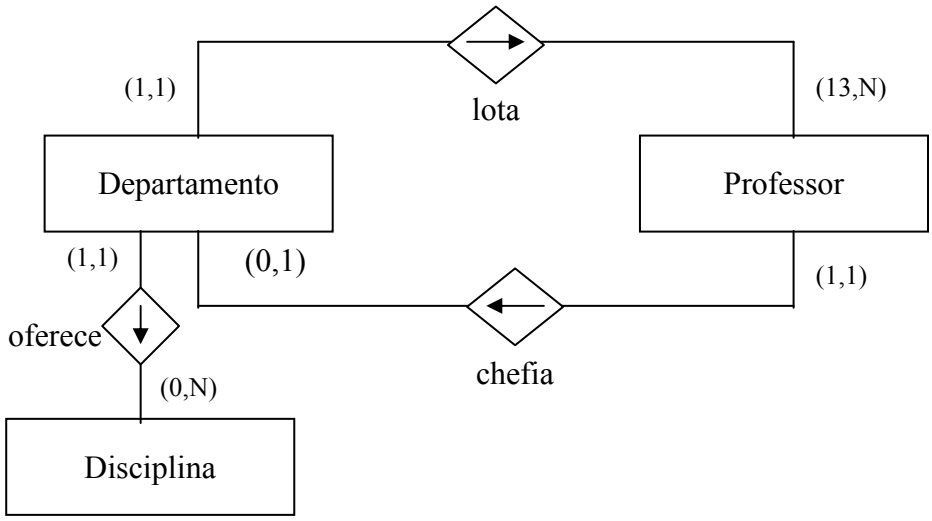


Figura 3.19 – Departamento como Nova Entidade.

3.6.2 - 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 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. Entretanto, há outras situações em que não conseguimos capturar tais restrições. Seja o exemplo da Figura 3.20.

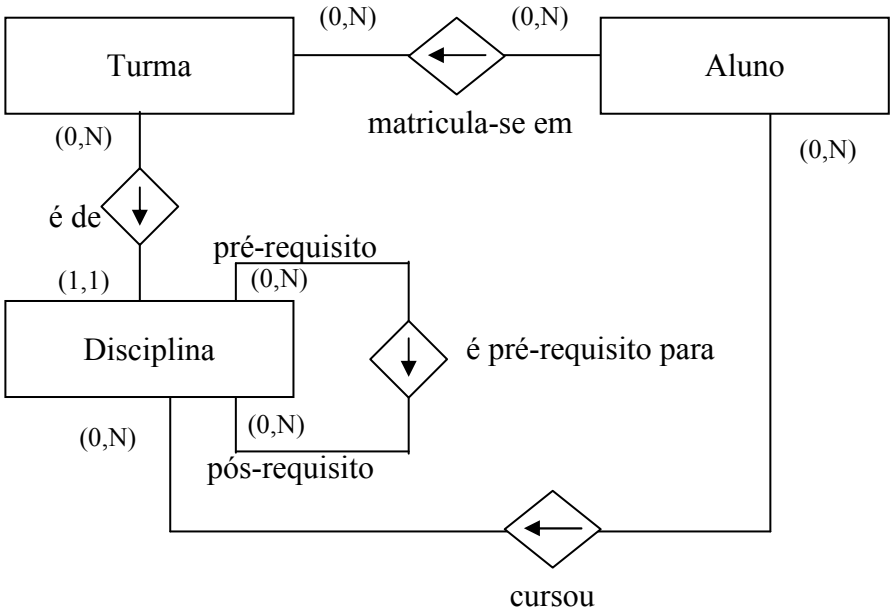


Figura 3.20 – Exemplo de ocorrência de restrições de integridade.

Nesse 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 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.

3.6.3 – 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 dos itens de dados manipulados pelo sistema. A Figura 3.21 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 3.21 – 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 = 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]

3.7 - Modelagem de Fluxos de Dados

Uma perspectiva igualmente importante para a análise dos requisitos de um sistema é aquela que considera as funções que o sistema deverá executar para atender às necessidades de seus usuários. A Modelagem de Fluxos de Dados é a técnica mais empregada para este fim pelos métodos de análise segundo o paradigma estruturado.

Diagramas de Fluxos de Dados (DFDs) são a parte gráfica de um Modelo de Fluxos de Dados. DFDs permitem representar 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 3.22 mostra a notação usada por Yourdon (1990), que será a adotada neste texto.

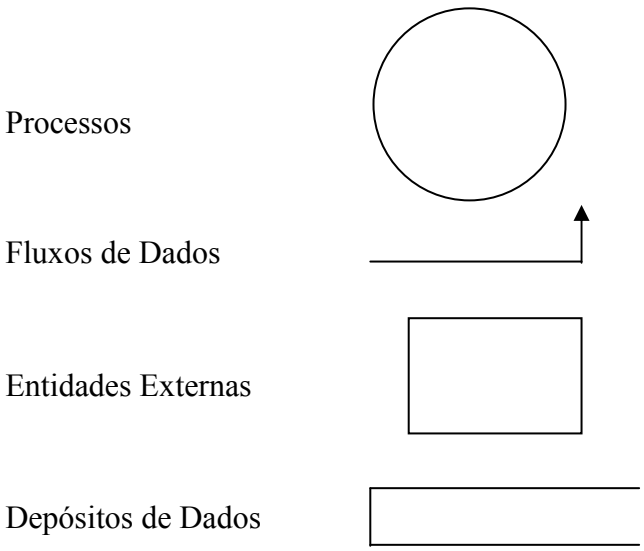


Figura 3.22 – Notação básica para construção de DFDs.

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

- Dicionário de Dados: adicionar no dicionário produzido para o Modelo ER um entrada para cada fluxo de dados presente nos DFDs;
- 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 fluxo de dados de entrada no conteúdo de um fluxo de dados de saída, sem armazenamento interno no sistema (Figura 3.23);

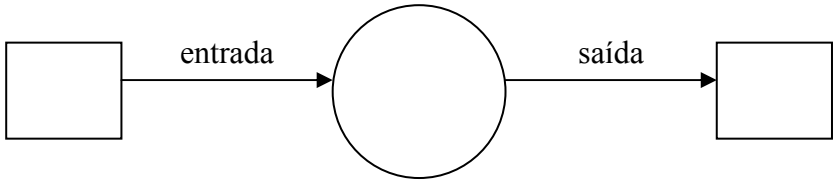


Figura 3.23 – Transformações de dados.

- inserções ou modificações do conteúdo de dados armazenados, a partir do conteúdo (possivelmente transformado) de dados de entrada, como mostra a Figura 3.24;

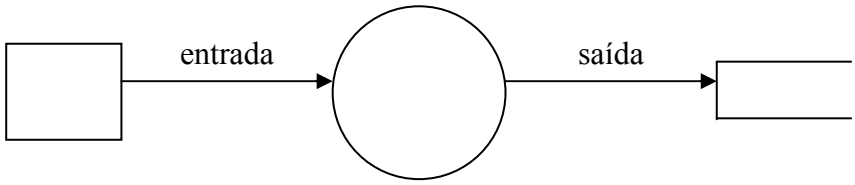


Figura 3.24 – Armazenamento de dados.

- transformações de dados previamente armazenados no conteúdo de um fluxo de dados de saída, como mostra a Figura 3.25.

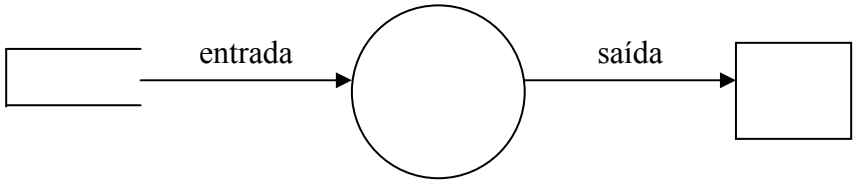


Figura 3.25 – Geração de dados de saída a partir de dados armazenados.

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 ambiguidades. Devem ser evitados nomes muito físicos (p. ex., gravar, imprimir etc) ou muito técnicos (p. ex., apagar, fazer backup etc.).

Toda transformação de dados deve ser representada e, deste modo, não se admite ligação direta entre entidades externas e depósitos de 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. 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 três 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.

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 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 3.26, 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.

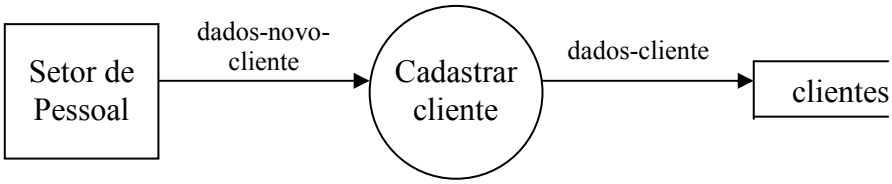


Figura 3.26 – Mesmo conteúdo de dados em fluxos diferentes.

Fluxos de dados que transportam exatamente o mesmo conteúdo de/para um depósito de dados, não precisam ser nomeados. No exemplo da Figura 3.26, se o fluxo *dados-cliente* apresentar exatamente o mesmo conteúdo do depósito *clientes*, não há necessidade de nomeá-lo, como mostra a Figura 3.27.

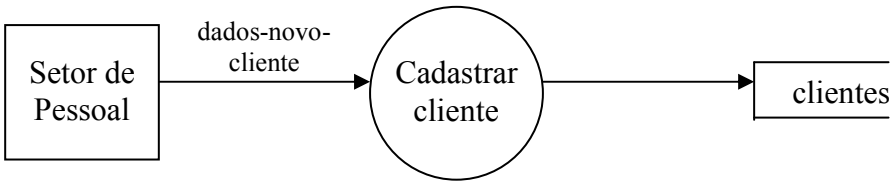


Figura 3.27 – Fluxo de dados não nomeado.

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.

É importante realçar que DFDs não indicam a sequencia na qual fluxos de dados entram ou saem de um processo. Essa sequencia 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. Representam um local de armazenamento de dados entre processos.

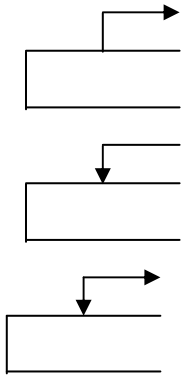
Um depósito de dados é representado por um retângulo sem a linha lateral direita, com um nome e um identificador (opcional) em seu interior. Às vezes, para evitar o cruzamento de linhas de fluxos de dados ou para impedir que longas linhas de fluxos de dados saiam de um lado para outro do diagrama, um 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 3.28.

Figura 3.28 – 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 3.29.



Leitura não destrutiva do conteúdo do depósito de dados.

Inclusão, exclusão ou alteração do conteúdo do depósito de dados.

Leitura e modificação do conteúdo do depósito de dados.

Figura 3.29 – Semântica dos acessos a depósitos de dados em um DFD.

É importante 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 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 usuário), 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 uma mesma entidade externa 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 3.30.

Figura 3.30 – 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, em algum momento, 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 as entidades externas, como o próprio nome indica, são externas ao sistema, os fluxos de dados que as interligam aos diversos processos representam a interface entre o sistema e o mundo externo.

3.7.1 - 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 as 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 3.31 mostra a representação gráfica desses acessos.

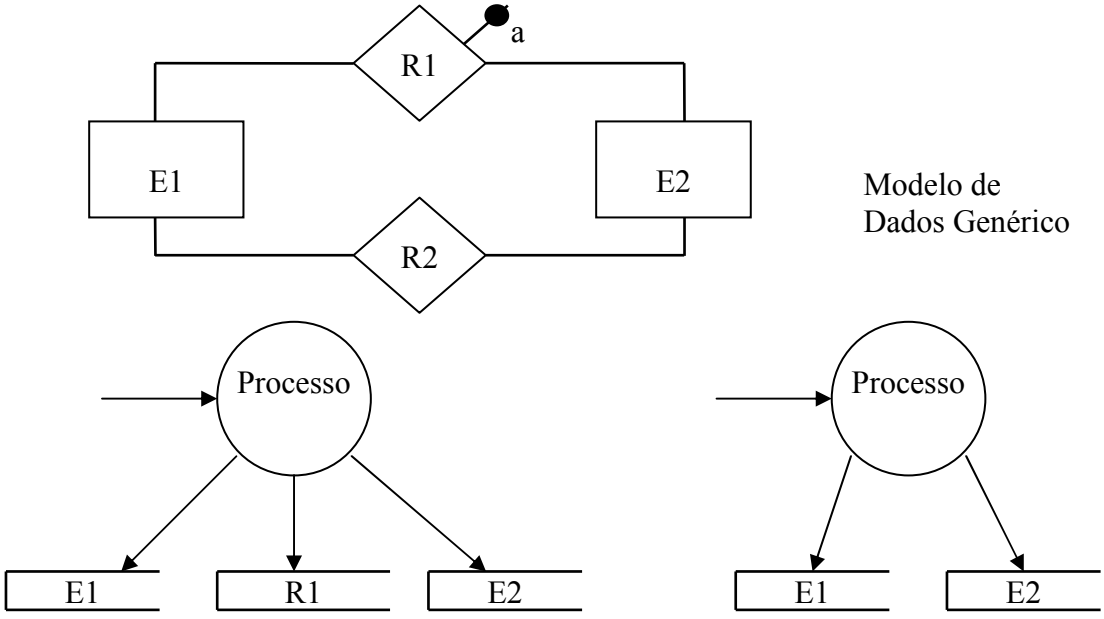


Figura 3.31 – Acessos a depósitos de dados.

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.

3.7.2 - Construindo DFDs

Uma boa opção para o uso de DFDs consiste em elaborar um DFD para cada funcionalidade identificada no levantamento de requisitos. As funcionalidades podem ser classificadas em fundamentais, aquelas que visam apoiar diretamente os principais processos de negócio da organização, e custodiais, aquelas necessárias para prover as informações necessárias para a realização das funcionalidades fundamentais (p.ex., cadastros).

Se um processo oriundo de uma funcionalidade for complexo demais, ele pode ser decomposto em outros processos, de modo a 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. Se tal nível de complexidade for superado, devemos utilizar uma das seguintes técnicas para decompor o DFD:

- Fissão: o processo complexo é substituído no próprio DFD por um número de processos mais simples.
- 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

Recomenda-se o uso da fissão quando a leitura do diagrama não ficar prejudicada pelo aparecimento de mais alguns processos em um DFD. 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. Evite desenhar DFDs complexos.
3. Cuidado com os processos sem fluxos de dados de entrada ou de saída.
4. Cuidado com os depósitos de dados que só possuem fluxos de dados de entrada ou de saída.
5. Depósitos de dados permanentes devem manter estreita relação com os conjuntos de entidades e de relacionamentos do modelo ER.
6. 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.

7. 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.
8. Não represente no DFD fluxos de controle ou de material. Como o nome indica, DFDs representam fluxos de dados.
9. Só especifique a lógica de processos primitivos, ou seja, aqueles que não são detalhados em outros diagramas.

3.7.3 - 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: (i) uso de expressões do tipo "mas, todavia, a menos que"; (ii) uso de comparativos (maior que / menor que, mais de / menos de); (iii) ambiguidades inerentes a frases concatenadas com "e/ou"; (iv) uso de adjetivos indefinidos (p.ex., bom, mau).

Para administrar os problemas oriundos da narrativa, são utilizadas técnicas de especificação de processos, entre as quais podemos citar: Português Estruturado e Árvores de Decisão.

Português Estruturado

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

- **Instruções de Sequência**: 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;
```

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;

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

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

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. A Figura 3.32 mostra um exemplo de Árvore de Decisão. Se for necessário descrever a lógica de um processo como um conjunto de instruções, combinando decisões e ações intermediárias, a árvore de decisão pode ser combinada com português estruturado.

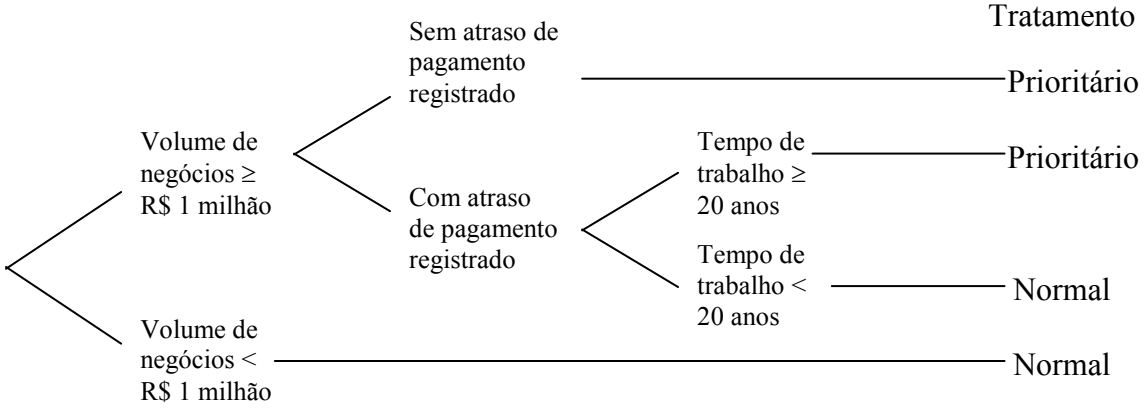


Figura 3.32 – Exemplo de Árvore de Decisão.

3.8 – Modelagem de Casos de Uso

Uma técnica alternativa para a modelagem funcional, amplamente utilizada nos dias de hoje é a Modelagem de Casos de Uso. A modelagem de casos de uso foi originalmente proposta no contexto de um método de análise orientada a objetos, o método OOSE (JACOBSON, 1992), mas percebeu-se que, na verdade, essa é uma técnica independente de paradigma, dado que não descreve a estrutura interna de um sistema, mas apenas provê uma forma de estruturar uma visão externa do mesmo.

O modelo de casos de uso é um modelo bastante simples, voltado para a comunicação com clientes e usuários e, portanto, bastante dependente de descrições textuais. De fato, como um modelo voltado para o levantamento de requisitos, sua principal finalidade é gerenciar a complexidade do domínio do problema, dividindo as funcionalidades do sistema em casos de uso, permitindo, assim, que as discussões e as correspondentes descrições textuais sejam feitas em contextos menores, mais específicos.

Um modelo de casos de uso é composto por um conjunto de Diagramas de Casos de Uso, geralmente um diagrama de casos de uso para cada subsistema, e de descrições para cada ator e para cada caso de uso identificado. As descrições de atores podem ser bastante simples, já que esses são externos ao sistema, enquanto as descrições de casos de uso são mais detalhadas, uma vez que representam funcionalidades que o sistema deve oferecer.

Como é possível deduzir pela visão geral dos modelos de casos de uso dada acima, os principais conceitos desse tipo de modelo são atores e casos de uso. De forma sucinta, um **ator** é um papel que um usuário, outro sistema ou dispositivo desempenha com respeito ao sistema. **Casos de uso** representam funcionalidades requeridas externamente. Uma associação entre um ator e um caso de uso significa que estímulos podem ser enviados entre atores e casos de uso. Os atores são conectados aos casos de uso somente por meio de associações. A associação entre um ator e um caso de uso indica que o ator e o caso de uso se comunicam entre si, cada um com a possibilidade de enviar e receber mensagens (BOOCH; RUMBAUGH; JACOBSON, 2006).

Nenhum sistema computacional existe isoladamente. Todo sistema interage com atores humanos ou outros sistemas, que utilizam esse sistema para algum propósito e esperam que o sistema se comporte de acordo com as maneiras previstas. Um caso de uso especifica um comportamento de um sistema segundo uma perspectiva externa e é uma descrição de um conjunto de sequências de ações realizadas pelo sistema para produzir um resultado de valor observável por um ator (BOOCH; RUMBAUGH; JACOBSON, 2006).

Em essência, um caso de uso é uma interação típica entre um ator (usuário humano, outro sistema computacional ou um dispositivo) e um sistema, que captura alguma função visível ao ator e, em especial, busca atingir uma meta do cliente.

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

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

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

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

Os casos de uso fornecem uma maneira para os engenheiros de software chegarem a uma compreensão comum acerca das funcionalidades do sistema com os usuários finais do sistema e com os especialistas do domínio. Além disso, servem para ajudar a validar e verificar o sistema à medida que ele evolui durante seu desenvolvimento. Assim, os casos de uso não apenas representam o comportamento desejado do sistema, mas também podem ser utilizados como base para a elaboração de casos de teste, principalmente nos testes de integração e de sistema (BOOCH; RUMBAUGH; JACOBSON, 2006).

Usualmente, em primeiro lugar, casos de uso são listados e discutidos, para só então se realizar alguma modelagem conceitual estrutural. Entretanto, em alguns casos, a modelagem conceitual estrutural (Modelo de Entidades e Relacionamentos) ajuda a descobrir casos de uso, razão pela qual essas atividades são conduzidas com alto grau de paralelismo.

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

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

3.8.1 - Diagramas de Casos de Uso

Diagramas de casos de uso especificam as funcionalidades que um sistema tem de oferecer, segundo diferentes perspectivas dos usuários. Em sua forma mais simples, um diagrama de casos de uso apresenta os dois elementos básicos: atores e casos de uso. A Figura 3.33 mostra a notação básica da Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML) (BOOCH; RUMBAUGH; JACOBSON, 2006) para diagramas de casos de uso.

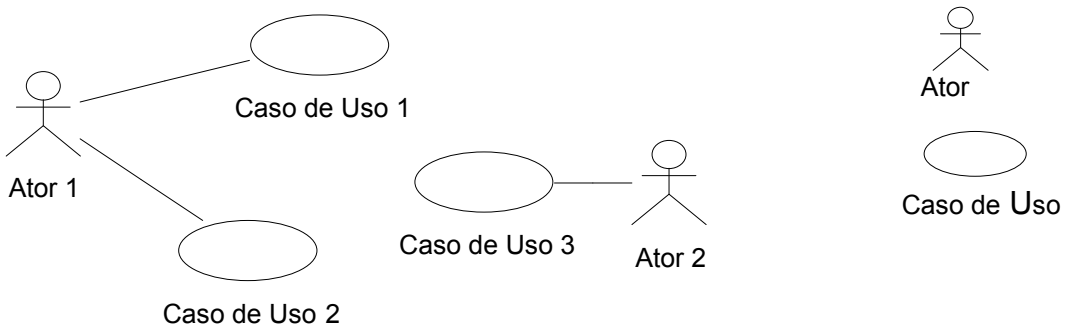


Figura 3.33 - Notação Básica da UML para Diagramas de Casos de Uso.

De fato, outros elementos de modelo podem ser usados em diagramas de casos de uso, tais como associações de dependência entre casos de uso, relações de generalização entre casos de uso e entre atores, mas fogem ao escopo deste texto. Maiores informações podem ser encontradas em (BOOCH; RUMBAUGH; JACOBSON, 2006).

3.8.2 - Descrição de Casos de Uso

Um caso de uso deve descrever *o que* um sistema faz. Geralmente, um diagrama de casos de uso é insuficiente para esse propósito. Assim, deve-se especificar o comportamento de um caso de uso pela descrição textual de seu fluxo de eventos, de modo que alguém de fora possa compreendê-lo. Ao escrever o fluxo de eventos, deve-se incluir como e quando o caso de uso inicia e termina, quando o caso de uso interage com os atores e quais são as informações transferidas e o fluxo básico e os fluxos alternativos do comportamento (BOOCH; RUMBAUGH; JACOBSON, 2006).

Uma vez que o conjunto inicial de casos de uso estiver estabilizado, cada um deles deve ser descrito em mais detalhes. Primeiro, deve-se descrever o fluxo de eventos principal (ou curso básico), isto é, o curso de eventos mais importante, que normalmente ocorre. Variantes do curso básico de eventos e erros que possam vir a ocorrer devem ser descritos em cursos alternativos. Normalmente, para cada curso básico de um caso de uso, há diversos cursos alternativos. As técnicas de especificação de processos descritas na Seção 3.7 podem ser aplicadas na descrição de casos de uso. Contudo, há muitas outras maneiras que podem ser usadas para a descrição de casos de uso, ainda que não abordadas neste texto.

3.9 – Modelagem de Estados

Diagramas de Estados são utilizados para descrever o comportamento de uma entidade, com o objetivo de mostrar o comportamento da mesmo ao longo do seu tempo de vida. Diagramas de Estado descrevem todos os possíveis estados pelos quais uma entidade pode passar e as transições dos estados como resultado de eventos (estímulos) que atingem a mesmo. A Figura 3.34 mostra a notação básica para diagramas de estado, segundo a UML.

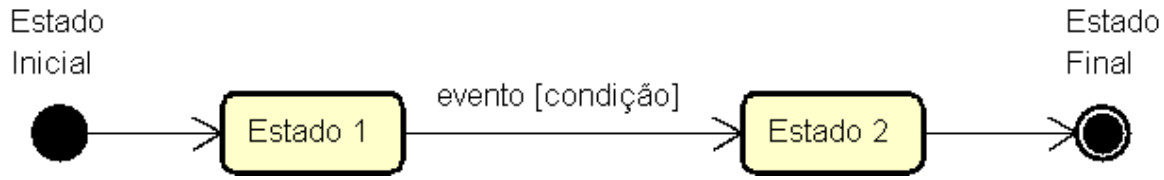


Figura 5.27 - Notação Básica para Diagramas de Estados.

Estados são representados por retângulos com os cantos arredondados e transições por setas, sendo que ambos podem ser rotulados. O rótulo de um estado contém tipicamente o seu nome. Já o rótulo de uma transição tem duas partes principais, ambas opcionais: Evento [condição]. Basicamente a semântica de um diagrama de estados é a seguinte: quando um **evento** ocorre, se a **condição** é verdadeira, a transição ocorre. A entidade passa, então, para um novo estado.

O fato de uma transição não possuir um evento associado, indica que a transição ocorrerá tão logo a condição torne-se verdadeira. Quando uma transição não possuir uma condição associada, então ela ocorrerá sempre que o evento ocorrer.

3.10 – Análise de Requisitos segundo o Paradigma Estruturado

Podemos notar que DFDs e Modelos de Casos de Uso têm alguns aspectos em comum:

- Ambos os modelos mostram as fronteiras do sistema, delimitando o que está fora do sistema (atores nos modelos de casos de uso e entidades externas nos DFDs) e o que é responsabilidade do sistema (casos de uso nos modelos de casos de uso e processos nos DFDs).
- Ambos os modelos mostram quem (atores nos modelos de casos de uso e entidades externas nos DFDs) pode realizar alguma funcionalidade do sistema (casos de uso nos modelos de casos de uso e processos nos DFDs).

Entretanto, há diferenças significativas:

- Nos DFDs, elementos internos ao sistema (depósitos de dados e fluxos de dados) são mostrados. Isso não ocorre nos modelos de casos de uso que buscam modelar as funcionalidades do sistema a partir de uma perspectiva externa.
- As associações entre atores e casos de uso não estabelecem que informação está trafegando, enquanto nos DFDs essa informação é definida pelos fluxos de dados.

Para a maior parte das situações, modelos de casos de uso são suficientes para representar a perspectiva funcional de um sistema. De fato, há muita crítica aos DFDs exatamente por misturarem perspectivas interna (pelo uso de depósitos de dados) e externa, bem como pelo nível de detalhe a que se propõem chegar. Assim, quando for elaborado um modelo de casos de uso, DFDs passam a ser desnecessários. Neste caso, o modelo de análise pode conter apenas:

- **Diagramas de Casos de Uso**
- **Descrições de Atores e Casos de Uso**
- **Diagramas de Entidades e Relacionamentos**
- **Dicionário de Dados**
- **Diagramas de Estados**

Referências do Capítulo

BOOCH, G., RUMBAUGH, J., JACOBSON, I., *UML Guia do Usuário*, 2a edição, Elsevier Editora, 2006.

CHEN, P., *Gerenciando Banco de Dados: A Abordagem Entidade-Relacionamento para Projeto Lógico*. McGraw-Hill, 1990.

DE MARCO, T., *Análise Estruturada e Especificação de Sistemas*. Editora Campus, 1983.

GANE, C. SARSON, T., *Análise Estruturada de Sistemas*. Livros Técnicos e Científicos Editora, 1983

JACOBSON, I., *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Addison-Wesley, 1992.

KOTONYA, G., SOMMERVILLE, I., *Requirements engineering: processes and techniques*. Chichester, England: John Wiley, 1998.

MCMENAMIN, S. M., PALMER, J. F. *Análise Essencial de Sistemas*. McGraw-Hill, 1991.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, 2ª Edição, São Paulo: Prentice Hall, 2004.

POMPILHO, S., *Análise Essencial: Guia Prático de Análise de Sistemas*. IBPI Press, Editora Infobook, Rio de Janeiro, 1995.

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

SETZER, W., *Bancos de Dados*. 2ª Edição, Editora Edgard Blucher, 1987.

SOMMERVILLE, I., *Engenharia de Software*, 9ª Edição. São Paulo: Pearson Prentice Hall, 2011.

YOURDON, E., *Análise Estruturada Moderna*. Editora Campus, 1990.

Capítulo 4 – Projeto de Software

O projeto ou design 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, pelo menos parcialmente, modelados e especificados, correspondendo à primeira dentre as três atividades do domínio da solução computacional – projeto, implementação e testes – requeridas para se construir um sistema de software (PRESSMAN, 2011).

Enquanto a atividade de análise concentra-se no problema a ser resolvido, de forma independente da tecnologia a ser adotada na sua solução, 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 4.1. 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.

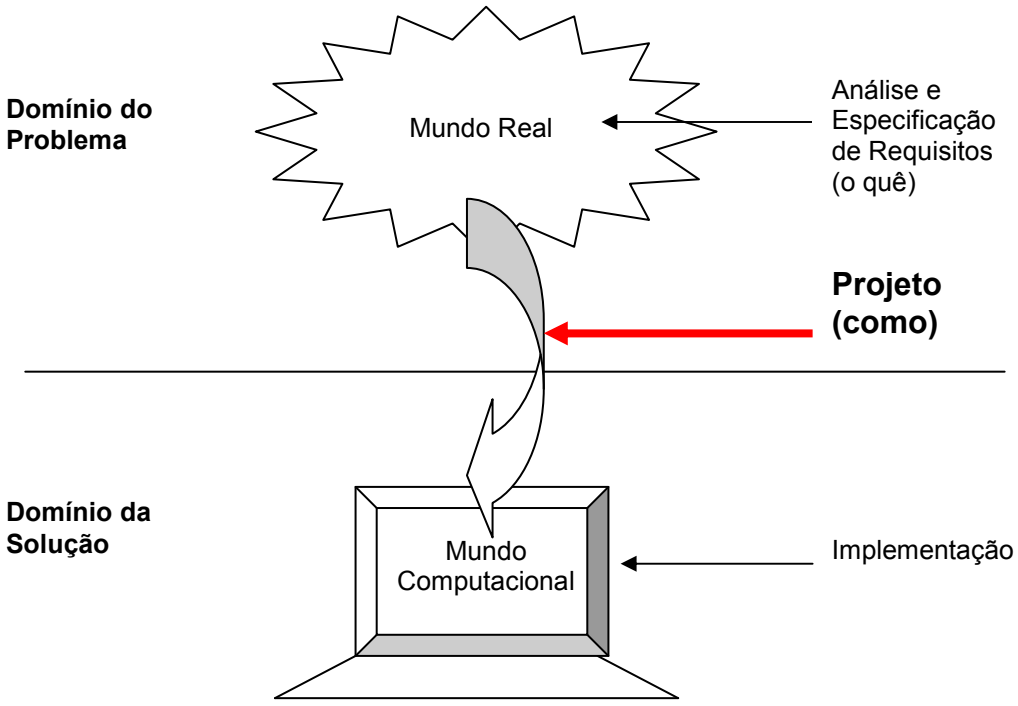


Figura 4.1 – Visão Geral da Atividade de Projeto.

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

Uma especificação de projeto deve:

- Contemplar todos os requisitos especificados nos documentos de requisitos;
- Ser um guia compreensível para aqueles que vão codificar, testar e manter o software.
- Prover um quadro completo do software, tratando aspectos funcionais, comportamentais e de dados, segundo uma perspectiva de implementação.

Na fase de projeto, modelos de projeto são gerados 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 de armazenamento de 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 Detalhado: tem por objetivo refinar e detalhar a descrição dos componentes estruturais da arquitetura do software.

A seguir, cada uma dessas subatividades do projeto de software é discutida à luz do paradigma estruturado.

4.1 - 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 relacional, por exemplo, um modelo relacional deve ser produzido, adequando a modelagem de entidades e relacionamentos a essa plataforma de implementação.

Atualmente, a plataforma mais utilizada para armazenamento de grandes volumes de dados são os Bancos de Dados Relacionais e, portanto, neste texto, discutiremos apenas o projeto lógico de bancos de dados relacionais.

4.1.1 – O Modelo Relacional

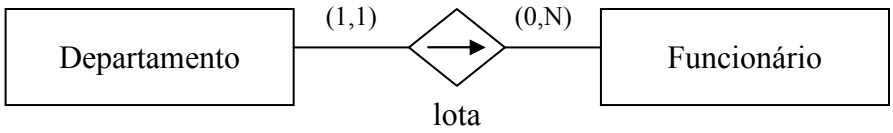
Em um modelo de dados relacional, os conjuntos de dados são representados por tabelas de valores. Cada tabela, denominada relação, é bidimensional, sendo organizada em linhas e colunas. Esse modelo está fortemente baseado na teoria matemática sobre relações, daí o nome relacional. Os principais conceitos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

- **Tabela**: tabela de valores bidimensional organizada em linhas e colunas. A Figura 4.2 mostra um exemplo de uma tabela *Funcionário*, derivada de uma entidade *Funcionário* que tem como atributos matrícula, nome, CPF e data de nascimento.

Matrícula	Nome	CPF	Dt-Nasc
0111	Marcos	17345687691	11/04/66
0208	Rita	56935101129	21/02/64
0789	Mônica	81176628911	01/11/70
1589	Márcia	91125769120	20/10/80

Figura 4.2 – Tabela Funcionário.

- **Linha**: representa uma entidade de um conjunto de entidades. Ex: A funcionária Mônica do conjunto de funcionários.
- **Coluna**: representa um atributo de uma entidade. Ex.: Matrícula, Nome, CPF, Dt-Nasc.
- **Célula**: Item de dado da linha *i*, coluna *j*. Ex.: Rita (linha 2, coluna 2).
- **Chave Primária**: coluna ou combinação de colunas que possui a propriedade de identificar de forma única uma linha da tabela e que é utilizada para estabelecer associações entre entidades via transposição de chave. Ex.: Matrícula, CPF.
- **Chave Estrangeira ou Transposta**: é a forma utilizada para associar linhas de tabelas distintas. A chave primária de uma tabela é transposta como uma coluna na outra tabela, onde é considerada uma chave estrangeira. A Figura 4.3 ilustra um relacionamento 1:N entre as entidades *Departamento* e *Funcionário*, indicando que um departamento pode lotar vários funcionários, enquanto um funcionário tem de estar lotado em um departamento. Essa figura mostra, ainda, o mapeamento para as correspondentes tabelas, indicando a chave transposta.



Departamento

Código	Nome
INF	Informática
LET	Letras
MAT	Matemática

Funcionário

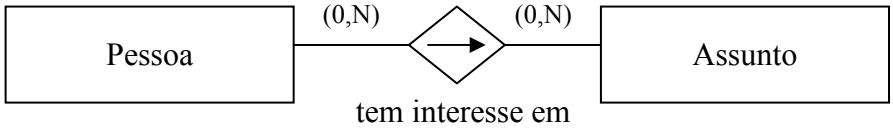
Matrícula	Nome	Cod-Depto
0158	José	MAT
5295	Ricardo	INF
7712	Rosane	INF

Chave Estrangeira

Figura 4.3 – Exemplo de ligação entre tabelas por meio de chave estrangeira.

- **Tabelas Associativas**: usadas para representar relacionamentos muitos para muitos. Seja o exemplo da Figura 4.4, no qual uma pessoa pode ter interesse em vários assuntos, enquanto um assunto pode ser de interesse de várias pessoas. A tabela

Interesse é uma tabela associativa, sendo suas duas colunas chaves transpostas de outras tabelas.



Pessoa

CPF	Nome
96100199	José
83467187	Maria
02765140	Luiza

Interesse

CPF-Pessoa	Código-Assunto
96100199	COMP
96100199	MUS
02765140	ENG

Assunto

Código	Nome
ENG	Engenharia
COMP	Computação
MUS	Música

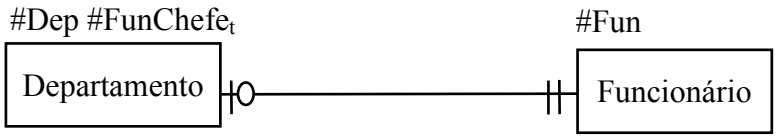
Figura 4.4 – Exemplo de Tabela Associativa.

O modelo relacional tem diversas propriedades que precisam ser respeitadas, a saber:

- Cada tabela possui um nome, o qual deve ser distinto do nome de qualquer outra tabela da base de dados.
- Nenhum campo parte de uma chave primária pode ser nulo.
- Cada célula de uma relação pode ser vazia (exceto os campos de chaves primárias) ou, ao contrário, pode conter no máximo um único valor.
- Não há duas linhas iguais.
- A ordem das linhas é irrelevante.
- Cada coluna tem um nome, o qual deve ser distinto dos demais nomes das colunas de uma mesma tabela.
- Usando-se os nomes para se fazer referência às colunas, a ordem destas torna-se irrelevante.
- Um campo que seja chave estrangeira só pode assumir valor nulo ou um valor para o qual exista um registro na tabela onde ele é chave primária.

Muitas vezes, durante o projeto de bancos de dados relacionais, é útil representar graficamente as tabelas e as ligações entre elas. Para tal, um Diagrama Relacional pode ser desenvolvido, representando as ligações entre tabelas de um modelo relacional. A Figura 4.5 mostra um exemplo de um fragmento de um diagrama relacional e suas tabelas correspondentes.

Diagrama Relacional



Tabelas do Modelo Relacional

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

Funcionários	
Matrícula	Nome
13888	Jorge
00877	Dede
06001	Pedro

Figura 4.5 – Exemplo de Diagrama Relacional e as respectivas tabelas

Nesse exemplo a coluna *Matrícula* foi considerada a chave primária da tabela *Funcionário* e foi transposta para a tabela *Departamento*. O contrário também poderia ser feito, isto é, transpor a chave primária de *Departamento* para *Funcionário*. A primeira opção é mais indicada, porque há poucos funcionários que são chefes, enquanto todos os departamentos têm chefes. Assim, a coluna *Matrícula-Chefe* não terá valores vazios e, portanto, ela é mais densa do que seria a coluna resultante da transposição da chave primária de *Departamento* para a tabela *Funcionário*.

Em um Diagrama Relacional são representados os seguintes elementos:

- Tabelas: são representadas por retângulos, com uma referência à chave primária em cima da tabela.

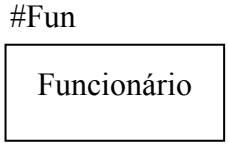


Figura 4.6 – Notação para Tabelas em um Diagrama Relacional

- Relacionamentos: são representadas por linhas contínuas, associadas aos símbolos abaixo:

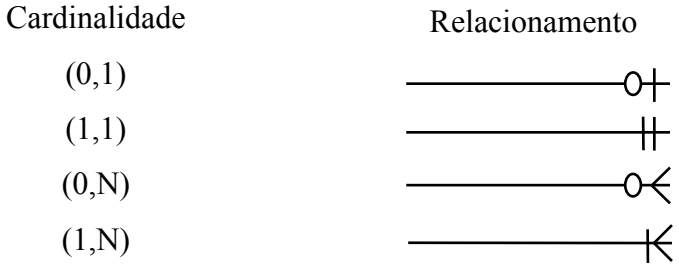


Figura 4.7 – Notação para Relacionamentos em um Diagrama Relacional

- Chaves estrangeiras: quando uma chave transposta não fizer parte da chave primária da relação destino, a mesma é representada em cima do retângulo da relação destino com um subscrito "t", como ilustra a Figura 4.5.

Colunas que não são chaves primárias ou estrangeiras não são representadas nos diagramas, mas sim em um dicionário de dados do modelo relacional.

4.1.2 – Construindo um Modelo Relacional a partir de um Modelo ER

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 entidades associativas devem dar origem a tabelas;
- Uma instância de uma entidade deve ser representada como uma linha da tabela correspondente;
- Um atributo de uma entidade 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 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 caso de relacionamentos um para um (1:1), para decidir qual chave transpor, deve-se considerar alguns aspectos. Seja um relacionamento um para um **R** entre dois conjuntos de entidades **A** e **B**:

- 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 4.8.
- Se **B** for total em **R** (todo **B** está associado a um **A**), é melhor colocar a chave de **A** (#A) em **B**.
- Nos demais casos, é melhor transpor a chave que dará origem a uma coluna mais densa, isto é, que terá menos valores nulos.

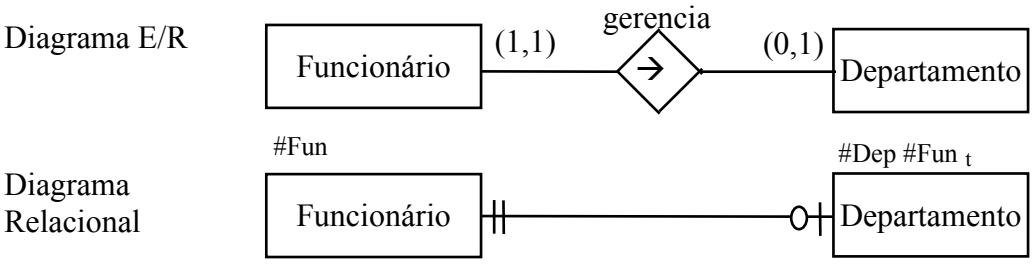


Figura 4.8 – Exemplo de Relacionamento 1:1.

Relacionamentos 1 : N

Neste caso, deve-se transpor a chave da tabela correspondente à entidade de cardinalidade máxima N para a tabela que representa a entidade cuja cardinalidade máxima é 1, como mostra a Figura 4.9.

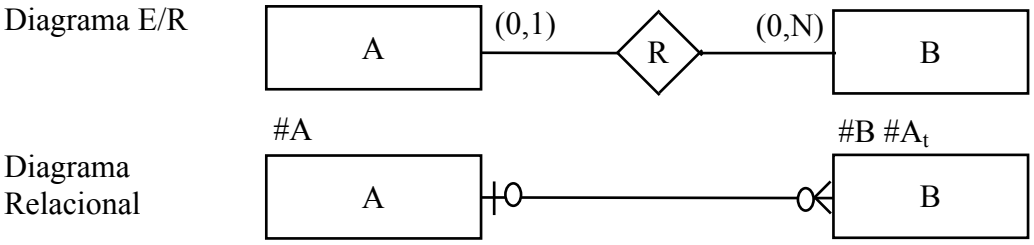


Figura 4.9 - Relacionamentos 1:N no Diagrama Relacional.

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 4.10 mostra um exemplo desta situação.

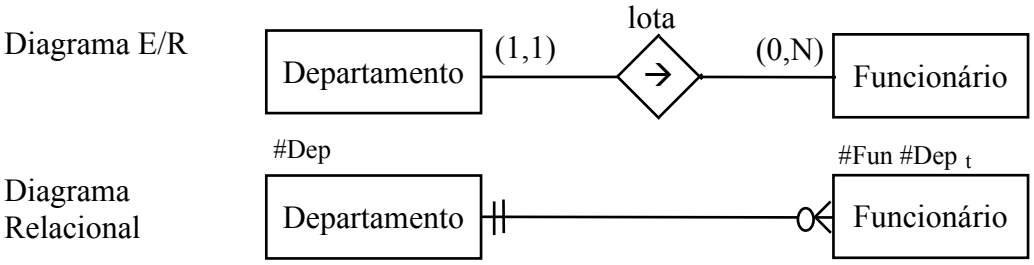


Figura 4.10 – Exemplo de Relacionamento 1:N.

Relacionamentos N : N

No caso de relacionamentos N:N, deve-se criar uma terceira tabela, transpondo as chaves primárias das duas tabelas que participam do relacionamento N:N, como mostra a Figura 4.11. 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 tabela associativa, como ilustra o exemplo da Figura 4.12.

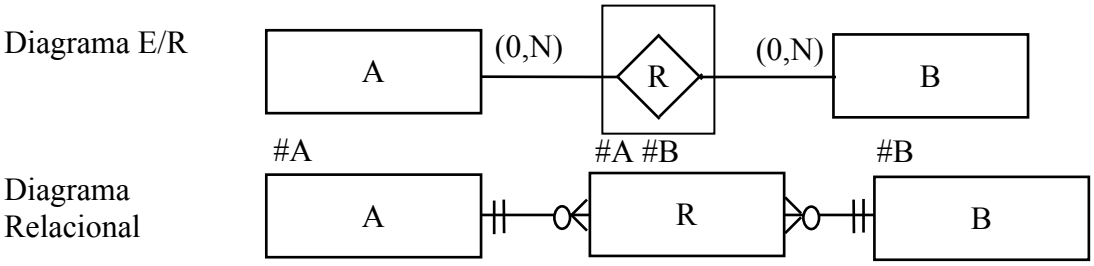


Figura 4.11 - Relacionamentos N:N no Diagrama Relacional.

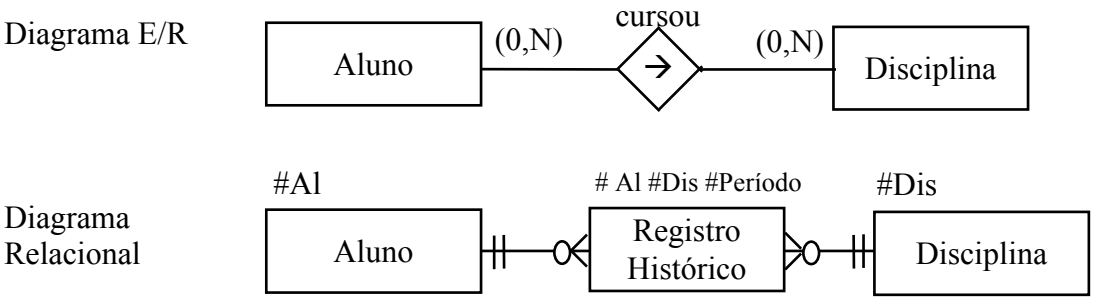


Figura 4.12 – Exemplo de relacionamento N:N.

Auto-Relacionamentos

Os auto-relacionamentos devem seguir as mesmas regras de tradução de relacionamentos, como ilustram os exemplos das figuras 4.13 e 4.14.

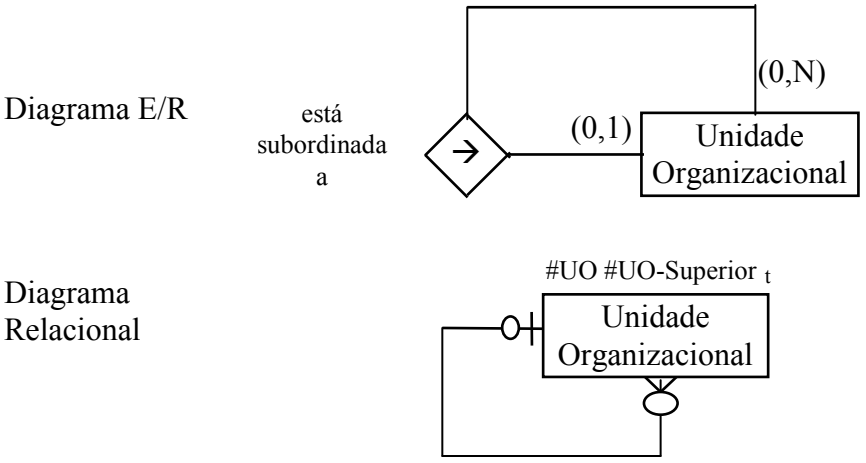


Figura 4.13 – Exemplo de Auto-relacionamento 1:N.

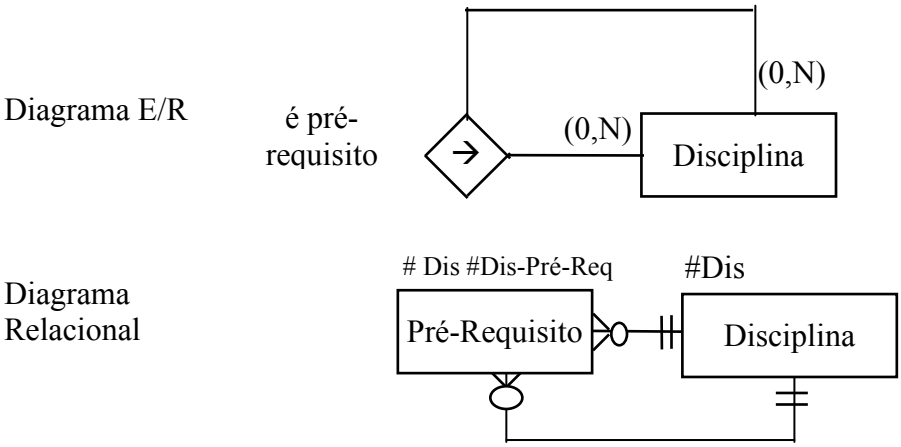


Figura 4.14 – Exemplo de um Auto-relacionamento N:N.

Relacionamento Ternário

No caso de relacionamentos ternários, deve-se criar uma nova tabela contendo as chaves das três entidades envolvidas, como mostra a Figura 4.15. Assim como no caso dos relacionamentos binários N:N, 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.

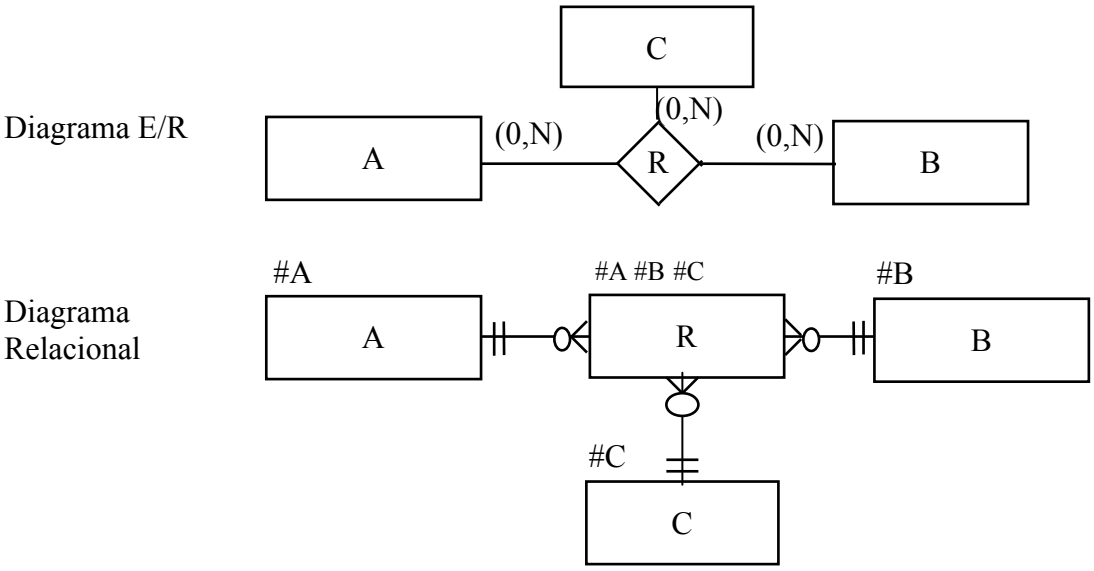


Figura 4.15 - Relacionamentos Ternários no Diagrama Relacional.

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 4.16. 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.

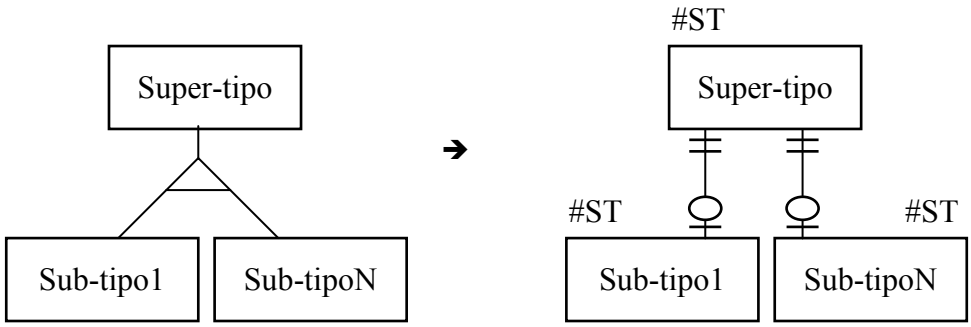


Figura 4.16 – Tradução de Particionamento.

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 4.17.

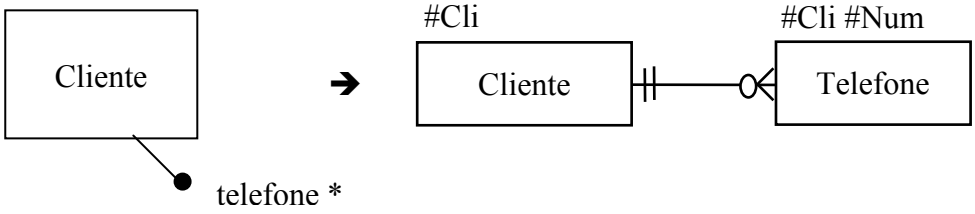


Figura 4.17 – Mapeamento Geral de Atributos Multivalorados.

4.2 - 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 é responsável por definir 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 (PRESSMAN, 2011). De maneira geral, o projeto de interfaces com o usuário segue o seguinte processo global, como mostra a Figura 4.18:

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.
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. 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).

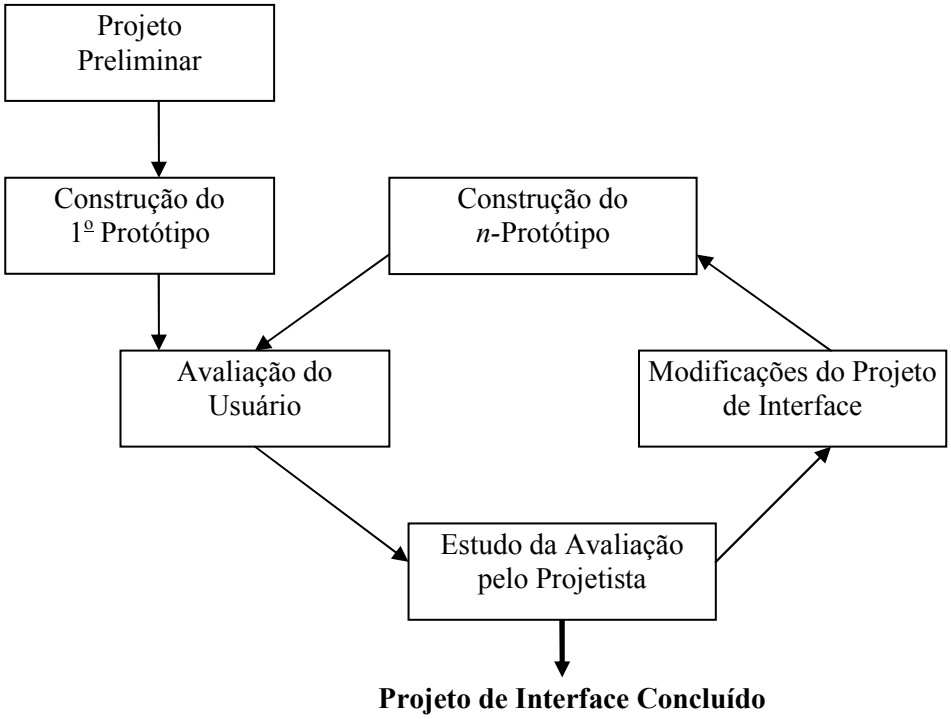


Figura 4.18 - Abordagem Iterativa para o Projeto de Interface com o Usuário.

4.3 - Projeto Modular de Programas

A tarefa de construção de sistemas computadorizados requer uma organização das ideias, 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 que visam a conduzir a bons programas a serem desenvolvidos segundo o paradigma estruturado. O objetivo é desenvolver programas com menor complexidade, usando o princípio "dividir para conquistar". Como resultado de um bom projeto de programas, espera-se obter:

- 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 de programas é um processo iterativo e de refinamento que pode ser descrito em duas etapas principais: 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 ideia 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.
- 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 uma determinada 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 os organizando 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.

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 (MARTIN; MCCLURE, 1991). Neste texto, serão explorados dois deles: o Diagrama Hierárquico de Funções (DHF), usado principalmente para o projeto arquitetural, e o Diagrama de Estrutura Modular (DEM), usado para o projeto detalhado de módulos. A diferença básica entre eles é que o DHF não representa o fluxo de dados e controles entre módulos, nem aspectos relacionados com detalhes lógicos de um módulo, tais como estruturas de repetição (laços) e condições. 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.

4.3.1 - 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 (MARTIN; MCCLURE, 1991). Cada módulo pode representar um subsistema, programa ou módulo de programa. Sua finalidade é mostrar os componentes funcionais gerais (arquitetura do sistema) e fazer referência a diagramas detalhados (tipicamente Diagramas de Estrutura Modular). Um DHF não mostra o fluxo de dados entre componentes funcionais ou qualquer informação de estruturas de controle, tais como laços (*loops*) ou condições.

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

A construção de um DHF deve procurar espelhar a estrutura do negócio que o sistema está tratando. A descrição do escopo, com sua subdivisão em subsistemas, e os casos de uso 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 lanches, cujo escopo é o seguinte:

- Subsistema Controle de Cardápio, envolvendo macro-funções para: Cadastrar Lanches 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 lanches 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 4.19. 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 4.19, 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. Assim, o projeto de IU deve ocorrer em paralelo com o projeto modular de programas.

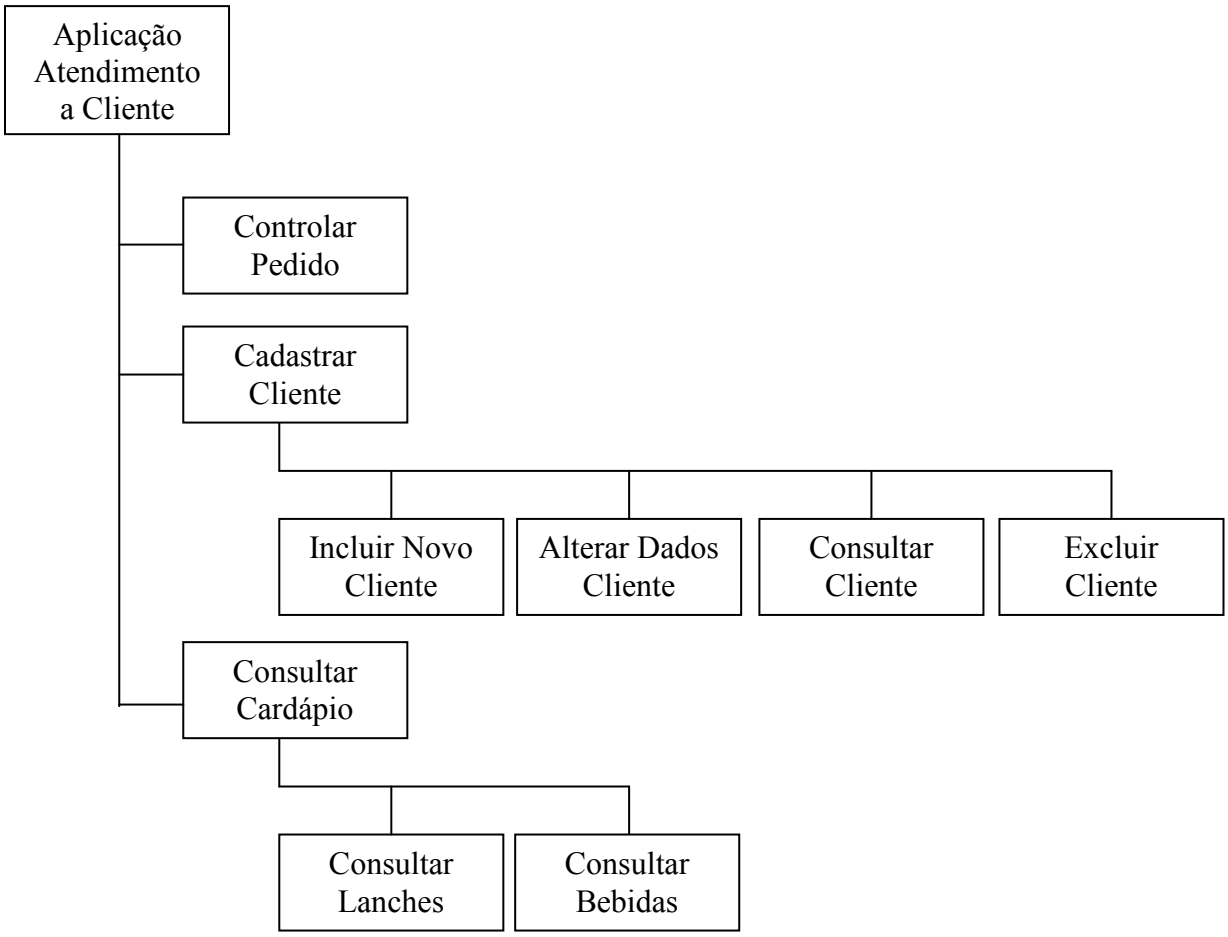


Figura 4.19 – Exemplo de DHF.

4.3.2 - 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 (MARTIN; MCCLURE, 1991).

Um módulo é definido como uma coleção de instruções de programa com quatro atributos básicos: entradas e saídas, função, lógica e dados internos. Entradas e saídas são, respectivamente, as informações que um módulo necessita e fornece. A função de um módulo é o que ele faz para produzir, a partir da informação de entrada, os resultados da saída. Entradas, saídas e função fornecem a visão externa do módulo e, portanto, apenas esses aspectos são representados em um 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 3.

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 (XAVIER; PORTILHO, 1995):

- **Módulo**: Em um DEM, um módulo é representado por um retângulo, dentro do qual está contido seu nome, como mostra a Figura 4.20. Um módulo pré-definido é aquele que já existe em uma biblioteca de módulos e, portanto, não precisa ser descrito ou detalhado.

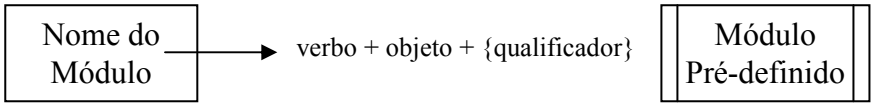


Figura 4.20 – Simbologia para Módulos em um DEM.

- **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 4.21, 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 a chamada de *B*, passando como resultado o dado *Z*. A ordem de chamada é sempre de cima para baixo, da esquerda para a direita.

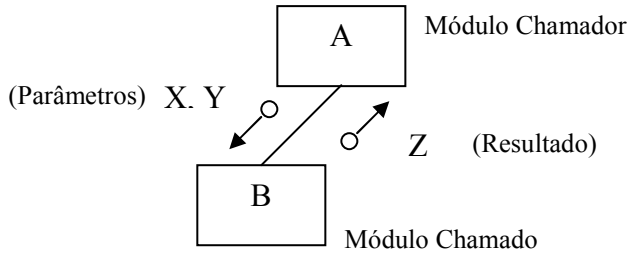


Figura 4.21 – Conexão entre módulos.

- **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 4.22 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.

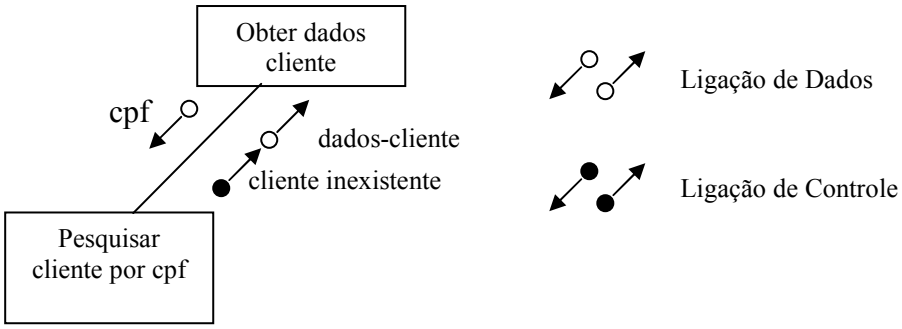


Figura 4.22 – Comunicação entre módulos.

- **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 4.23. 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*.

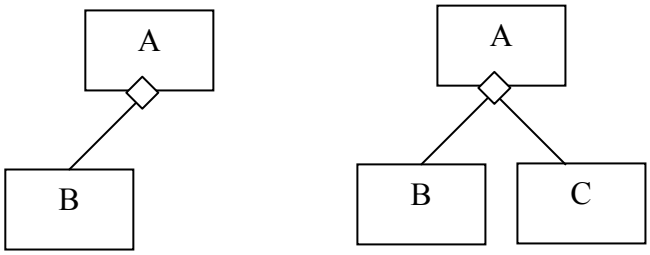


Figura 4.23 – Chamada Condicional.

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

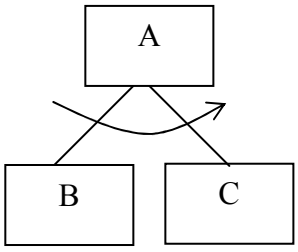


Figura 4.24 – Chamada Iterativa.

- **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 4.25.

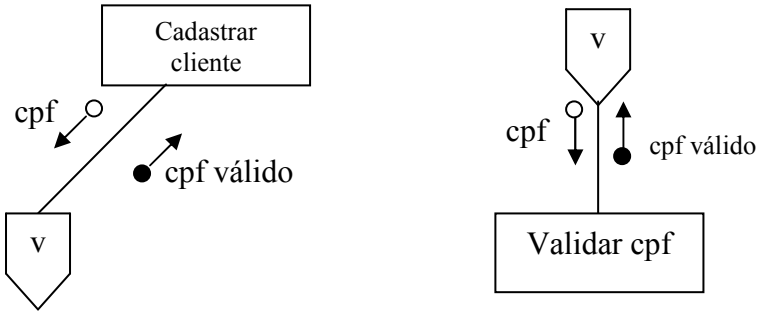


Figura 4.25 – Conectores.

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.
- Idealmente, cada módulo só deveria aparecer uma única vez no diagrama. Para se evitar cruzamento de linhas, podem-se usar conectores.
- Não segmentar demais.

Critérios de Qualidade de Diagramas de Estrutura Modular

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. Podemos citar como razões para 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.

As seguintes orientações podem ajudar a minimizar o acoplamento entre módulos:

- 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 inadmissível que um módulo se refira a uma parte interna de outro.
- Ter pontos únicos de entrada e saída em um módulo.

Coesão define como as atividades de um módulo estão relacionadas umas com as outras. 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.

Vale a pena ressaltar que coesão e acoplamento são interdependentes e, portanto, uma boa coesão deve nos levar a um pequeno acoplamento. A Figura 4.26 procura ilustrar este fato.

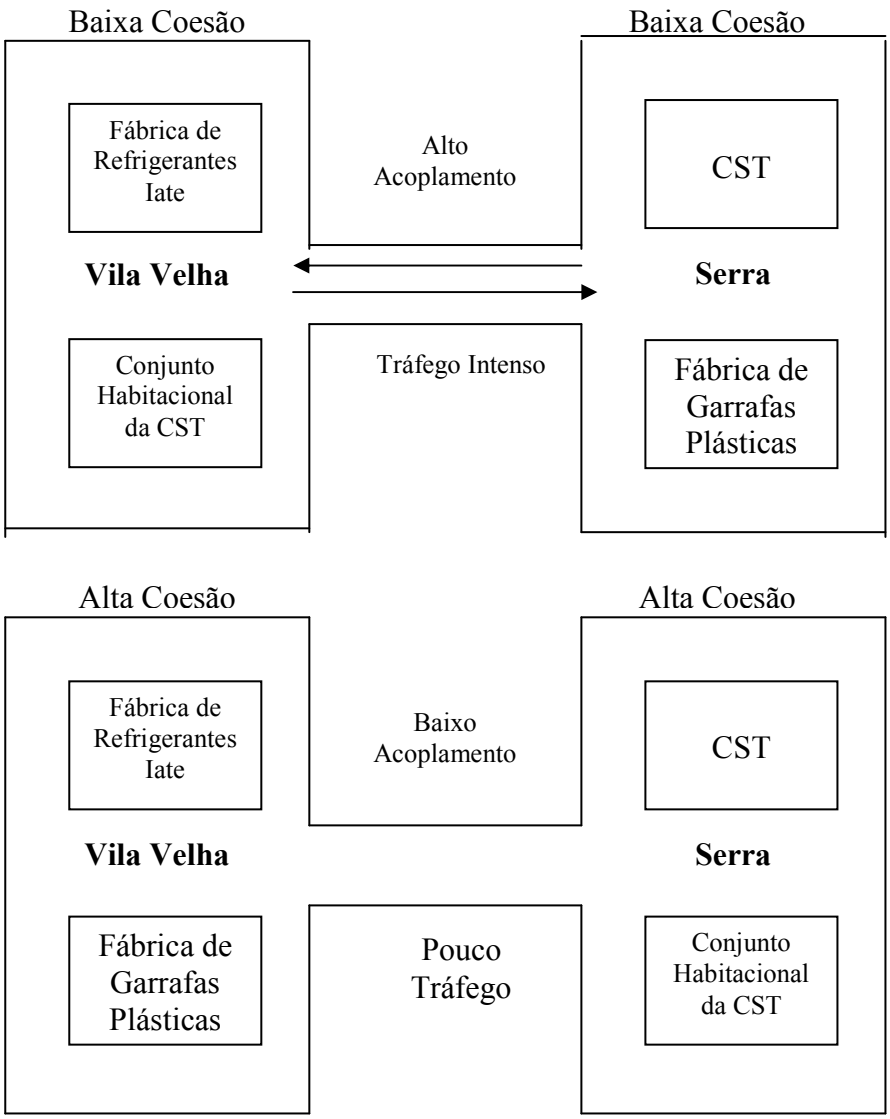


Figura 4.26 – Coesão e Acomplamento.

Referências do Capítulo

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

MARTIN, J., MCCLURE, C., *Técnicas Estruturadas e CASE*. Makron Books, São Paulo, 1991.

XAVIER, C.M.S. PORTILHO, C., *Projetando com Qualidade a Tecnologia de Sistemas de Informação*. Livros Técnicos e Científicos Editora, 1995.

Capítulo 5 – Implementação e Teste de Software

Uma vez projetado o sistema, é necessário escrever os programas que implementem esse projeto e testá-los. Os testes devem ser feitos em diversos níveis, começando pelos módulos isolados (teste de unidade), passando a integrá-los (teste de integração), até atingir os testes do sistema como um todo (teste de sistema). Diversas técnicas podem ser empregadas para este fim.

De fato, dada sua importância, testes não devem ser tratados apenas como uma atividade no ciclo de vida de software, mas sim como um processo. O processo de teste deve ocorrer em paralelo com outras atividades do processo de desenvolvimento de software (análise de requisitos, projeto de software e implementação) e envolve também atividades de planejamento.

Este capítulo aborda brevemente, na Seção 5.1, a atividade de implementação. A Seção 5.2 discute princípios gerais de teste de software. A Seção 5.3 trata de níveis de teste, enquanto a Seção 5.4 trata de técnicas de teste. Por fim, a Seção 5.5 aborda o processo de teste.

5.1 - 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 (PFLEEGER, 2004). 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 (PFLEEGER, 2004).

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, grupos de componentes e da inter-relação entre eles (PFLEEGER, 2004).

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 tenha certa flexibilidade para implementá-lo como código (PFLEEGER, 2004).

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

5.2 – Princípios Gerais de Teste de Software

O desenvolvimento de software está sujeito a diversos tipos de problemas, os quais acabam resultando na obtenção de um produto diferente daquele que se esperava. Muitos fatores podem ser identificados como causas de tais problemas, mas a maioria deles tem como origem o erro humano (DELAMARO et al., 2007). Para avaliar a qualidade de um produto de software, há dois tipos principais de atividades:

- Verificação: visa assegurar que o software, ou determinada função do mesmo, está sendo desenvolvido corretamente, o que inclui verificar se os métodos e processos estão sendo aplicados adequadamente;
- Validação: visa garantir que o software que está sendo desenvolvido é o software correto.

As atividades de Verificação e Validação (V&V) podem ser divididas em estáticas e dinâmicas. A análise estática não envolve a execução do produto e, portanto, ela pode ser aplicada em qualquer artefato intermediário. Já a análise dinâmica envolve a execução do produto. Revisões técnicas e inspeção de código são exemplos de técnicas que podem ser aplicadas para analisar estaticamente um produto de software (ou partes dele). Técnicas para revisão de software são abordadas no Capítulo 7 destas notas de aula. Neste capítulo, são abordadas apenas atividades de V&V dinâmicas: os testes de software.

Teste de software é o processo de executar um programa com o objetivo de encontrar defeitos (MYERS, 2004). 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 (MALDONADO; FABBRI, 2001). Entretanto, vale ressaltar que, mesmo se um teste não detectar defeitos, isso não quer dizer necessariamente que o produto é um produto de boa qualidade. Teste só é capaz de apontar a existência de defeitos e não a ausência deles. 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 (MALDONADO; FABBRI, 2001).

Assim, o objetivo é projetar casos de teste que potencialmente descubram diferentes classes de erros e fazê-lo com uma quantidade mínima de esforço (PRESSMAN, 2011). 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 ideia 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 (PFLEEGER, 2004).

Do ponto de vista psicológico, o teste de software é uma atividade com certo viés destrutivo, ao contrário das outras atividades do processo de desenvolvimento de software. A perspectiva de teste requer um modo de olhar um produto e questionar a sua validade. Um testador deve abordar um software com a atitude de questionar tudo sobre ele. A perspectiva de teste requer que um fragmento de software demonstre não apenas que ele executa de acordo com o especificado, mas que executa apenas o especificado. Assim, bons testadores necessitam de um conjunto especial de habilidades, dentre elas: querer prova de qualidade, não fazer suposições, não deixar passar áreas importantes e procurar ser reproduzível (MCGREGOR; SYKES, 2001).

Em um teste de software, executa-se um programa utilizando algumas entradas em particular e verificar-se se seu comportamento está de acordo com o esperado. Caso a execução apresente algum resultado não especificado, um defeito foi identificado. Os dados da execução podem servir como fonte para a localização e correção de defeitos, mas teste não é depuração (DELAMARO et al., 2007).

Seja *P* um programa a ser testado. O domínio de entrada de *P* (denominado *D(P)*) é o conjunto de todos os valores possíveis que podem ser utilizados para executar *P*. Um dado de teste para *P* é um elemento de *D(P)*. O domínio de saída de *P* é o conjunto de todos os possíveis resultados produzidos por *P*. Um caso de teste de *P* é um par formado por um dado de teste mais o resultado esperado para a execução de *P* com aquele dado de teste. Ao conjunto de todos os casos de teste usados durante uma determinada atividade de teste dá-se o nome de conjunto de casos de teste ou conjunto de teste (DELAMARO et al., 2007).

Definido um conjunto de casos de teste *T*, executa-se *P* com *T* e verificam-se os resultados obtidos. Se os resultados obtidos coincidem com os resultados esperados, então nenhum defeito foi identificado e diz-se que o software passou no teste. Se, para algum caso de teste, o resultado obtido difere do esperado, então um defeito foi detectado e o software não passou no teste. De maneira geral, fica por conta do testador, baseado na especificação do programa, decidir sobre a correção da execução (DELAMARO et al., 2007). A Figura 5.1 ilustra um cenário típico da atividade de teste.

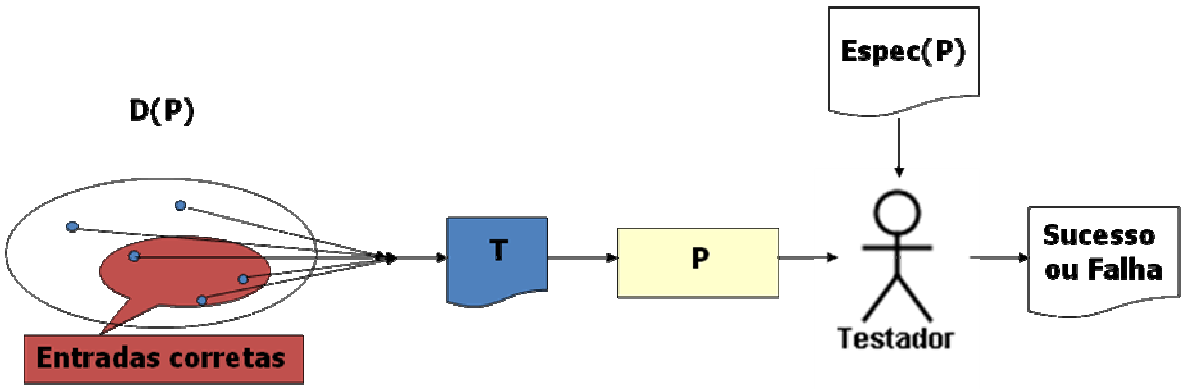


Figura 5.1 – Atividade de Teste.

Para se poder garantir que um programa *P* não contém defeitos, *P* deveria ser executado com todos os elementos de *D(P)*. Seja um programa `exp` com a seguinte especificação: `exp(int x, int y)=` x^y . *D(*`exp`*)* corresponde a todos os pares de números inteiros (x,y) passíveis de representação. A cardinalidade (#) de *D(*`exp`*)* = $2^n * 2^n$, onde n = número de bits usado para representar um inteiro. Em uma arquitetura de 32 bits, #(D(exp)) = 2^{64}. Se cada teste puder ser executado em 1 milissegundo, seriam necessários aproximadamente 5,85 milhões de séculos para executar todos os testes. Logo, em geral, teste exaustivo não é viável e, por conseguinte, testes podem mostrar apenas a presença de defeitos, mas não a ausência deles. Para se testar um programa *P*, devem ser selecionados alguns pontos específicos de *D(P)* para executar (DELAMARO et al., 2007).

Um aspecto crucial para o sucesso na atividade de teste é a escolha correta dos casos de teste. Um teste bem-sucedido identifica defeitos que ainda não foram detectados. Um bom caso de teste é aquele que tem alta probabilidade de encontrar um defeito ainda não descoberto. A escolha de casos de teste passa pela identificação de subdomínios de teste. Um subdomínio de teste é um subconjunto de *D(P)* que contém dados de teste semelhantes, ou seja, que se comportam do mesmo modo; por isso, basta executar *P* com apenas um deles. Fazendo-se isso com todos os subdomínios de *D(P)*, consegue-se um conjunto de teste T bastante reduzido em relação a *D(P)*, mas que, de certa maneira, representa cada um de seus elementos (DELAMARO et al., 2007).

Existem duas maneiras de se selecionar elementos de cada um dos subdomínios de teste (DELAMARO et al., 2007):

- Teste Aleatório: um grande número de casos de teste é selecionado aleatoriamente, de modo que, probabilisticamente, se tenha uma boa chance de ter todos os subdomínios representados em *T*.
- Teste de Subdomínios ou Teste de Partição: procura-se estabelecer quais são os subdomínios a serem utilizados e, então, selecionam-se os casos de teste em cada subdomínio.

A identificação dos subdomínios é feita com base em critérios de teste. Dependendo dos critérios estabelecidos, são obtidos subdomínios diferentes (DELAMARO et al., 2007). Diferentes técnicas de teste utilizam diferentes critérios e, por conseguinte, levam a partições diferentes.

Em essência, são importantes princípios de testes a serem observados (PRESSMAN, 2011; PFLEEGER, 2004):

- 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, pelo menos parcialmente, 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.

5.3 – Níveis de Teste

Conforme apontado anteriormente, o teste de software envolve vários estágios ou níveis. Basicamente, há três grandes fases de teste (MALDONADO; FABBRI, 2001; DELAMARO et al., 2007):

- **Teste de Unidade**: tem por objetivo testar a menor unidade do projeto, procurando identificar erros de lógica e de implementação em cada módulo separadamente. No paradigma estruturado, procedimentos e funções são exemplos de unidades.
- **Teste de Integração**: visa a descobrir erros associados às interfaces dos módulos quando esses são integrados para formar a estrutura do produto de software.
- **Teste de Sistema**: tem por objetivo identificar erros de funções (requisitos funcionais) e outras características (requisitos não funcional) que não estejam de acordo com as especificações.

Tomando por base essas fases, o processo de teste pode ser estruturado de modo que, em cada fase, diferentes tipos de erros e aspectos do software sejam considerados (MALDONADO; FABBRI, 2001). Tipicamente, os primeiros testes focalizam componentes individuais. Os testes de unidade podem ser realizados à medida que ocorre a implementação das unidades e podem ser realizados pelos próprios desenvolvedores (DELAMARO et al., 2007). 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 (PRESSMAN, 2011; PFLEEGER, 2004).

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 (MALDONADO; FABBRI, 2001).

Pode não ser prático e viável testar todas as unidades individualmente para depois integrá-las, dada a necessidade de uma grande quantidade de *drivers* e *stubs* a ser construída. É possível adotar estratégias de teste de integração que diminuam a quantidade de *drivers* e *stubs* necessários. 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 (PFLEEGER, 2004). Neste caso, apenas *drivers* são necessários. Seja o exemplo da Figura 5.1. 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.

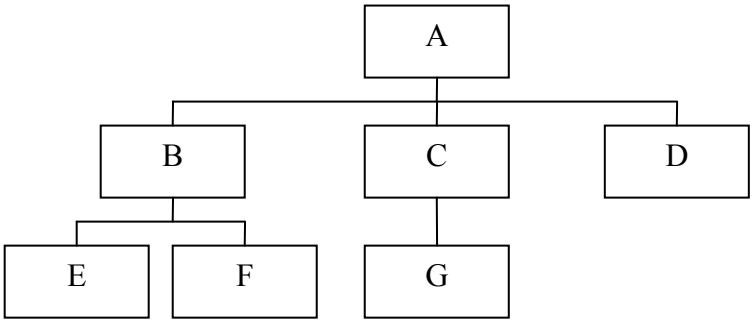


Figura 5.1 – Exemplo de uma hierarquia de módulos.

- **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, os módulos chamados pelo módulo testado são combinados e testados como uma grande unidade. Essa abordagem é repetida até que todos os módulos tenham sido incorporados (PFLEEGER, 2004). Neste caso, apenas *stubs* são necessários. Tomando o exemplo da Figura 5.1, o teste iniciaria pelo módulo A e três *stubs* (para B, C e D) seriam necessários. Na sequência 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 (PFLEEGER, 2004), 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 depois integrar todos de uma vez (teste *big-bang*). Neste caso,

tanto *drivers* quanto *stubs* têm de ser construídos para cada módulo, o que leva a muito mais codificação e problemas em potencial (PFLEEGER, 2004).

Outras abordagens de integração fazem uso dos modelos construídos na fase de análise para testar a integração dos módulos. Na estratégia de integração baseada em casos de uso, a integração dos módulos é testada no contexto da realização de um caso de uso. Uma vez testados casos de uso individualmente, outras estratégias podem ser usadas para testar vários casos de uso integrados. A estratégia de integração baseada no ciclo de vida do domínio, por exemplo, consiste em realizar os processos de negócio (através dos correspondentes casos de uso) como eles tipicamente acontecem. Na estratégia de integração baseada em máquina de estados, vários casos de uso são testados para exercitar estados e transições em um diagrama de estados.

Uma vez integrados todos os módulos do sistema, parte-se para os testes de sistema. Os testes de sistema incluem diversos tipos de teste, realizados, geralmente, na seguinte ordem (PFLEEGER, 2004):

- Teste funcional de sistema: verifica se o sistema integrado realiza as funções especificadas nos requisitos;
- Teste não funcional de sistema: verifica se o sistema integrado atende os requisitos não funcionais do sistema (eficiência, segurança, confiabilidade etc);
- Teste de aceitação: os testes funcionais e não funcionais citados anteriormente são realizados por testadores (ou eventualmente por desenvolvedores, ainda que desaconselhável) tomando por base a especificação do sistema e, portanto, são testes de verificação. Entretanto, é necessário que o sistema seja testado também pelos clientes e usuários (testes de validação). No teste de aceitação, clientes e usuários 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 o cliente queria. 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.

Além dos testes de unidade, integração e sistema, outros tipos de teste são realizados ao longo do ciclo de vida de um sistema. Os testes de regressão, por exemplo, são realizados por ocasião da ocorrência de mudanças. A cada novo módulo adicionado ou a cada alteração, o software se modifica. Essas modificações podem introduzir defeitos, inclusive em funções que antes funcionavam corretamente. Assim, é necessário verificar se as alterações efetuadas estão corretas, reexecutando algum subconjunto dos testes para garantir que as modificações não estão propagando efeitos colaterais indesejados (PRESSMAN, 2011).

5.4 – Técnicas de Teste

Diversas técnicas de teste têm sido propostas visando apoiar o projeto de casos de teste. Essas técnicas podem ser classificadas, segundo os critérios utilizados para estabelecer os objetivos de teste, em testes funcionais, estruturais, baseados em modelos, baseados em defeitos, dentre outros. Neste texto, são discutidas apenas algumas técnicas funcionais e estruturais. Para maiores detalhes, vide (DELAMARO et al., 2007).

5.4.1 – Testes Funcionais

Os testes funcionais, ou de 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. Os testes são conduzidos na interface do software e o programa ou sistema é considerado uma caixa-preta. Para testar um módulo, programa ou sistema, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com a correspondente especificação. Detalhes de implementação não são levados em conta. Deste modo, o software é avaliado segundo o ponto de vista do usuário (DELAMARO et al., 2007).

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 (PRESSMAN, 2011).

As técnicas de teste funcional estabelecem critérios para particionar o domínio de entrada em subdomínios, a partir dos quais serão definidos os casos de teste (DELAMARO et al., 2007). Cada técnica estabelece um critério de particionamento. Dentre as diversas técnicas de teste caixa-preta, podem ser citadas (DELAMARO et al., 2007; PRESSMAN, 2011): particionamento de equivalência, análise de valor limite e teste funcional sistemático.

Todos os critérios das técnicas funcionais baseiam-se apenas na especificação do produto testado e, portanto, o teste funcional depende fortemente da qualidade da especificação sendo testada. Além disso, podem ser aplicados em todas as fases de teste e são independentes de paradigma, pois não levam em consideração a implementação.

O critério de **particionamento de equivalência** divide o domínio de entrada de um módulo em classes de equivalência, a partir das quais casos de teste são derivados. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos das condições de entrada. A meta é minimizar o número de casos de teste, ficando apenas com um caso de teste para cada classe de equivalência, uma vez que, em princípio, todos os elementos de uma mesma classe devem se comportar de maneira equivalente.

Quando o critério de particionamento de equivalência é aplicado, inicialmente deve-se repartir o domínio de entrada de um módulo ou programa em classes ou partições de equivalência. Caso as classes de equivalência se sobreponham ou os elementos de uma mesma classe não se comportem da mesma maneira, elas devem ser revistas, a fim de torná-las distintas (DELAMARO et al., 2007).

Uma vez identificadas as classes de equivalência, devem-se definir os casos de teste, escolhendo um elemento de cada classe. Qualquer elemento da classe pode ser considerado um representante desta e, portanto, basta ter definir um caso de teste por classe de equivalência (DELAMARO et al., 2007).

Seja o caso de uma função para calcular o imposto de renda devido de um valor informado, cuja interface é a seguinte: `calcularIR(valor: float): float`. A especificação desta função é dada pela Tabela 5.1, levando em conta, ainda, as seguintes considerações:

- Entrada válida: Valor monetário, maior ou igual a zero.
- Saídas possíveis (incluindo erros decorrentes de entradas inválidas):
  - Valor monetário referente ao imposto de renda devido.
  - Mensagem informando que o valor informado não é um valor monetário.
  - Mensagem de erro informando que o valor informado é menor do que zero.

Tabela 5.1 – Tabela de Imposto de Renda (IR)

Base de Cálculo (R$)	Alíquota (%)	Parcela a Deduzir do IR (R$)
Até 1.710,78	-	-
De 1.710,79 até 2.563,91	7,5	128,31
De 2.563,92 até 3.418,59	15	320,60
De 3.418,60 até 4.271,59	22,5	577,00
Acima de 4.271,59	27,5	790,58

Tomando por base a especificação acima apresentada, a Tabela 5.2 apresenta o conjunto de classes de equivalência correspondentes e a Tabela 5.3 mostra casos de teste representativos de cada uma dessas classes.

Tabela 5.2 – Classes de Equivalência

Entrada	Classes de Equivalência	Tipo de Classe de Equivalência
valor	valor é um valor monetário entre 0,00 e 1.710,78	Válida
	valor é um valor monetário entre 1.710,79 e 2.563,91	Válida
	valor é um valor monetário entre 2.563,92 e 3.418,59	Válida
	valor é um valor monetário entre 3.418,59 e 4.271,59	Válida
	valor é um valor monetário maior do que 4.271,59	Válida
	valor não é um valor monetário	Inválida
	valor é um valor monetário menor do que 0	Inválida

Tabela 5.3 – Casos de Teste

Entrada	Saída Esperada
1.320,50	0,00
2.000,00	21,69
3.410,00	190,90
3.912,55	303,32
8.475,29	1.540,12
-2.000,00	Erro: valor deve ser maior ou igual a 0.
Axt89	Erro: Entre com um valor monetário.

O critério de particionamento de equivalência possibilita uma boa redução do tamanho do domínio de entrada, sendo especialmente adequado para aplicações em que as variáveis de entrada podem ser facilmente identificadas e assumem valores específicos. Contudo, não é tão facilmente aplicável mesmo quando o domínio de entrada é simples, se o processamento do módulo for complexo. Nestes casos, a especificação pode sugerir que um grupo de dados seja processado de forma idêntica, mas isso pode não ocorrer na prática (DELAMARO et al., 2007).

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, o critério de **análise de valor limite** tem por premissa que casos de teste que exploram condições limites têm maior probabilidade de encontrar defeitos. Tais condições estão exatamente sobre ou imediatamente acima ou abaixo dos limitantes das classes de equivalência. Assim, este critério é usado em conjunto com o particionamento de equivalência e leva à seleção de casos de teste que exercitem valores limítrofes (DELAMARO et al., 2007). As recomendações do critério de análise de valor limite são as seguintes:

- Se a condição de entrada especificar um intervalo de valores de entrada, então se devem definir casos de teste para os limites desse intervalo e para valores imediatamente subsequentes, acima e abaixo, que explorem as classes vizinhas.
- Se a condição de entrada especificar uma quantidade de valores de entrada (n), então se devem definir casos de teste com nenhum valor de entrada, somente um valor de entrada, com a quantidade máxima de valores de entrada (n) e com a quantidade máxima de valores + 1 ($n + 1$).
- Aplicar as recomendações acima para condições de saída, quando couber.
- Se a entrada ou a saída for um conjunto ordenado, deve ser dada atenção especial aos primeiro e último elementos desse conjunto.

Seja o exemplo da função para calcular o imposto de renda. Neste caso, a entrada especifica 5 intervalos de valor e, portanto, aplicando-se a primeira das recomendações acima listadas, são necessários mais 15 casos de teste, 3 para cada limite de valor, como mostra a Tabela 5.4.

Tabela 5.4 – Casos de Teste Adicionais – Análise de Valor Limite

Entrada	Saída Esperada
-0,01	Erro: valor deve ser maior ou igual a 0.
0,00	0,00
0,01	0,00
1.710,97	0,00
1.710,98	0,00
1.710,99	0,00
2.563,90	63,98
2.563,91	63,98
2.563,92	63,98
3.418,58	192,18
3.418,59	192,18
3.418,60	192,18
4.271,58	384,10
4.271,59	384,10
4.271,60	384,11

O critério de análise de valor limite complementa o critério de particionamento de equivalência, aumentando a cobertura a situações limítrofes muito sujeitas a erros. Assim como o critério de particionamento de equivalência, a análise de valor limite é especialmente adequado para aplicações em que as variáveis de entrada podem ser facilmente identificadas e assumem valores específicos. Contudo, não é tão facilmente aplicável mesmo quando o domínio de entrada é simples, se o processamento do módulo for complexo.

A técnica de **teste funcional sistemático** combina as diretrizes do particionamento de equivalência e da análise de valor limite e define diretrizes adicionais. Primeiramente, requer ao menos dois casos de teste de cada partição para minimizar o problema de defeitos coincidentes que mascaram falhas. Além disso, define outras diretrizes, dentre elas:

- Se o domínio de entrada aceita valores numéricos e esses valores são discretos, ou seja, se fazem parte de um conjunto enumerável, então se devem definir casos de teste para testar todos os valores.
- Se o domínio de entrada aceita valores numéricos dentro de um certo intervalo, então se devem definir casos de teste para testar os extremos e um valor no interior do intervalo.
- Se o domínio de saída é de valores numéricos discretos, então se devem definir casos de teste gerando todos os valores.

- Se o domínio de saída é de valores numéricos dentro de um certo intervalo, então se devem definir casos de teste para gerar os extremos e pelo menos um valor no interior do intervalo.
- Para testar valores de entrada ilegais, devem ser incluídos casos de teste para verificar se o software os rejeita. Neste caso, diretrizes do critério de análise de valor limite devem ser empregadas.
- Definir casos de teste para explorar tipos ilegais que podem ser interpretados como valores válidos (p.ex., valor real para um campo inteiro).
- Definir casos de teste para explorar entradas válidas, mas que podem ser interpretadas como tipos ilegais (p.ex., números em campos que requerem caracteres).
- Definir casos de teste para explorar limites da representação binária dos dados (p.ex., para campos inteiros de 16 bits, selecionar os valores -32768 e +32767).

Para tratar o problema da correção coincidente, o teste funcional sistemático enfatiza a seleção de mais de um caso de teste por partição ou limite, aumentando, assim, a chance de revelar defeitos. Contudo, por ser baseado nos critérios de particionamento de equivalência e análise de valor limite, apresenta as mesmas limitações desses critérios (DELAMARO et al., 2007).

Em relação às técnicas de teste funcional de maneira geral, vale ressaltar que, como os critérios funcionais se baseiam apenas na especificação, não podem assegurar que partes críticas e essenciais do código tenham sido cobertas. Assim, é importante que outras técnicas sejam aplicadas em conjunto, para que o software seja explorado por diferentes pontos de vista (DELAMARO et al., 2007). Os testes estruturais são uma boa opção para este fim.

5.4.2 – Testes Estruturais

Os testes estruturais, ou de caixa-branca, estabelecem os objetivos do teste com base em uma dada implementação, verificando detalhes do código. Baseia-se no conhecimento da estrutura interna de um módulo ou programa. Caminhos lógicos internos são testados, estabelecendo casos de teste que põem à prova condições, laços, definições e usos de variáveis (DELAMARO et al., 2007; PRESSMAN, 2011).

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. Em geral, a maioria dos critérios estruturais utiliza uma representação de Grafo de Fluxo de Controle – GFC. Em um GFC, blocos disjuntos de comandos são considerados nós. A execução do primeiro comando do bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos em um bloco têm um único predecessor (possivelmente com exceção do primeiro) e um único sucessor (possivelmente com exceção do último) (DELAMARO et al., 2007).

A representação de um programa como um GFC estabelece uma correspondência entre nós e blocos de comandos e indica possíveis fluxos de controle entre blocos por meio de arcos. Um GFC é um grafo orientado com um único nó de entrada e um único nó de saída. Cada nó representa um bloco indivisível (sequencial) de comandos e cada arco representa um

possível desvio de um bloco para outro. A partir de um GFC, podem ser escolhidos os elementos que devem ser executados (DELAMARO et al., 2007).

Seja o caso de uma função que retorna o enésimo termo de uma série de Série de Fibonacci. A Série de Fibonacci tem a seguinte forma 1,1,2,3,5,8,13,21,34..., sendo o enésimo termo calculado da seguinte forma: se n = 1, $F_n = 1$; se n = 2, $F_n = 1$; se n > 2, $F_n = F_{n-1} + F_{n-2}$. A Figura 5.2 mostra uma possível implementação para esta função e seu correspondente GFC.

```
long fibonacci (int n) {

1       int i =  2;
2       long a =  1;
3       long b =  1;

4       if (n > 1) {
5          while (i < n) {
6             long temp  =  b;
7             b = b + a;
8             a = temp;
9             i = i + 1;
10         }
11      }
12      return (b);
   }
}
```

Nó	Bloco de Comandos
1	1 a 4
2	5
3	6 a 10
4	11 e 12

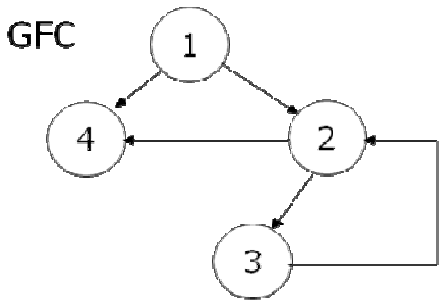


Figura 5.2 – Função que retorna o enésimo termo da Série de Fibonacci.

Dentre os principais critérios de teste estrutural podem ser citados os critérios baseados na complexidade e os critérios baseados no fluxo de controle.

Os **critérios baseados na complexidade** utilizam informações sobre a complexidade do programa para derivar casos de teste. O Critério de McCabe ou **Teste de Caminho Básico** é o critério baseado em complexidade mais conhecido. Caminhos linearmente independentes, i.e., que introduzam pelo menos um novo conjunto de instruções ou uma nova condição, estabelecem um conjunto básico para o GFC. Se os casos de teste puderem ser projetados de modo a forçar a execução dos caminhos do conjunto básico, então cada instrução terá sido executada pelo menos uma vez e cada condição terá sido executada com V e F.

O conjunto básico não é único. De fato, diferentes conjuntos básicos podem ser derivados para um GFC. O tamanho do conjunto básico é dado pela medida da complexidade ciclomática, que pode ser calculada, dentre outros, da seguinte forma: n^o_arcos – n^o_nós + 2 (DELAMARO et al., 2007). No exemplo do GFC da Figura 5.2, o tamanho do conjunto básico é 3 (5 – 4 + 2). Ou seja, 3 casos de teste são suficientes para testar os caminhos linearmente independentes, a saber: (i) 1 – 4 (n = 1); (ii) 1 – 2 – 4 (n = 2); e (iii) 1 – [2 – 3 – 2] – 4 (n > 2).

Os **critérios baseados no fluxo de controle**, como o próprio nome diz, utilizam apenas características de controle da execução do programa (comandos, condições e laços) para derivar casos de teste. Os principais critérios baseados no fluxo de controle são:

- Todos os nós: requer que cada nó do grafo (e por conseguinte cada comando do programa) seja executado pelo menos uma vez. Isso é o mínimo esperado de uma boa atividade de teste.
- Todos os arcos: requer que cada arco do grafo (cada desvio de fluxo de controle) seja exercitado pelo menos uma vez.
- Todos os caminhos: requer que todos os caminhos possíveis no programa sejam exercitados, o que muitas vezes é impraticável.

No exemplo do GFC da Figura 5.2, se for adotado o critério "todos os nós", basta um único caso de teste, com n > 2. Se for adotado o critério "todos os arcos", são necessários pelo menos dois casos de teste: n =1 (arco 1 – 4), e n > 2 (demais arcos). Já o critério "todos os caminhos", requer infinitos casos de teste.

O teste estrutural apresenta uma importante limitação: se um programa não implementa uma função, não existirá um caminho que corresponda a ela e nenhum dado de teste será requerido para exercitá-la. Assim, essa técnica deve ser empregada de modo complementar a outras técnicas (p.ex., técnicas funcionais). Além disso, alguns elementos requeridos (p.ex., uma certa combinação de arcos) podem não ser executáveis, sendo impossível derivar casos de teste para eles.

5.4.3 – Aplicando Técnicas de Teste

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 (MALDONADO; FABBRI, 2001). À primeira vista, pode parecer que realizando testes de caixa branca rigorosos poderíamos chegar a programas corretos. Contudo, isso pode não ser prático, uma vez que todas as combinações possíveis de caminhos e valores de variáveis teriam de ser exercitados, o que geralmente é impossível. Além disso, se um programa não implementa uma função definida em uma especificação, não existirá um caminho que corresponda a ela e nenhum dado de teste será requerido para exercitá-la. Assim, se existir um problema (a ausência da função), ele não será detectado. 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 de um módulo tenham sido exercitados pelo menos uma vez (PRESSMAN, 2011).

Uma boa abordagem para a introdução de práticas mais sistemáticas de teste em uma organização consiste em aplicar inicialmente critérios mais fracos, e talvez menos eficazes, porém menos custosos. Em função da disponibilidade de orçamento e de tempo, da criticidade de um programa e da maturidade da organização, poder-se-iam aplicar critérios mais fortes, e eventualmente mais eficazes, porém mais caros.

Todas as técnicas de teste se aplicam ao teste de unidade, sendo que os testes estruturais tem foco neste nível. Todas as técnicas de teste se aplicam ao teste de integração, com destaque para o teste funcional. Por fim, para testes de sistema, tipicamente aplicam-se técnicas de teste funcional.

5.5 – Processo de Teste

O processo de teste pode ser definido como um processo separado, mas intimamente ligado, ao processo de desenvolvimento. Isso porque eles têm metas e medidas de sucesso diferentes. Por exemplo, quanto menor a taxa de defeitos (razão entre o nº de casos de teste que falham pelo total de casos de teste), mais bem sucedido é considerado o processo de desenvolvimento. Por outro lado, quanto maior a taxa de defeitos, considera-se mais bem sucedido o processo de teste (MCGREGOR; SYKES, 2001).

O processo de teste envolve quatro atividades principais (PFLEEGER, 2004; MALDONADO; FABBRI, 2001):

- Planejamento de Testes: trata da definição dos requisitos de teste, 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. A seguir, 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. Uma vez, definidos os casos de teste, eles devem ser implementados.
- 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 (depuração). Não detectadas falhas, deve-se fazer uma avaliação final da qualidade dos casos de teste e definir pelo encerramento ou não da atividade de teste.

Em relação ao planejamento de teste, as seguintes questões devem ser respondidas:

- **Quem deve realizar os testes?** Para tratar esta questão, três abordagens podem ser utilizadas: (i) testes feitos pelos próprios desenvolvedores; (ii) testes feitos por testadores independentes; (iii) Compartilhamento de responsabilidades entre desenvolvedores e testadores independentes. Contudo, delegar toda a atividade de testes para desenvolvedores é perigoso, tendo em vista que o código é resultado de seu trabalho. Logo, procurar defeitos é, de certo modo, a destruição do mesmo, e o próprio desenvolvedor não tem motivação psicológica para projetar casos de teste que demonstrem que seu produto tem defeitos (dissonância cognitiva). Assim, dificilmente o desenvolvedor consegue criar casos de teste que rompam com a lógica de funcionamento de seu próprio código (KOSCIANSKI; SOARES, 2006). Testadores independentes são mais indicados, uma vez que são capazes de assumir uma perspectiva de testes. Contudo, se testadores independentes forem responsáveis por todos os testes, em todas as fases, isso pode tornar o processo lento e pouco produtivo. Assim compartilhar responsabilidades pode ser uma opção conciliatória. A divisão pode ser feita por: (i) fases, na qual desenvolvedores testam unidades, muitas vezes em paralelo com a implementação das mesmas,

enquanto testadores independentes realizam testes de integração e de sistema; ou (ii) por atividades do processo de teste, em que testadores independentes são responsáveis pelo planejamento e projeto de casos de teste, e os desenvolvedores são responsáveis pela construção e execução dos casos de teste.

- **O que testar?** Que partes do sistema devem ser mais cuidadosamente testadas? Esta resposta é obtida definindo-se os requisitos de teste. O Princípio de Pareto adaptado ao teste aponta que 20% dos componentes de software concentram 80% dos defeitos. Esse princípio sugere que os esforços devem ser concentrados nas partes mais importantes e/ou frágeis. Um bom teste é ao mesmo tempo econômico e encontra o máximo de defeitos (KOSCIANSKI; SOARES, 2006). Por exemplo, será que é necessário testar todas as unidades, inclusive unidades reutilizadas de outros projetos? McGregor e Sykes (2001) sugerem que pode não ser necessário testar unidades reutilizadas de outros projetos ou de bibliotecas. Além disso, algumas unidades não são fáceis de serem testadas individualmente, requerendo *drivers* e/ou *stubs* complexos. Assim, esses autores sugerem concentrar esforços em usos prováveis.
- **Quanto teste é necessário?** Conforme discutido anteriormente, é impossível testar tudo. Testes podem somente mostrar a presença de erros, mas não a sua ausência. Assim, o importante é ter uma boa cobertura nos testes. A cobertura de teste pode ser medida, dentre outras, de duas maneiras (MCGREGOR; SYKES, 2001): (i) Em relação a requisitos (quanto dos requisitos especificados foi testado? Considerar requisitos funcionais e não funcionais); (ii) Em relação à execução de linhas de código (teste de caminhos possíveis). O esforço de teste deve ser compensatório, ou seja, deve haver um balanceamento entre tempo/custo do teste e a quantidade de defeitos encontrados. O número de defeitos encontrados segue uma curva logarítmica, ou seja, embora ainda possam existir defeitos, o esforço para encontrá-los passa a ser muito grande, fazendo com que a atividade de teste comece a ser percebida como muito custosa (KOSCIANSKI; SOARES, 2006).
- **Quando testar?** Dentre as abordagens possíveis, destacam-se duas: (i) teste como uma atividade do processo de software (testar no final); (ii) teste como um processo paralelo ao processo de desenvolvimento. É recomendável planejar e projetar casos de teste à medida que o processo de desenvolvimento progride. A execução dos casos de teste pode ser feita em atividades destinadas para este fim. Deve-se levar em conta, ainda, que testes de unidade dependem da produção das unidades ao longo do processo de desenvolvimento. Testes de Integração dependem do ciclo de vida (incrementos / iterações) e da estrutura (subsistemas / casos de uso), podendo ser programados em intervalos específicos. Por fim, a execução de testes de sistema deve ocorrer nas entregas e, portanto, também é dependente do ciclo de vida.
- **Como testar?** Diz respeito à definição de técnicas de teste para cada fase de teste. Ainda que seja possível testar um sistema usando apenas uma técnica, é recomendável utilizar várias técnicas diferentes, sobretudo dependendo da fase. Testes estruturais, que utilizam o conhecimento da implementação, são mais adequados para testes de unidade, ainda que possam ser usados em alguns casos de integração. Testes funcionais, que utilizam o conhecimento da especificação,

podem ser utilizados em quaisquer níveis, sendo a principal opção para testes de integração e de sistema.

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.

A automatização do processo de teste é um importante fator para o sucesso no teste de software. O processo de teste tende a ser extremamente dispendioso e é muito importante utilizar ferramentas de apoio ao teste para buscar aumentar a produtividade. Há diversos tipos de ferramentas de teste. Contudo, ainda assim, muito trabalho humano é necessário para criar os casos de teste adequados aos critérios utilizados (DELAMARO et al., 2007).

Referências do Capítulo

DELAMARO, M.E., MALDONADO, J.C., JINO, M., *Introdução ao Teste de Software*, Série Campus – SBC, Editora Campus, 2007.

KOSCIANSKI, A., SOARES, M.S., *Qualidade de Software*, Editora Novatec, 2006.

MALDONADO, J.C., FABBRI, S.C.P.F., "Teste de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.

MCGREGOR, J.D., SYKES, D.A., *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001.

MYERS, G.J., *The Art of Software Testing*, 2nd edition, John Wiley & Sons, 2004.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, 2ª Edição, São Paulo: Prentice Hall, 2004.

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

Capítulo 6 – Entrega e Manutenção

Uma vez que o sistema é 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 requisições de mudanças, sejam de caráter corretivo, sejam de caráter evolutivo, certamente ocorrerão. De fato, como o mundo é dinâmico, ocorrendo mudanças de pessoas, de regras e processos de negócio, da estratégia das organizações e de tecnologia, dentre outros, é fundamental que os sistemas evoluam para acompanhar essas mudanças, de modo a continuamente satisfazer as necessidades de clientes e usuários.

6.1 - Entrega

A entrega não é meramente uma formalidade. No momento em que o sistema (ou uma versão dele) 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 (PFLEEGER, 2004).

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 e ajuda (*help*), preferencialmente on-line (PFLEEGER, 2004).

6.2 - 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 ou evolução (SANCHES, 2001).

Há muitas causas para a manutenção, dentre elas (SANCHES, 2001): falhas no processamento devido a erros no software, falhas de desempenho e outros requisitos não funcionais, alterações no ambiente de produção, mudanças nos processos de negócio, levando à necessidade de modificações em funções existentes, necessidade de inclusão de novas capacidades no sistema.

Ao contrário do que podemos pensar, a manutenção não é uma tarefa 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 (2004) 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 consequência 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 do Capítulo

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, 2ª Edição, São Paulo: Prentice Hall, 2004.

SANCHES, R., "Processo de Manutenção". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001.

Parte II

Gerência de Software

Capítulo 7 – Gerência da Qualidade

Uma das principais metas da Engenharia de Software é a produção de software de qualidade. Entretanto, que características um determinado produto precisa apresentar para considerarmos que o mesmo tem qualidade? Seja o exemplo de automóveis. Para se definir o conceito de qualidade de um automóvel, diversos fatores são levados em conta. Fatores como conforto, segurança, desempenho, beleza e custo têm estreita relação com a qualidade. No caso de produtos de software, características como adequação funcional, desempenho, confiabilidade, usabilidade, portabilidade e manutenibilidade estão diretamente relacionados à qualidade, mas o grau em que cada uma delas precisa ser atendida pode variar de sistema para sistema. Assim, por estes exemplos, podemos perceber que qualidade é um conceito relativo.

Ainda que qualidade seja um conceito relativo, é possível estabelecer duas diretrizes básicas, a saber:

- Qualidade está fortemente relacionada à conformidade com os requisitos.
- Qualidade diz respeito à satisfação do cliente.

Problemas no produto de software resultante podem decorrer de problemas na especificação dos seus requisitos ou na derivação dos diversos artefatos produzidos a partir dela. Uma coisa é fato: qualidade do produto de software não se atinge de forma espontânea. Ela tem de ser construída ao longo do processo de software. Assim, qualidade do produto de software está intimamente relacionada a diversos processos relacionados, dentre eles: Documentação, Verificação & Validação, Gerência de Configuração de Software e Medição.

Este capítulo aborda o tema Gerência da Qualidade de Software e está estruturado da seguinte forma: a Seção 7.1 trata da documentação produzida ao longo de processo de desenvolvimento de software; a Seção 7.2 retoma o tema Verificação & Validação (V&V), parcialmente abordado no Capítulo 5, o qual trata de Teste de Software, para discutir técnicas complementares de revisão, as quais se aplicam a documentos; a Seção 7.3 discute o processo de Gerência de Configuração de Software; finalmente, a Seção 7.4 trata de Medição aplicada à garantia da qualidade.

7.1 – Documentação de Software

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 (p.ex., plano de projeto), registrando requisitos e modelos do sistema (p.ex., documentos de especificação de requisitos e de projeto) e apoiando o uso do sistema gerado (p.ex., manual do usuário, ajuda, tutoriais).

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 (SANCHES, 2001a).

Assim, é importante planejar a documentação de um projeto, definindo, dentre outros, os documentos que serão gerados por cada atividade do processo, modelos de documentos a serem adotados e critérios de análise, aprovação ou reprovação. Algumas dessas atividades estão estreitamente relacionadas com o controle e a garantia da qualidade de software, outras com a gerência da configuração do software, conforme discutido na Seção 7.3.

Uma eficaz maneira de se melhorar a qualidade da documentação produzida em um projeto (e, por conseguinte, do sistema de software resultante) consiste em se definir padrões a serem aplicados na elaboração dos documentos. Os padrões aplicam-se aos artefatos produzidos ao longo do processo de software e podem ser, dentre outros, modelos de documentos, roteiros, normas e padrões de nomes, 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, subseções, 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, Documento de Especificação de Requisitos e Documento de 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 em que não é possível definir uma estrutura, tal como no código-fonte, normas e padrões de nomeação devem ser providos. 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 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 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.

7.2 – Verificação e Validação de Software por meio de Revisões

Durante o processo de desenvolvimento de software, ocorrem enganos, interpretações errôneas e outras falhas, principalmente provocados por problemas na comunicação e na transformação da informação, que podem resultar em mau funcionamento do sistema resultante. Assim, é muito importante detectar 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 (MALDONADO; FABBRI, 2001).

Conforme discutido no Capítulo 5, as atividades de controle e garantia da qualidade podem ser de dois tipos principais: Verificação e Validação (V&V). 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 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 satisfazem às necessidades dos clientes e usuários.

As atividades de V&V podem ser de dois tipos: estáticas e dinâmicas. Testes são análises dinâmicas, uma vez que envolvem a execução de código. Contudo, a execução do código requer, obviamente, que código fonte tenha sido produzido. Assim, a avaliação de um produto de software apenas por meio de testes é normalmente insuficiente. Conforme dito na introdução deste capítulo, a qualidade do produto de software tem de ser construída ao longo do processo. Assim, é fundamental realizar atividades de V&V estáticas, o que envolve a realização de revisões.

Uma Revisão de Software visa assegurar que um artefato produzido em uma fase possui qualidade suficiente para ser usado em uma fase subsequente. Revisões são tipicamente realizadas em documentos produzidos ao longo do processo de software, tais como documentos de requisitos, modelos diversos, especificações de projeto etc.

Quando realizadas por clientes e usuários, revisões visam à validação. Documentos de requisitos têm de ser submetidos à avaliação de clientes e usuários, pois, do contrário, há grande risco do sistema de software resultante não atender às suas reais necessidades.

Quando uma revisão é realizada por um membro da organização de desenvolvimento, a revisão é dita uma revisão por pares e o foco é a verificação. Assim, revisões por pares visam checar basicamente: (i) se padrões organizacionais de documentação estão sendo corretamente aplicados e (ii) se há consistência entre diferentes artefatos produzidos para representar a mesma coisa. Neste último caso, o foco pode ser a verificação da consistência de artefatos produzidos em uma mesma fase, tal como diferentes modelos conceituais detalhando requisitos previamente especificados (p.ex., consistência entre o modelo de entidade e relacionamento, o modelo de casos de uso e diagramas de estados produzidos para um mesmo sistema), ou a verificação da consistência de artefatos produzidos em fases diferentes, tal como verificar a consistência de modelos produzidos na fase de projeto contra os correspondentes modelos de análise (p.ex., consistência entre um modelo de entidades e relacionamentos e o modelo relacional dele derivado no projeto de dados).

Nas revisões, processos, documentos e outros artefatos são revisados por um grupo de pessoas, com o objetivo de avaliar 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 dos usuários. Assim, o objetivo de uma revisão é detectar erros e inconsistências em artefatos e processos, sejam eles relacionados à forma, sejam eles relacionados ao conteúdo, e apontá-los aos responsáveis pela sua elaboração.

Uma revisão é dita uma revisão técnica formal quando segue um processo bem definido. 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 ser 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.

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.

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 consistência e 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, o resultado das atividades de garantia da qualidade é fortemente dependente da documentação produzida.

7.3 – Gerência de Configuração de Software

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, também podem ser substituídos por versões mais recentes ou mesmo por outras 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 repercutir 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.

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 permitir 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.

Dentre as principais funções da GCS, destacam-se:

- 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.
- Gerenciamento de Liberação e Entrega: diz respeito à construção do produto de software (ou de partes dele), produzindo itens de configuração (ICs) derivados a partir de ICs fonte; liberação, identificando as versões particulares de cada IC que serão disponibilizadas; e entrega, implantando os produtos de software no ambiente final de execução.

7.3.1 - O Processo de GCS

O primeiro passo do processo de GCS 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 (ICs). Os itens mais relevantes para serem submetidos à gerência de configuração são aqueles mais usados durante o ciclo de vida, os mais importantes para segurança, os projetados para reutilização e os que podem ser modificados por vários desenvolvedores ao mesmo tempo (SANCHES, 2001b). 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 é 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 (PRESSMAN, 2011; SANCHES, 2001b).

Após a identificação dos ICs, 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 ICs 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 (PRESSMAN, 2011; SANCHES, 2001b).

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 ICs que são criadas durante o processo de software (PRESSMAN, 2011; SANCHES, 2001b). A ideia é que a cada modificação que ocorra em um IC, 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 (SANCHES, 2001b).

Uma das mais importantes atividades do processo de GCS é o controle de alterações. O controle de alterações combina procedimentos humanos e ferramentas automatizadas para controlar alterações realizadas em ICs (PRESSMAN, 2011). Assim que uma alteração é solicitada, o impacto em outros itens e o custo para a modificação devem ser avaliados. Um responsável deve decidir se a alteração deverá ou não ser realizada. Caso a alteração seja liberada, pessoas são indicadas para sua execução. Assim que não houver ninguém utilizando os ICs 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* (PRESSMAN, 2011; SANCHES, 2001b).

Porém, mesmo com uma aplicação bem sucedida dos mecanismos de controle, não é possível garantir que as modificações foram corretamente implementadas. Assim, revisões e auditorias de configuração de software são necessárias (PRESSMAN, 2011). 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 (SANCHES, 2001b).

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 foi 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 (SANCHES, 2001b).

7.4 – Medição de Software

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 (PRESSMAN, 2011).

Avaliações da qualidade assumem particularidades que começam pelo tipo de entidade que se está avaliando e vão até diferentes características a serem avaliadas e métodos de

avaliação. No entanto, é possível perceber também aspectos comuns, dentre eles a forte relação com a medição. Inicialmente, é necessário definir as entidades que serão avaliadas e quais de suas características serão usadas na avaliação. Segundo, é preciso definir quais medidas serão usadas para quantificar essas características. Uma vez planejada a medição, pode-se passar efetivamente à sua execução, o que envolve a aplicação de procedimentos de medição para se obter um valor para uma medida da entidade em questão. Uma vez realizada a medição, devem-se analisar seus resultados para avaliar as entidades, gerando observações, tais como problemas e não conformidades detectados. Essas observações devem dar origem a ações para tratar os problemas detectados.

Uma medida fornece uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de um atributo de um produto ou de um processo. Uma medida é dita uma medida base, quando ela é funcionalmente independente de outras medidas (p.ex., quantidade de erros descobertos em uma revisão). Uma medida é dita uma medida derivada, quando ela é definida como uma função de duas ou mais medidas (p.ex., número de erros encontrados por linha de código). Medidas derivadas procuram relacionar medidas base com o objetivo de se prover uma ideia da eficácia do processo, projeto ou produto sendo medido. Diz que uma medida é um indicador, quando ela é usada para se ter uma compreensão do processo, produto ou projeto sendo medido. Medição é o ato de medir, isto é, de determinar um valor para uma medida.

Seja o seguinte exemplo: deseja-se saber se uma pessoa está com seu peso ideal ou não. Para tal, duas medidas base são importantes: altura (H) e peso (P). Ao medir essas dimensões, está-se efetuando uma medição. A medida derivada "índice de massa corporal (IMC)" é calculada segundo a seguinte fórmula: $IMC = P / H^2$. A partir dessa medida, 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 a seguir:

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 seus valores de peso e altura. A partir desses valores, ela pode calcular a medida derivada 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 modo de vida (alimentação, prática de exercícios físicos etc.), visando a melhorias reais para sua saúde.

Uma vez que a medição de software se preocupa em obter valores numéricos 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. Esse modelo de qualidade é subdividido em dois modelos: (i) o modelo de qualidade para características externas e internas e (ii) o modelo de qualidade para qualidade em uso. O primeiro classifica os atributos de qualidade de software em seis características (funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade) que são, por sua vez, desdobradas em subcaracterísticas. As subcaracterísticas podem ser desdobradas em mais níveis, até se ter subcaracterísticas diretamente mensuráveis, para as

quais medidas são aplicadas. As normas ISO/IEC 9126-2 e 9126-3 apresentam, respectivamente, medidas externas e internas.

Esse modelo de qualidade preconiza a análise de características de qualidade a partir de suas subcaracterísticas de forma recursiva até que se tenham medidas para as quais seja possível coletar dados. Esse modelo é aplicável não somente a produtos de software, 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 subcaracterísticas de qualidade relevantes que tenham influência sobre a mesma, estabelecendo um modelo ou fórmula de computar a característica a partir das subcaracterísticas. Fórmulas baseadas em peso são bastante utilizadas: $cq = p_1 * scq_1 + ... + p_n * scq_n$.
3. Usar procedimento análogo ao anterior para as subcaracterísticas não passíveis de mensuração direta.
4. Para as subcaracterísticas diretamente mensuráveis, selecionar medidas, coletar dados e computar as medidas segundo a fórmula ou modelo estabelecido.
5. Fazer o caminho inverso, agora computando subcaracterísticas 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 da qualidade.

Vale destacar que, especialmente no caso da avaliação da qualidade de software, medidas relacionadas a defeitos são bastante úteis, tal como número de erros por linhas de código.

O único modo racional de melhorar um produto ou processo é medir atributos específicos, obter um conjunto de medidas significativas baseadas nesses atributos e usar os valores medidos para fornecer indicadores que conduzirão um processo de melhoria da qualidade (PRESSMAN, 2011).

Referências do Capítulo

MALDONADO, J.C., FABBRI, S.C.P.F., " 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.

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

SANCHES, R., "Documentação de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001a.

SANCHES, R., "Gerência de Configuração de Software". In: *Qualidade de Software: Teoria e Prática*, Eds. A.R.C. Rocha, J.C. Maldonado, K. Weber, Prentice Hall, 2001b.

Capítulo 8 – Gerência de Projetos

Antes de contratar o desenvolvimento de um software, geralmente, um cliente quer saber se seu fornecedor é capaz de realizar esse trabalho, quanto o projeto custará e qual será a sua duração. Para responder a essas perguntas, é necessário definir o escopo do projeto, através de um levantamento preliminar de requisitos, realizar estimativas, levantar riscos, alocar recursos e definir cronograma de execução e orçamento. Todas essas informações são registradas em um documento, chamado Plano de Projeto, que deve ser sistematicamente revisado ao longo do projeto, de modo a permitir acompanhar o progresso e tomar ações corretivas, no caso de se detectar desvios em relação ao inicialmente planejado. Esse conjunto de atividades faz parte da Gerência de Projetos de software.

8.1 – Projeto de Software e Gerência de Projetos

Projeto, como definido pelo PMBOK[1], é um empreendimento temporário com o objetivo de criar um produto ou serviço único (VIEIRA, 2003). É um trabalho que visa à criação de um produto ou à execução de um serviço específico, temporário, não repetitivo e que envolve um certo grau de incerteza na sua realização (MARTINS, 2005). Normalmente, é caracterizado por uma sequência de atividades (o processo do projeto), sendo executada por pessoas dentro de limitações de tempo, recursos (no caso de projetos de software, sobretudo, recursos humanos) e custos.

Assim sendo, a Gerência de Projetos de Software envolve, dentre outros, o planejamento e o acompanhamento das **pessoas** envolvidas no projeto, do **produto** sendo desenvolvido e do **processo** seguido para evoluir o software de um conceito preliminar até uma implementação concreta e operacional (PRESSMAN, 2011).

8.1.1 – As Pessoas

Em um projeto de software, há várias pessoas envolvidas, exercendo diferentes papéis, tais como: Gerente de Projeto, Desenvolvedor (Analistas, Projetistas, Programadores, Testadores), 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

[1] O PMBOK (*Project Management Body of Knowledge* – Corpo de Conhecimento em Gerência de Projetos) é um guia de orientação do conhecimento envolvido na gerência de projetos, cujo objetivo é identificar e descrever conceitos e práticas da gerência de projetos em geral, padronizando a terminologia e os processos adotados nesta área de estudo. Esse documento foi produzido e é periodicamente atualizado pelo PMI (*Project Management Institute* – Instituto de Gerência de Projetos), uma entidade internacional sem fins lucrativos que congrega profissionais atuando na área de gerência de projetos (MARTINS, 2005).

de equipes é importante para uma ampla variedade de situações, tal como uma formação de 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 interpessoais, tipo do projeto, criatividade etc.

No que se refere à organização / estrutura das equipes, há diversas formas de se estruturar equipes. A maneira mais tradicional consiste em estabelecer uma hierarquia de autoridade, na qual o projeto possui um líder, o qual é responsável por atribuir as tarefas e acompanhar o andamento do projeto. Nesta estrutura, a comunicação entre o líder e os demais membros da equipe é vertical. Esta estrutura pode se tornar mais flexível, mantendo a figura do líder de projeto, mas estabelecendo uma comunicação mais horizontal, na qual os membros da equipe têm mais liberdade e poder de decisão. Uma maneira bem mais flexível consiste em não haver um líder permanente e as decisões serem tomadas por consenso do grupo. A comunicação entre os membros da equipe é horizontal. Geralmente, projetos de inovação são organizados desta maneira última maneira, pois ela estimula a criatividade, a colaboração e a participação engajada no projeto.

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.

8.1.2 – O Produto

Na gerência de projetos, um gerente se depara, logo no início, com um sério problema: são necessárias estimativas quantitativas (de tempo e custo) e um plano organizado do trabalho a ser feito. Entretanto, não há informação suficiente para tal. Assim, a primeira coisa a fazer é definir o escopo do software, realizando um levantamento preliminar de requisitos. Neste contexto, ganha força a ideia de 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 submódulos ou funções, até que se tenha uma visão geral das funcionalidades a serem tratadas no projeto. Características especiais relacionadas a essas funções devem ser apontadas, tais como requisitos de desempenho.

8.1.3 – O Processo

Para poder ser gerenciado, um projeto tem de ser planejado em termos das atividades a serem realizadas, definindo, dentre outros, os responsáveis pela sua realização e os produtos de trabalho a serem produzidos. Os modelos de processo discutidos no Capítulo 2 podem estabelecer uma base para a definição do processo de projeto, mas há de se levar em conta que cada projeto tem características únicas e que, portanto, além de selecionar o modelo de processo mais indicado para o projeto em questão, o gerente de projeto precisa, ainda, adaptá-lo para as reais necessidades do projeto.

A decomposição do processo de software em subprocessos e a decomposição destes em diferentes níveis de atividade é a base para a definição do processo.

8.1.4 - Estrutura Analítica do Projeto

Uma boa gerência de projetos precisa fundir as visões de produto e processo. 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 Analítica do Projeto – EAP (*Work Breakdown Structure* – WBS), considerando essa duas dimensões - produto e processo, como ilustra a Tabela 8.1.

Tabela 8.1 – Estrutura de Divisão do Trabalho considerando a fusão das visões de produto e processo.

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

Uma EAP fornece um esquema para identificação e organização das unidades lógicas de trabalho a serem gerenciadas, que são chamadas de "pacotes de trabalho" (*work packages*).

8.2 – O Processo de Gerência de Projetos de Software

Como ilustra a Figura 8.1, tipicamente, um processo de gerência de projetos envolve quatro atividades principais:

- Iniciação: é quando é realizada a autorização formal para que o projeto seja iniciado.
- Planejamento: um plano organizado de como o projeto será conduzido deve ser elaborado. O planejamento do projeto deve tratar da definição do escopo do software, da definição do processo de software do projeto, da realização de estimativas, da elaboração de cronograma e orçamento, e da identificação e tratamento dos riscos associados ao projeto.
- Acompanhamento: conforme anteriormente apontado, no início do projeto há pouca informação disponível, o que pode comprometer a precisão do escopo identificado, das estimativas realizadas e, por conseguinte, do cronograma e do orçamento elaborados. À medida que o trabalho avança (execução do projeto), maior conhecimento se tem e, portanto, é possível refinar e ajustar esses elementos. Além disso, projetos são dinâmicos e, portanto, estão sujeitos às mudanças que ocorrem no contexto em que o produto será inserido. Sendo assim, é

fundamental acompanhar o progresso do trabalho, refinar escopo e estimativas, alterar o processo do projeto, o cronograma e o orçamento, além de monitorar riscos e tomar ações corretivas. Deste modo, as atividades realizadas (sumarizadas na Figura 8.1) no planejamento, são novamente consideradas no acompanhamento do projeto, que tipicamente se dá nos marcos definidos no projeto.

- Encerramento: terminado o projeto, a gerência ainda tem um importante trabalho a fazer: fazer uma análise crítica do que deu certo e o que não funcionou, procurando registrar lições aprendidas de sucesso e oportunidades de melhoria. Comparações entre valores estimados e realizados, identificação de problemas que ocorreram e causas dos desvios devem ser discutidas com os membros da equipe, procurando fazer com que haja um aprendizado, não só da equipe, mas da organização como um todo. Uma técnica bastante empregada neste contexto é a análise *post-mortem*.

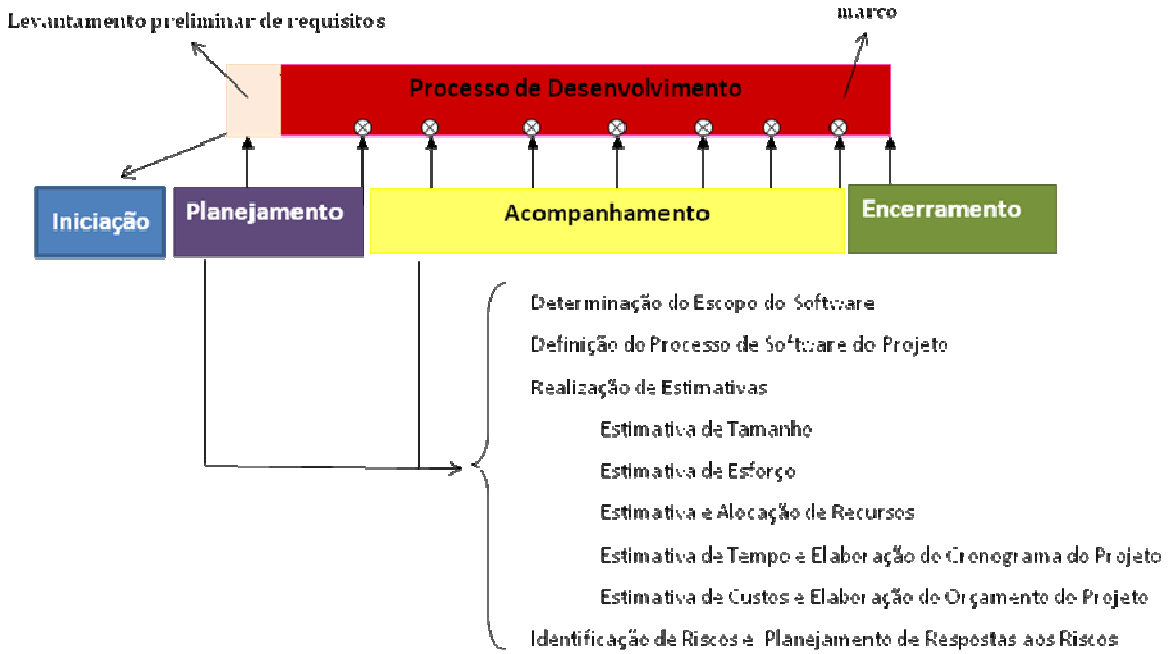


Figura 8.1 – O Processo de Gerência de Projetos de Software.

As atividades relacionadas no quadro na parte inferior na Figura 8.1 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, duração e custo).

Na sequência, cada uma das atividades inerentes ao planejamento e acompanhamento de projetos é discutida.

8.3 – Determinação do Escopo do Software

A primeira atividade de gerência em um projeto de software consiste na determinação do escopo do produto de software a ser desenvolvido (PRESSMAN, 2011). Basicamente, o escopo do produto é composto pela especificação de um conjunto de funcionalidades (requisitos funcionais) associada a outras características desejadas (requisitos não funcionais), tais como desempenho, confiabilidade etc.

Para que o escopo do software seja determinado, um levantamento preliminar de requisitos deve ser realizado. O escopo pode ser documentado de várias formas, sempre contendo uma breve descrição das funções do sistema, requisitos não funcionais importantes e o contexto e objetivos do sistema.

8.4 – Definição do Processo de Software do Projeto

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.

O processo de software não pode ser definido de forma universal. Para ser eficaz e conduzir à construção de produtos de boa qualidade, o processo deve ser adequado às especificidades do projeto em questão. Deste modo, processos devem ser definidos caso a caso, considerando-se as características da aplicação (domínio do problema, tamanho, complexidade etc), a tecnologia a ser adotada na sua construção (paradigma de desenvolvimento, linguagem de programação, mecanismo de persistência etc), a organização onde o produto será desenvolvido e a equipe de desenvolvimento.

Há vários aspectos a serem considerados na definição de um processo de software. No centro da arquitetura de um processo de desenvolvimento estão as suas atividades-chave: análise 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 (ou modelo de processo), o detalhamento (decomposição) de suas macro-atividades, a escolha de métodos, técnicas e roteiros (procedimentos) para a sua realização, e a definição de recursos (p.ex., hardware e software), papéis responsáveis (p.ex., analista, programador) e artefatos requeridos e produzidos.

A escolha de um modelo de ciclo de vida (ou modelo de processo) é o ponto de partida para a definição do processo de desenvolvimento. Conforme discutido no Capítulo 2, um modelo de processo organiza as macro-atividades básicas do processo de desenvolvimento, estabelecendo precedência e dependência entre as mesmas. Para a definição completa do processo de desenvolvimento, cada atividade descrita no modelo de processo deve ser decomposta e, para suas subatividades, devem ser associados métodos, técnicas, ferramentas e critérios de qualidade, entre outros, formando uma base sólida para o desenvolvimento.

Adicionalmente, outros processos, tipicamente de cunho gerencial, devem ser definidos, dentre eles processos de gerência de projetos e de garantia da qualidade.

8.5 - Estimativas

Antes mesmo de serem iniciadas as atividades do processo de desenvolvimento, o gerente e a equipe do projeto devem estimar o trabalho a ser realizado, os recursos necessários, a duração e, por fim, o custo do projeto. Apesar das estimativas serem um pouco de arte e um pouco de ciência, essa importante atividade não deve ser conduzida de forma desordenada, afinal as estimativas são a base para todas as outras atividades do planejamento de projeto.

Para alcançar boas estimativas de esforço, prazo e custo, existem algumas opções (PRESSMAN, 2011):

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 (planejamento do projeto). No entanto, deve-se reconhecer que quanto mais tarde forem feitas as estimativas, maior é o conhecimento que se tem sobre o projeto e por conseguinte menores as chances de se cometer erros. Assim, é fundamental revisar as estimativas na medida em que o projeto avança (acompanhamento do projeto).

Técnicas de decomposição, a segunda opção, usam 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), estimativas são geradas para porções do projeto, ditos pacotes de trabalho. Assim, a Estrutura Analítica do Projeto, como a mostrada na Tabela 8.1, 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, pontos de função, dentre outras medidas). 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 são úteis e podem ser usados em conjunto com informações históricas da organização para 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 sequência, alocam-se os recursos necessários, elabora-se o cronograma do projeto (estimativa de duração) e, por fim, estima-se o custo do projeto e define-se o seu orçamento (cronograma de desembolso).

8.5.1 – Gerência de Projetos e Medição

É muito importante observar a estreita relação entre gerência de projetos e medição. Para acompanhar o andamento do projeto, é preciso medir o progresso e comparar com o estimado. Mesmo no planejamento, sobretudo quando se pretende utilizar dados de projetos anteriores, dados de medidas são muito importantes. Não é possível controlar o que não se pode medir e, principalmente, só é possível chegar a boas estimativas com base em dados históricos, se dados forem coletados criteriosamente. Assim, as medidas dão visibilidade ao estado do projeto, permitindo tanto saber para onde ir no início do projeto quanto verificar se o rumo está correto, tomando ações corretivas quando necessário (VAZQUEZ; SIMÕES; ALBERT, 2005).

Mas não basta coletar dados aleatoriamente. Conforme discutido no Capítulo 7, para que possam ser usados eficientemente, dados têm de ser arranjados de modo a prover indicadores. Por exemplo, o que se pode dizer a respeito da qualidade de um produto de software que tenha apresentado cinco defeitos depois de implantado? É boa? Não? Isoladamente, esse dado pouco diz. Neste caso, combinar dados em medidas, obtendo medidas derivadas, é uma boa opção. No exemplo anterior, se combinássemos a medida de número de defeitos com uma medida de tamanho (tal como linhas de código – LOC), teríamos a medida derivada "número de defeitos por linha de código" capaz de nos dizer bem mais do que os dados isolados. Se agora os cinco defeitos medidos fossem em um software de 10.000 linha de código, chegar-se-ia ao valor de 0,5 defeito/KLOC. A partir dessa medida, comparando-a com indicadores da organização, aí sim poder-se-ia chegar a alguma conclusão sobre a qualidade do produto.

Em uma abordagem desta natureza, os resultados da medição permitem uma comunicação efetiva com os vários interessados no desenvolvimento. A falta de medidas de projeto, por outro lado, prejudica de forma geral o seu acompanhamento, uma vez que pode haver um problema, mas ele não está sendo percebido por aqueles que podem direcionar esforços para a sua solução. Assim, medidas têm um importante papel na rápida identificação e correção de problemas ao longo do desenvolvimento do projeto. Com a sua utilização, fica muito mais fácil justificar e defender as decisões tomadas. Afinal o gerente de projeto não decidiu apenas com base em seu sentimento e experiência, mas também fundamentado na avaliação de indicadores que refletem uma tendência de comportamento futuro. Essa tendência não é derivada apenas das experiências no projeto corrente, mas também de experiências semelhantes de outros projetos da organização (conhecimento organizacional) e até mesmo de fora dela (VAZQUEZ; SIMÕES; ALBERT, 2005).

No que tange à gerência de projetos, estabelecer classes de projetos e coletar algumas medidas pode ser bastante importante para apoiar a realização de estimativas. Por exemplo, se uma organização tem indicadores para produtividade (tamanho/esforço) e custo (R$/tamanho) para diversas classes de projetos diferentes, é possível, a partir de uma estimativa de tamanho, chegar a estimativas de esforço e custo. Dada a importância da estimativa de tamanho nessa abordagem, ela é, geralmente, a primeira estimativa a ser realizada.

8.5.2 - Estimativa de Tamanho

Ainda que anteriormente o tamanho tenha sido basicamente utilizado para normalizar indicadores de produtividade, custo e qualidade, mesmo isoladamente pode ser uma medida importante, como, por exemplo, na contratação de serviços de desenvolvimento e manutenção de software.

Dois modelos de contratação são bastante difundidos atualmente (VAZQUEZ; SIMÕES; ALBERT, 2005): preço global fixo e por preço unitário por homem-hora. No primeiro, um preço total fixo é estabelecido por um produto ou serviço bem definido. O grande problema desse modelo é que, normalmente, é alta a probabilidade de haver um aumento do escopo inicialmente previsto. A questão passa a ser: incorporar ou não novos requisitos ao produto? Se a decisão for por incorporar novos requisitos, quem vai pagar a conta? Afinal, aumento do escopo implica em aumento no trabalho a ser realizado. Renegociar contratos nem sempre é fácil. Além disso, as alterações nos requisitos podem ser muitas (e às vezes pequenas), sendo inviável a realização de tantas renegociações. Se a decisão for por não incorporar novos requisitos, o resultado final pode ser ainda mais desastroso: a insatisfação do cliente.

No modelo baseado em homem-hora, o risco passa a ser outro. Se a organização responsável pelo fornecimento do produto ou serviço é pouco produtiva, ela não é penalizada. Muito pelo contrário. Ela recebe ainda mais para fazer o mesmo serviço (VAZQUEZ; SIMÕES; ALBERT, 2005).

Uma forma alternativa para contratação consiste em uma variação do segundo modelo na qual o preço unitário é pago não mais por homem-hora (uma unidade de esforço), mas por uma unidade de tamanho. Assim, é possível acomodar mudanças no esforço, mas sem os desvios observados na modalidade por homem-hora. A questão passa a ser, então, que medida usar para medir tamanho.

Entre as várias formas de se medir tamanho de um software, uma das mais simples, direta e intuitiva 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 medida e o tempo necessário para se desenvolver um sistema. Entretanto, o uso de LOCs apresenta várias desvantagens. Primeiro, verifica-se que ela é fortemente ligada à tecnologia empregada, sobretudo a linguagem de programação e ao estilo do código escrito. 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. Por fim, essa medida é pouco significativa para o cliente.

Visando possibilitar a realização de estimativas de tamanho mais cedo no processo de software, e sem os problemas de dependência em relação à tecnologia, foram propostas outras medidas 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.

A Análise de Pontos de Função (APF) é um método padrão para a medição do tamanho de uma aplicação de software, que estabelece uma medida de tamanho do software em Pontos de Função (PFs). Os objetivos da APF são (HAZAN, 2001):

- Medir as funcionalidades do sistema requisitadas e recebidas pelo usuário;
- Medir projetos de desenvolvimento e manutenção de software, sem se preocupar com a tecnologia que será utilizada na implementação.

O processo para contagem de PFs compreende sete passos, mostrados na Figura 8.2 e descritos sucintamente adiante. Uma descrição um pouco mais detalhada do método da Análise de Pontos de Função é apresentada no Anexo A. Para uma visão completa do método, recomenda-se a leitura de (VAZQUEZ; SIMÕES; ALBERT, 2005).

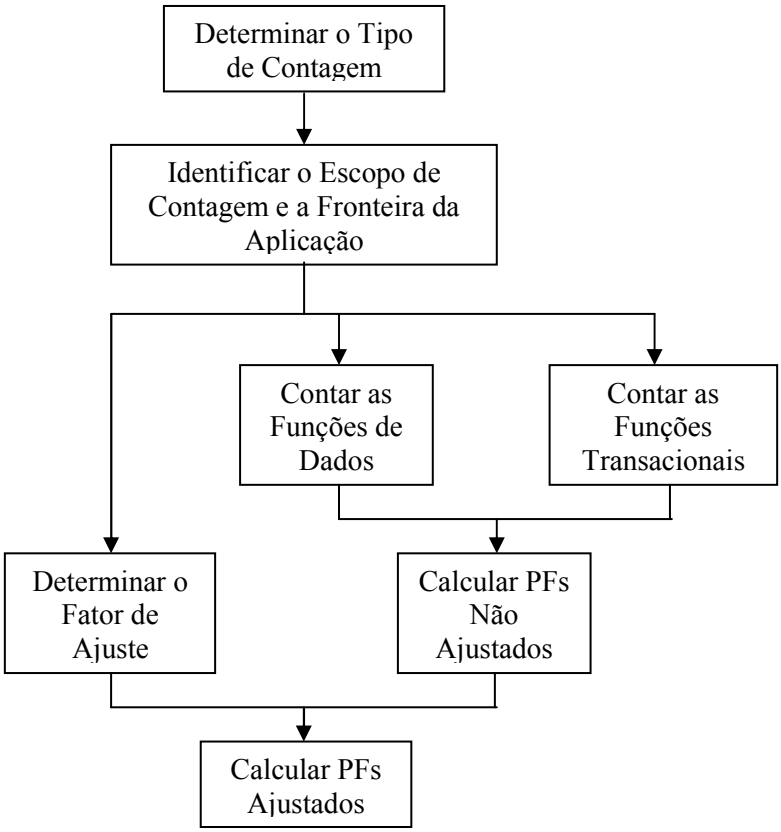


Figura 8.2 – Passos do Método de Análise de Pontos de Função.

- **Determinar o tipo de contagem de pontos de função**: este é o primeiro passo no processo de contagem. Existem três tipos de contagem: contagem de PF de projeto de desenvolvimento, de aplicações instaladas e de projetos de manutenção.
- **Identificar o escopo de contagem e a fronteira da aplicação**: neste passo, definem-se as funcionalidades que serão incluídas em uma contagem de PFs específica. A fronteira da aplicação é definida estabelecendo um limite lógico entre a aplicação que está sendo medida, os usuários e outras aplicações. O escopo de contagem define a parte do sistema (funcionalidades) a ser contada.

- **Determinar a contagem de pontos de função não ajustados**: os pontos de função não ajustados (PFNA) refletem as funcionalidades fornecidas pelo sistema para o usuário. Essa contagem leva em conta dois tipos de funções: funções de dados e funções transacionais, bem como sua complexidade (simples, média ou complexa).
- **Contar as funções de dados**: as funções de dados representam as necessidades de dados internos e externos à aplicação, sendo considerados arquivos lógicos internos e os arquivos de interface externa. Ambos são grupos de dados logicamente relacionados ou informações de controle que foram identificados pelo usuário. A diferença está no fato de um **Arquivo Lógico Interno (ALI)** ser mantido dentro da fronteira da aplicação, isto é, armazenar os dados mantidos através de um ou mais processos elementares da aplicação, enquanto um **Arquivo de Interface Externa (AIE)** é apenas referenciado pela aplicação, ou seja, ele é mantido dentro da fronteira de outra aplicação. Assim, o objetivo de um AIE é armazenar os dados referenciados por um ou mais processos elementares da aplicação sendo contada, mas que são mantidos por outras aplicações.
- **Contar as funções transacionais**: as funções transacionais representam as funcionalidades efetivamente fornecidas para o usuário, sendo categorizadas em três tipos: entradas externas, saídas externas e consultas externas. As **Entradas Externas (EEs)** são processos elementares que processam dados (ou informações de controle) que entram pela fronteira da aplicação. O objetivo principal de uma EE é manter um ou mais ALIs ou alterar o comportamento do sistema. As **Saídas Externas (SEs)** são processos elementares que enviam dados (ou informações de controle) para fora da fronteira da aplicação. Seu objetivo é mostrar informações recuperadas através de um processamento lógico (isto é, que envolva cálculos ou criação de dados derivados) e não apenas uma simples recuperação de dados. Uma SE pode, também, manter um ALI ou alterar o comportamento do sistema. Por fim, uma **Consulta Externa (CE)**, assim como uma SE, é um processo elementar que envia dados (ou informações de controle) para fora da fronteira da aplicação, mas sem a realização de nenhum cálculo nem a criação de dados derivados. Seu objetivo é apresentar informação para o usuário, por meio apenas de recuperação de informações. Nenhum ALI é mantido durante sua realização, nem o comportamento do sistema é alterado.
- **Calcular os pontos de função não ajustados**: Para calcular os pontos de função não ajustados, é preciso identificar a complexidade e a contribuição, em pontos por função, de cada uma das funções contadas. Para tal, são utilizadas tabelas de complexidade específicas para cada tipo de função (ver Anexo A).
- **Determinar o valor do fator de ajuste**: o fator de ajuste é baseado em 14 características gerais de sistemas, que avaliam a funcionalidade geral da aplicação que está sendo contada e seus níveis de influência. O nível de influência de uma característica é determinado com base em uma escala de 0 (nenhuma influência) a 5 (forte influência). Assim, o fator de ajuste visa ajustar os pontos de função não ajustados em ± 35%. Esse passo tornou-se opcional em 2002 para que o método da APF passasse a ser um padrão internacional de medição funcional (ISO/IEC 20926). As principais críticas são a grande variação na interpretação das 14 características gerais de sistemas e a constatação que algumas delas estão desatualizadas.

- **Calcular os pontos de função ajustados**: finalmente, os PFs ajustados são calculados, considerando-se o tipo de contagem definido no primeiro passo e o fator de ajuste.

A Figura 8.3 apresenta uma visão esquemática dos tipos de função que são considerados na contagem da APF.

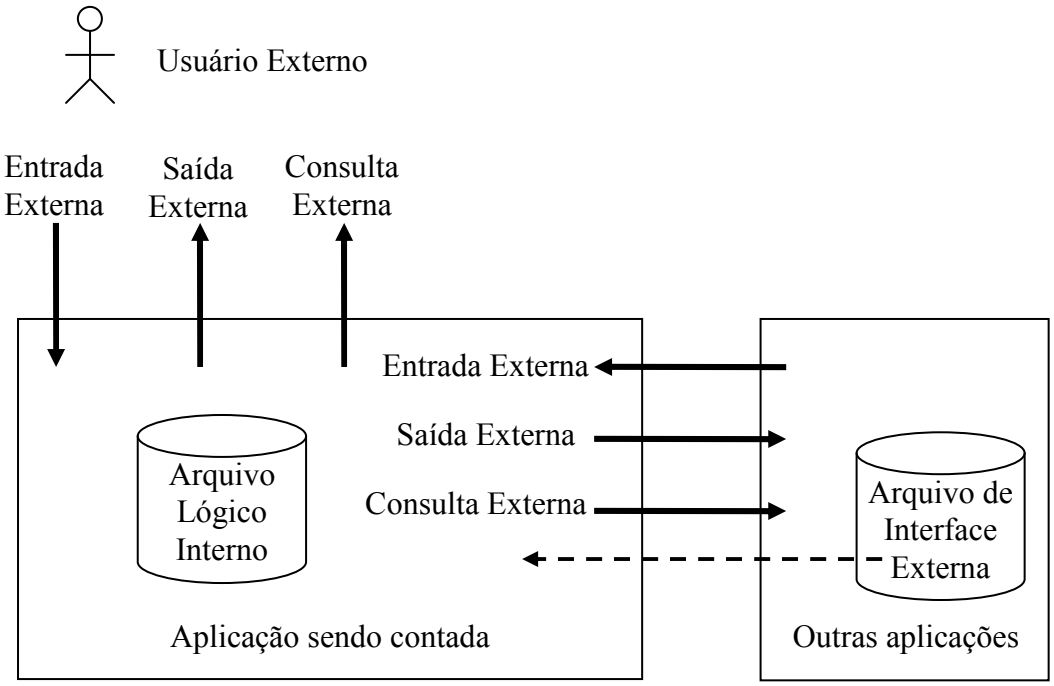


Figura 8.3 – Visão Esquemática das Funções de uma Aplicação segundo a APF.

Ainda que a obtenção dos pontos de função seja dependente unicamente do conhecimento das funcionalidades requeridas e não da tecnologia a ser empregada, o maior problema da APF é que os dados necessários para essa análise 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, pode ser difícil utilizar o método da APF integralmente para realizar as primeiras estimativas. A APF é, antes de mais nada, um método de medição e, portanto, seu uso, da forma como proposta, para a realização de estimativas de tamanho é uma adaptação.

Tendo em vista isso, foram propostas algumas variações mais simples do método voltadas para a realização de estimativas e que apresentam resultados satisfatórios, dentre elas a *Contagem Indicativa*, o uso de *Complexidades Médias* e a *Contagem Estimativa*. Todas consideram apenas PFs não ajustados (VAZQUEZ; SIMÕES; ALBERT, 2005).

Na Contagem Indicativa, é necessária apenas a identificação das funções de dados (ALIs e AIEs). Considera-se, então, 35 PFs para cada ALI e 15 PFs para cada AIE identificados, ou seja o número de PFs não ajustados é dado por: nPFNA = 35* nALI + 15* nAIE.

Nos casos de Complexidades Médias e Contagem Estimativa, é necessário identificar todas as funções de dados (ALIs e AIEs) e transacionais (EEs, CEs e SEs), mas não é necessário usar as tabelas de complexidade. No caso das Complexidades Médias do ISBSG

Benchmark, considera-se 7,4 PFs para cada ALI, 5,5 PFs para cada AIE, 4,3 PFs para cada EE, 5,4 PFs para cada SE e 3,8 PFs para cada CE. Assim, o número de PFs não ajustados é dado por: nPFNA = 7,4* nALI + 5,5* nAIE + 4,3* nEE + 5,4* nSE + 3,8*nCE. No caso da Contagem Estimativa da NESMA, os pesos são um pouco diferentes, sendo o número de PFs não ajustados dado por: nPFNA = 7* nALI + 5* nAIE + 4* nEE + 5* nSE + 4*nCE.

Obviamente, a contagem detalhada de pontos de função é mais precisa que os demais tipos de contagem. Entretanto, requer um tempo maior para ser realizado e depende de especificações mais detalhadas que, na grande maioria das vezes, não existem no início de um projeto. Vale destacar que estudos realizados pela NESMA com mais de 100 projetos apontam que a contagem estimativa tem menor dispersão em relação à contagem detalhada do que a contagem indicativa.

Assim, uma boa opção é iniciar as estimativas com uma técnica simplificada, tal como a Contagem Estimativa da NESMA, e, à medida que um maior entendimento dos requisitos é obtido, passar à contagem detalhada. Além disso, 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.

8.5.3 - Estimativas de Esforço

Para a realização de estimativas de tempo cronológico (duração) 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 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 8.1 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, pode-se considerar um fator de produtividade, indicando quanto em termos de esforço é necessário para completar um projeto (ou módulo), descrito em termos de tamanho. Assim, uma organização pode coletar dados de vários projetos e estabelecer, por exemplo, quantos em homens-hora (uma medida de esforço) são necessários para desenvolver um ponto de função (medida 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.

Por exemplo, o modelo proposto por Bailey-Basili (PFLEEGER, 2004) 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:

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

Outros modelos são apresentados em (PRESSMAN, 2011), (PFLEEGER, 2004) e (SOMMERVILLE, 2011). 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 (PRESSMAN, 2011).

8.5.4 - Alocação de Recursos

Estimar os recursos necessários para o desenvolvimento de um produto de software é outra importante tarefa. Quando falamos em recursos, estamos englobando pessoas, hardware e software, dentre outros. No caso de software, devemos pensar em ferramentas de software, tais como ferramentas CASE ou software de infraestrutura (p.ex., um sistema gerenciador de banco de dados), 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 duração. Assim, alocação de recursos e estimativa de duração 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 interpessoal etc., importantes para a formação de equipes, são igualmente relevantes para a alocação de recursos humanos a atividades.

8.5.5 - Estimativa de Duração e Elaboração de Cronograma

De posse das estimativas de esforço e paralelamente à alocação de recursos, é possível estimar o tempo cronológico (duração) de cada atividade e, por conseguinte, do projeto como um todo. 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 8.2 (PRESSMAN, 2011).

Tabela 8.2 – 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 (PFLEEGER, 2004). 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.

8.5.6 - Estimativa de Custo

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 (2011) 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 item, 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 (SOMMERVILLE, 2011).

Estimativas de custo podem ser derivadas de estimativas de tamanho ou esforço. Por exemplo, muitas organizações que trabalham com pontos de função têm valores pré-definidos da relação R$/PF a serem utilizados na elaboração de estimativas de custo e na cotação de preços para clientes.

Uma vez definido o custo de um projeto, deve-se elaborar um orçamento ou cronograma financeiro do projeto, indicando o momento e o valor de cada desembolso e de cada entrada de recursos do projeto.

8.6 - Gerência de Riscos

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: (i) *incerteza* – um risco pode ou não acontecer, isto é, não existe nenhum risco 100% provável; (ii) *perda* – se o risco se tornar realidade, consequências 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 (PRESSMAN, 2011): (i) estabelecer uma escala que reflita a probabilidade de um risco, (ii) avaliar as consequências 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 (1999) propõe os valores quantitativos mostrados nas tabelas 8.3 e 8.4 para as escalas acima.

Tabela 8.3 - Categorias de Probabilidade (JALOTE, 1999)

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

Tabela 8.4 - Categorias de Impacto (JALOTE, 1999)

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 em um plano de riscos. Esse plano servirá de base para a atividade de monitoramento dos riscos, quando os riscos serão 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 consequências da sua ocorrência.

8.7 - Elaboração do Plano de Projeto

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.

Tipicamente, um plano de projeto é composto de outros artefatos, dentre eles: processo do projeto, estrutura analítica do projeto, estimativas, cronograma, orçamento e plano de riscos.

Referências do Capítulo

HAZAN, C., "Medição da Qualidade e Produtividade em Software", In: *Qualidade e Produtividade em Software*, 4ª edição, K.C. Weber, A.R.C. Rocha, C.J. Nascimento (organizadores), Makron Books, p. 25 – 41, 2001.

JALOTE, P., *CMM in Practice: Processes For Executing Software Projects At Infosys*, Addison-Wesley Publishing Company, 1999.

MARTINS, J.C.C., *Gerenciando Projetos de Desenvolvimento de Software com PMI, RUP e UML*, 2ª edição revisada, Rio de Janeiro: Brasport, 2005.

PFLEEGER, S.L., *Engenharia de Software: Teoria e Prática*, 2ª Edição, São Paulo: Prentice Hall, 2004.

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

SOMMERVILLE, I., *Engenharia de Software*, 9ª Edição. São Paulo: Pearson Prentice Hall, 2011.

VAZQUEZ, C.E., SIMÕES, G.S., ALBERT, R.M., *Análise de Pontos de Função: Medição, Estimativas e Gerenciamento de Projetos de Software*, 3ª edição, São Paulo: Editora Érica, 2005.

VIEIRA, M.F., *Gerenciamento de Projetos de Tecnologia da Informação*, Rio de Janeiro: Editora Elsevier – Campus, 2003.

Capítulo 9 – Tópicos Avançados em Engenharia de Software

Estudos mostram que a qualidade do produto de software depende diretamente da qualidade dos processos adotados no seu desenvolvimento (FUGGETTA, 2000). Assim, cada vez mais, as organizações têm despendido esforços significativos na melhoria contínua de seus processos de software. Este capítulo discute os seguintes aspectos relacionados à melhoria de processos de software: normas e modelos de qualidade de processo, processos de software padrão, processos ágeis e apoio automatizado ao processo de software.

9.1 – Normas e Modelos de Qualidade de Processo de Software

Os modelos de ciclo de vida se atêm apenas às atividades básicas do processo de desenvolvimento e suas inter-relações. Eles são um importante ponto de partida para a definição de processos, mas não são suficientes. Afinal, um processo de software é, na verdade, um conjunto de processos, dentre eles o processo de desenvolvimento. Mas há outros, tais como os processos de gerência de projetos e de garantia da qualidade. Para o sucesso na definição e melhoria dos processos de software, é fundamental que vários aspectos sejam considerados. Vários modelos e normas de qualidade de processo têm surgido com o objetivo de apoiar a busca por processos de maior qualidade, apontando os principais aspectos que um processo de qualidade deve considerar. Dentre essas normas e modelos destacam-se as normas ISO 9000 (ISO, 2005), ISO/IEC 12207 (ISO/IEC, 2008), ISO/IEC 15504 (ISO/IEC, 2004) e os modelos CMMI (SEI, 2010) e MPS.BR (SOFTEX, 2011).

9.1.1 – Normas ISO

As normas da família ISO 9000 (ISO, 2005) foram desenvolvidas para apoiar organizações, de todos os tipos e tamanhos, na implementação e operação de sistemas eficazes de gestão da qualidade. As normas que compõem a série ISO 9000 são:

- ISO 9000: descreve os fundamentos de sistemas de gestão da qualidade e estabelece a terminologia para esses sistemas;
- ISO 9001: especifica os requisitos para um sistema de gestão da qualidade com enfoque na satisfação do cliente. Para uma organização ser certificada ISO 9001, ela precisa demonstrar sua capacidade para fornecer produtos que atendam aos requisitos do cliente (explícitos e implícitos) e os requisitos regulamentares aplicáveis;
- ISO 9004: fornece diretrizes que consideram tanto a eficácia como a eficiência do sistema de gestão da qualidade. Seu objetivo é melhorar o desempenho da organização e a satisfação dos clientes e das demais partes interessadas;
- ISO 19011: fornece diretrizes sobre auditoria internas e externas de sistemas de gestão da qualidade.

A ISO 9001 é de caráter geral, ou seja, não se destina especificamente à indústria de software e estabelece requisitos mínimos da garantia da qualidade que devem ser atendidos pelos fornecedores de produtos ou serviços. Ela é uma norma certificadora. Essa certificação, mundialmente reconhecida, é feita por organismos certificadores, em geral, credenciados por organismos nacionais de acreditação, no caso do Brasil, o INMETRO. Assim, a conquista da certificação ISO 9001 por uma empresa significa que a mesma alcançou um padrão internacional de qualidade em seus processos (ROCHA; MALDONADO; WEBER, 2001).

O principal problema para se adotar essa norma é precisamente o fato dela ser geral. Assim, quando aplicada ao contexto da indústria de software, muitos problemas surgem pela falta de diretrizes mais focadas nas características de processos de software. Assim, de maneira geral, outras normas e modelos de qualidade são usadas por organizações de software para apoiar uma certificação ISO 9001, com destaque para a norma ISO/IEC 12207.

A norma ISO/IEC 12207 – *Systems and software engineering: Software life cycle processes* (Engenharia de Software e de Sistemas: Processos de Ciclo de Vida de Software) (ISO/IEC, 2008) estabelece uma estrutura comum para os processos de ciclo de vida de software, com terminologia bem definida, que pode ser referenciada pela indústria de software. A estrutura contém processos, atividades e tarefas que devem ser aplicados na aquisição, fornecimento, desenvolvimento, operação e manutenção de produtos de software. Esse conjunto de processos, atividades e tarefas foi projetado para ser adaptado de acordo com as características de cada projeto de software, o que pode envolver o detalhamento, a adição e a supressão de processos, atividades e tarefas não aplicáveis ao mesmo.

A ISO/IEC 12207 abrange tanto a Engenharia de Software quanto a Engenharia de Sistemas. Ela agrupa os processos do ciclo de vida em sete grupos de processo, a saber: Processos de Acordo, Processos Organizacionais Habilitadores de Projetos, Processos de Projeto, Processos Técnicos, Processos de Implementação de Software, Processos de Suporte de Software, Processos de Reutilização de Software, sendo os quatro primeiros processos de contexto de sistema e os três últimos processos específicos de software. A Figura 9.1 mostra os grupos de processo da ISO/IEC 12207 e seus processos de ciclo de vida. Dentre os processos mostrados, destacam-se os seguintes abordados anteriormente nestas notas de aula:

- Grupo de Processos de Projeto: Os processos de Planejamento de Projetos, Avaliação e Controle de Projetos e Gerência de Riscos correspondem ao Processo de Gerência de Projetos estudado no Capítulo 8. O processo de Medição está relacionado à medição, tema abordado tanto no Capítulo 7 quanto no Capítulo 8.
- Grupo de Processos de Implementação de Software: Os processos de Análise de Requisitos de Software, Projeto da Arquitetura de Software, Projeto Detalhado de Software, Construção de Software, Integração de Software e Teste de Qualificação de Software correspondem às atividades do processo de desenvolvimento de software estudadas na Parte I destas notas de aula.
- Grupo de Processos de Suporte de Software: Os processos de Gerência de Documentação de Software, Gerência de Configuração de Software, Garantia da Qualidade, Verificação, Validação e Revisão de Software foram abordados no Capítulo 7.

Além de atividades e tarefas, cada processo tem um propósito e um resultado associados. Os propósitos e resultados dos processos de ciclo de vida constituem um Modelo de Referência de Processos.

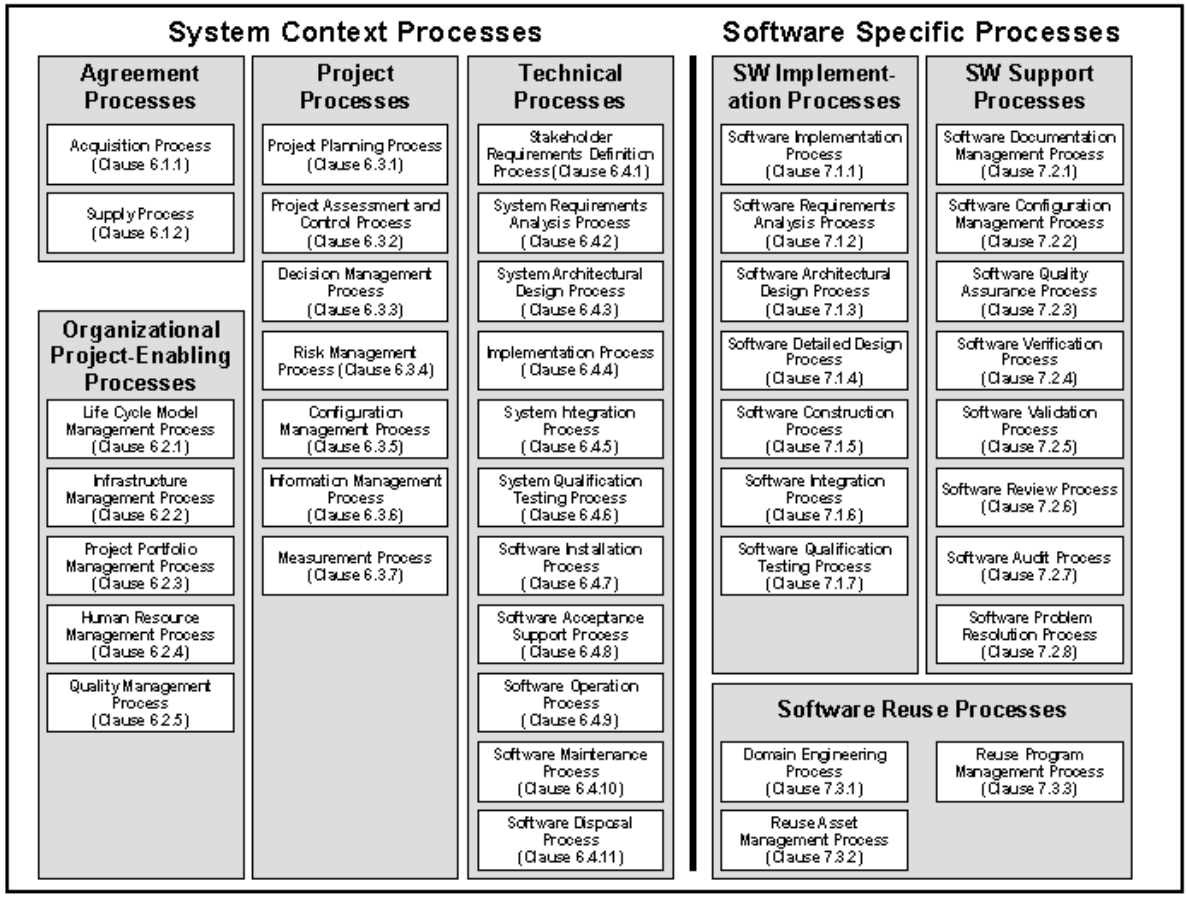


Figura 9.1 – Processos de Ciclo de Vida da ISO/IEC 12207.

Por fim, a norma ISO/IEC 15504 – *Information Technology – Process Assessment* (Tecnologia da Informação – Avaliação de Processos) (ISO/IEC, 2004) é um padrão para avaliação de processos de software. Juntas as normas ISO/IEC 12207 e ISO/IEC 15504 estabelecem um *framework* com terminologias bem definidas e boas práticas para a definição e avaliação de processos de software.

9.1.2 - O Modelo CMMI

O Modelo de Maturidade e Capacidade Integrado (*Capability Maturity Model Integration - CMMI*) (SEI, 2010) foi desenvolvido no Instituto de Engenharia de Software (*Software Engineering Institute - SEI*) da Universidade de Carnegie Mellon, com o intuito de quantificar a capacidade de uma organização produzir produtos de software de alta qualidade, de forma previsível e consistente. Sua motivação inicial foi apontar as necessidades de melhoria nos projetos de desenvolvimento de software do Departamento de Defesa dos EUA.

O CMMI é estruturado em cinco níveis de maturidade, de 1 a 5, onde o nível 1 é o menos maduro e o nível 5 é o mais maduro. Cada nível de maturidade, com exceção do nível

1, é composto de várias áreas de processo (*process areas* - PAs). As características de cada nível são apresentadas a seguir.

- Nível 1 – Inicial: O processo de software é caracterizado como *ad hoc* e, eventualmente, caótico. Poucos processos são definidos e o sucesso depende de esforços individuais. Neste nível, a organização, tipicamente, opera sem formalizar procedimentos, sem realizar estimativas de custo e sem ter planos de projeto. Ferramentas não são bem integradas ao processo ou não são uniformemente aplicadas. O controle de alterações é superficial e há pouco entendimento sobre os problemas.
- Nível 2 – Gerenciado: Os processos básicos de gerência são estabelecidos para acompanhar custo, cronograma e funcionalidade. Os sucessos em projetos anteriores com aplicações similares podem ser repetidos.
- Nível 3 – Definido: A organização possui um processo padrão definido que é usado como base para todos os projetos. As atividades de engenharia e gerência de software são estáveis e há um entendimento comum e amplo das atividades, papéis e responsabilidades no processo.
- Nível 4 – Gerenciado Quantitativamente: A organização fixa metas quantitativas de qualidade para produtos e processos e fornece instrumentos para medições consistentes e bem definidas. Tanto o processo de software como os produtos são quantitativamente entendidos e controlados. É possível que a organização preveja tendências na qualidade do processo e dos produtos, dentro de fronteiras quantificadas.
- Nível 5 – Em Otimização: A organização possui uma base para melhoria contínua e otimização do processo. Dados sobre a eficiência de um processo são usados para efetuar análises custo-benefício de novas tecnologias e para propor mudanças no processo.

A Figura 9.2 mostra os níveis de maturidade do CMMI e, em seguida, na Tabela 9.1 são apresentadas as áreas de processo de cada nível.

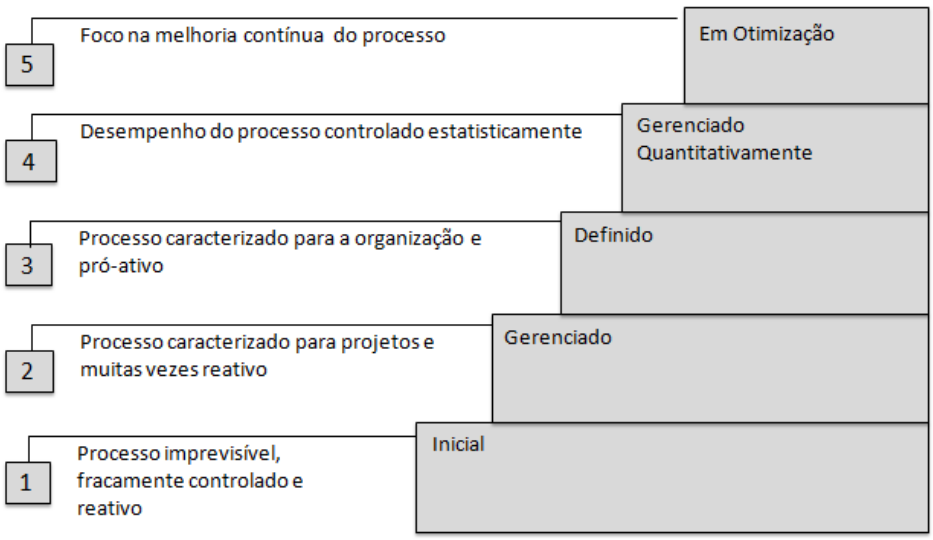


Figura 9.2 - Os níveis de maturidade do CMMI.

Tabela 9.1 – As áreas de processo dos níveis de maturidade do CMMI.

Nível	Área de Processo
2	Planejamento de Projeto
	Monitoração e Controle de Projeto
	Gerência de Requisitos
	Gerência de Acordo com Fornecedor
	Gerência de Configuração
	Medição e Análise
	Garantia da Qualidade de Produto e de Processo
3	Definição do Processo Organizacional
	Foco no Processo Organizacional
	Gerência de Projetos Integrada
	Treinamento Organizacional
	Desenvolvimento de Requisitos
	Solução Técnica
	Integração do Produto
	Verificação
	Validação
	Gerência de Riscos
	Análise e Tomada de Decisões
4	Desempenho do Processo Organizacional
	Gerência de Projetos Quantitativa
5	Análise e Resolução de Causas
	Gerência do Desempenho Organizacional

9.1.3 - O Modelo de Referência Brasileiro – MPS.BR

O MPS.BR – Melhoria de Processo do Software Brasileiro (SOFTEX, 2011) tem como objetivo definir um modelo de melhoria e avaliação de processo de software, adequado, preferencialmente, às micro, pequenas e médias empresas brasileiras, de forma a atender as suas necessidades de negócio e a ser reconhecido nacional e internacionalmente como um modelo aplicável à indústria de software. Por este motivo, está aderente a modelos e normas internacionais.

A base técnica utilizada para a construção do MPS.BR é composta pelas normas ISO/IEC 12207 e ISO/IEC 15504, estando totalmente aderente a essas normas. Além disso, o MPS.BR também cobre o conteúdo do CMMI.

O MPS.BR está dividido em três componentes:

- Modelo de Referência (MR-MPS): contém os requisitos que as organizações deverão atender para estar em conformidade com o MPS.BR. Define, também, os níveis de maturidade e de capacidade de processos e os processos em si.
- Método de Avaliação (MA-MPS): contém o processo de avaliação, os requisitos para os avaliadores e os requisitos para averiguação da conformidade ao modelo MR-MPS. Está descrito de forma detalhada no Guia de Avaliação e foi baseado na norma ISO/IEC 15504.
- Modelo de Negócio (MN-MPS): contém uma descrição das regras para a implementação do MR-MPS pelas empresas de consultoria, de software e de avaliação.

O MR-MPS define sete níveis de maturidade: A (Em Otimização), B (Gerenciado Quantitativamente), C (Definido), D (Largamente Definido), E (Parcialmente Definido), F (Gerenciado) e G (Parcialmente Gerenciado). A escala de maturidade se inicia no nível G e progride até o nível A. Para cada um desses sete níveis de maturidade, foi atribuído um perfil de processos e de capacidade de processos que indicam onde a organização tem que colocar esforços para melhoria de forma a atender os objetivos de negócio. A Tabela 9.2 mostra os níveis de maturidade do MPS.BR e seus processos. A Tabela 9.3 mostra um comparativo dos níveis e processos do MR-MPS-SW, níveis G a D, com seus correspondentes no modelo CMMI-DEV versão 1.3, níveis 2 e 3.

Tabela 9.2 – Níveis de Maturidade e Processos do MPS.BR.

Nível	Processos
A	
B	Gerência de Projetos (evolução)
C	Gerência de Decisões (GDE) Desenvolvimento para Reutilização (DRU) Gerência de Riscos (GRI)
D	Desenvolvimento de Requisitos (DRE) Projeto e Construção do Produto (PCP) Integração do Produto (ITP) Verificação (VER) Validação (VAL)
E	Avaliação e Melhoria de Processo Organizacional (AMP) Definição do Processo Organizacional (DFP) Gerência de Recursos Humanos (GRH) Gerência de Reutilização (GRU) Gerência de Projetos (evolução)
F	Medição (MED) Gerência de Configuração (GCO) Aquisição (AQU) Garantia da Qualidade (GQA) Gerência de Portfólio de Projetos (GPP)
G	Gerência de Requisitos (GRE) Gerência de Projetos (GPR)

Tabela 9.3 – Processos do MR-MPS-SW e Áreas de Processos do CMMI-DEV

MR-MPS-SW		CMMI-DEV	
Nível	**Processo**	**Nível**	**Área de Processo**
G	Gerência de Projetos	**2**	Planejamento de Projeto
			Monitoração e Controle de Projeto
	Gerência de Requisitos		Gerência de Requisitos
F	Aquisição		Gerência de Acordo com Fornecedor
	Gerência de Configuração		Gerência de Configuração
	Garantia da Qualidade		Garantia da Qualidade de Produto e de Processo
	Gerência de Portfólio de Projetos		-
	Medição		Medição e Análise
E	Avaliação e Melhoria do Processo Organizacional	**3**	Foco no Processo Organizacional
	Definição do Processo Organizacional		Definição do Processo Organizacional
	Gerência de Projetos (evolução)		Gerência de Projetos Integrada
	Gerência de Recursos Humanos		Treinamento Organizacional
	Gerência de Reutilização		-
D	Desenvolvimento de Requisitos		Desenvolvimento de Requisitos
	Integração do Produto		Integração do Produto
	Projeto e Construção do Produto		Solução Técnica
	Verificação		Verificação
	Validação		Validação
C	Desenvolvimento para Reutilização		-
	Gerência de Decisões		Análise e Tomada de Decisões
	Gerência de Riscos		Gerência de Riscos

9.2 – Processos Padrão

Vários dos modelos e normas de qualidade de processo discutidos anteriormente (p.ex., processo Definição do Processo Organizacional do MR-MPS) preconizam que, embora diferentes projetos requeiram processos com características específicas para atender às suas particularidades, é possível estabelecer um conjunto de ativos de processo (subprocessos, atividades, subatividades, 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 de software padrão. 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 tipos de software, 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 padrão 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. A Figura 9.3 ilustra essa abordagem de definição de processos de software em níveis (ROCHA; MALDONADO; WEBER, 2001).

Uma vez que objetivo das 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, elas são uma importante base para a definição dos processos padrão das organizações. 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.

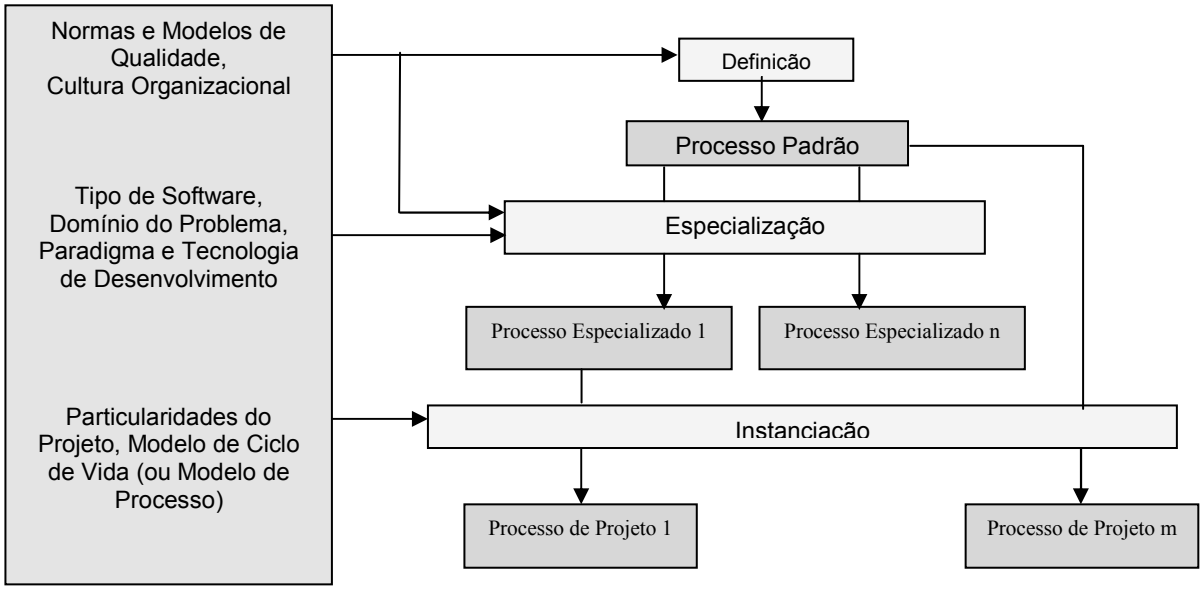


Figura 9.3 – Modelo para Definição de Processos em Níveis.

Sob o ponto de vista do conhecimento do processo, é essencial que os processos padrão (da organização ou especializados) estejam internalizados nas pessoas, ou seja, os desenvolvedores devem executá-los naturalmente. Além disso, o processo padrão organizacional deve estar institucionalizado, isto é, toda a organização deve executá-lo.

9.3 – Processos Ágeis

Um dos principais desafios no desenvolvimento de software é lidar com mudanças. Diversos desenvolvedores apontam que uma abordagem tradicional de desenvolvimento, centrada no planejamento e na execução de processos, é inapropriada para o desenvolvimento de sistemas muito sujeitos a mudanças. Uma alternativa para esses casos seria a agilidade.

De maneira bem geral, agilidade pode ser vista como a capacidade de adaptação das organizações face às mudanças ocorridas no ambiente de negócios (SANTANA JÚNIOR, 2012). Uma equipe ágil é aquela que consegue responder rapidamente a mudanças, dá valor às características e habilidades de cada membro e reconhece que a colaboração é a chave para o sucesso do projeto (PRESSMAN, 2011).

As bases para o desenvolvimento ágil foram definidas por Kent Beck e outros 16 desenvolvedores por meio do Manifesto para Desenvolvimento Ágil de Software (PRESSMAN, 2011). Esse manifesto diz que devem ser valorizados: (i) indivíduos e interações ao invés de processos e ferramentas; (ii) software funcional ao invés de documentação; (iii) colaboração ao invés de negociação; (iv) responder às mudanças ao invés de seguir um plano. São 12 os princípios ágeis:

1) Satisfazer o cliente é a prioridade máxima, por meio de entrega contínua de software de valor.
2) Mudanças em requisitos devem ser bem recebidas, mesmo em fases mais avançadas do desenvolvimento. Os processos ágeis direcionam as mudanças para obter vantagens competitivas para o cliente.
3) Entregar software funcional com frequência (de semanas a meses), de preferência no menor espaço de tempo.
4) Interessados (*stakeholders*) e desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.
5) Desenvolver projetos em torno de indivíduos motivados. Dê a eles o ambiente e o apoio que precisam e confie que eles farão o trabalho.
6) O método mais eficiente e efetivo para troca de informação internamente é a conversa cara a cara.
7) Software funcionando é a principal medida de progresso.
8) Desenvolvimento sustentável: manter um ritmo constante do desenvolvimento, indefinidamente.
9) Atenção contínua à excelência técnica e ao bom projeto promovem agilidade.
10) Simplicidade (a arte de maximizar a quantidade de trabalho não efetuado) é essencial.
11) As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
12) Em intervalos regulares, a equipe deve refletir sobre como se tornar mais efetiva e ajustar seu comportamento.

As premissas para o uso de processos ágeis são (PRESSMAN, 2011): (i) É difícil predizer quais requisitos serão mantidos e quais vão mudar, assim como quais prioridades mudarão ou não; (ii) Análise, projeto, construção e teste não são tão previsíveis (do ponto de vista de planejamento) como gostaríamos; (iii) Uma vez que é difícil prever o quanto de

projeto (*design*) é necessário antes de podem começar a implementação, projeto e implementação devem ser feitos em paralelo.

São características necessárias das pessoas e da equipe ágil em si:

- Competência: habilidades específicas de desenvolvimento de software e conhecimento do processo a ser usado.
- Foco comum: todos devem ter o mesmo foco: entregar ao cliente a próxima versão (incremento) do sistema no prazo prometido.
- Colaboração: deve haver colaboração entre membros da equipe e com o cliente.
- Autonomia para tomar decisões técnicas e de planejamento.
- Versatilidade: aceitar o fato de que o problema sendo resolvido hoje pode mudar amanhã.
- Respeito e confiança mútuas: a equipe é mais do que a soma das partes.
- Auto-organização: o time se auto-gerencia em relação ao trabalho a ser feito, às adaptações ao processo e ao cronograma de trabalho.

A motivação para os processos ágeis é tentar combater as fraquezas dos processos convencionais de Engenharia de Software. Contudo, processos ágeis não são uma antítese dos processos tradicionais. De fato, princípios de agilidade podem ser aplicados a quaisquer processos de software. Dentre os métodos ágeis de desenvolvimento de software, destacam-se o eXtreme Programming (XP) e o Scrum.

9.3.1 – eXtreme Programming - XP

Ainda que as ideias subjacentes ao XP existissem desde o final da década de 1980, sua concepção foi formalizada em 1999, com o livro "*Extreme Programming Explained: Embrace Change*", de Kent Beck. O processo de software preconizado pelo XP possui quatro atividades principais (PRESSMAN, 2011): planejamento, design, implementação e testes. Estas atividades são organizadas de maneira incremental, em diversos ciclos, sendo que a cada ciclo, uma versão operacional (ou incremento) é produzida e entregue ao cliente.

No planejamento, histórias de usuário (*user stories*) são escritas e priorizadas (atribuição de valor) pelo cliente, e membros da equipe estimam uma duração em semanas de desenvolvimento para cada uma delas. Se a duração de uma história for maior do que 3 semanas, é pedido ao cliente que divida a história. Clientes e desenvolvedores decidem juntos quais histórias entrarão na próxima versão do sistema. No lançamento da primeira versão, a velocidade do projeto (quantidade de histórias implementadas) é calculada. A velocidade do projeto é usada para estimar datas de entrega futuras. À medida que o trabalho progride, o cliente pode adicionar, alterar, excluir ou dividir histórias. O time XP aceita as mudanças e se replaneja.

Na fase de projeto (design), parte-se do princípio que um modelo simples é sempre preferível a um modelo complexo e que não se deve projetar funcionalidade extra assumindo que ela será necessária no futuro. XP recomenda o uso de refatoração (*refactoring*), que consiste na reorganização interna do código-fonte sem alteração no seu comportamento externo. A refatoração permite melhorias no projeto depois que a implementação já iniciou, o

que é importante em XP, uma vez que projeto e implementação ocorrem em paralelo. Contudo, é importante ressaltar que o esforço necessário para refatoração cresce à medida que o tamanho da aplicação aumenta.

A implementação começa com a criação de testes de unidade para cada história incluída no incremento. A ideia subjacente a esta abordagem é que desenvolver com o teste pronto é mais fácil. É como estudar para uma prova já sabendo as questões. Nada extra deve ser desenvolvido. Só o que está incluído no teste. Assim que o código é produzido, os testes podem ser efetuados em seguida (*feedback* instantâneo). Outro conceito chave em XP é a *programação em pares*: duas pessoas trabalham juntas (na mesma máquina) durante a implementação. Com isso, tem-se um mecanismo de resolução de problemas em tempo real (duas cabeças pensam melhor do que uma) e garantia da qualidade. A intenção é manter os programadores focados na tarefa, sendo que cada programador assume um papel ligeiramente diferente. A integração é feita pelos programadores (pares) ou por uma equipe de integração diariamente.

Conforme anteriormente citado, os testes de unidade são construídos durante a implementação. Para tal, XP preconiza que deve ser usado um *framework* de automatização de testes, de modo que uma estratégia de testes de regressão possa ser adotada. A premissa de XP é que consertar problemas pequenos a cada duas ou três horas gasta menos tempo do que consertar problemas enormes perto da data de entrega. Testes de aceitação são especificados pelo cliente com base nas histórias de usuário.

9.3.2 – Scrum

Scrum tem como premissa a existência do caos. O nome deste método vem de uma atividade que ocorre em partidas de Rugby. Os principais princípios de Scrum são (PRESSMAN, 2011):

- Equipes pequenas são organizadas para maximizar a comunicação, minimizar o *overhead* e compartilhar conhecimento tácito e informal.
- O processo deve ser adaptável a mudanças técnicas e de negócio.
- Há incrementos frequentes e regulares de software, que podem ser inspecionados, ajustados, testados, documentados e expandidos.
- O trabalho e os membros da equipe são divididos em partições de baixo acoplamento.
- Documentação e testes constantes são feitos à medida que o produto é construído.
- O processo tem a capacidade de declarar o produto como pronto "a qualquer momento, por qualquer motivo" (a concorrência saiu na frente, o usuário precisa do sistema, acabou o prazo etc).

O processo de Scrum envolve as seguintes atividades: levantamento de requisitos, análise, projeto, evolução e entrega. As tarefas de cada atividade são feitas dentro de um padrão de processo chamado "*sprint"*.

Uma lista priorizada de requisitos, dita *backlog*, é mantida. Requisitos podem ser adicionados, removidos e alterados a qualquer momento, bem como as prioridades podem ser alteradas. Um *sprint* é uma unidade de trabalho com tempo pré-determinado (tipicamente de 30 dias). Durante um *sprint*, são escolhidos alguns requisitos do *backlog* e estes são

considerados "congelados", i.e., não sujeitos a mudanças. Assim, a equipe pode trabalhar em um ambiente estável por um curto período de tempo.

Pequenas reuniões, de aproximadamente 15 minutos, são feitas diariamente pela equipe com o *Scrum Master* (uma espécie de líder do projeto), na qual três perguntas chave são feitas para cada membro da equipe: O que você fez desde a última reunião? Que obstáculos você tem encontrado? O que você planeja fazer até a próxima reunião? Os objetivos dessas reuniões scrum são descobrir potenciais problemas cedo, socializar o conhecimento e promover a auto-organização da equipe.

O produto resultante de um *sprint* é um demo. Demos são incrementos de software entregues ao cliente como demonstrações. O cliente avalia as demonstrações para dar *feedback*. Cada demo contém as funções até o último *sprint*.

9.4 – Apoio Automatizado ao Processo de Software

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, 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. Por exemplo, para apoiar atividades da gerência de projetos há diversas ferramentas disponíveis, tais como Microsoft Project ou dotProject, sendo esta última uma ferramenta livre, de código aberto.

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 apoiar 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 do Capítulo

FUGGETTA, A., Software Process: A Roadmap, In: Proceedings of The Future of Software Engineering, ICSE'2000, Limerick, Ireland, 2000.

ISO, *ISO 9000 – Quality management systems: Fundamentals and vocabulary*, 2005.

ISO/IEC, *ISO/IEC 12207 – Systems and software engineering: Software life cycle processes*, 2008.

ISO/IEC, *ISO/IEC 15504 Information technology – Process assessment – Part 1: Concepts and vocabulary*, 2004.

PRESSMAN, R.S., *Engenharia de Software: Uma Abordagem Profissional*, 7ª Edição, McGraw-Hill, 2011.

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C., *Qualidade de Software: Teoria e Prática*, São Paulo: Prentice Hall, 2001.

SEI, *CMMI for Development, Version 1.3*, Technical Report ESC-TR-2010-33, 2010.

SOFTEX, *MPS.BR – Melhoria de Processo do Software Brasileiro – Guia Geral*, 2011.

Anexo A – Análise de Pontos de Função

A Análise de Pontos de Função (APF) é um método padrão para a medição do desenvolvimento de software, visando estabelecer uma medida de tamanho do software em Pontos de Função (PFs), com base na funcionalidade a ser implementada, sob o ponto de vista do usuário.

Os objetivos da APF são:

- Medir as funcionalidades do sistema requisitadas e recebidas pelo usuário;
- Medir projetos de desenvolvimento e manutenção de software, sem se preocupar com a tecnologia que será utilizada na implementação.

O processo para contagem de PFs compreende sete passos, mostrados na figura A.1.

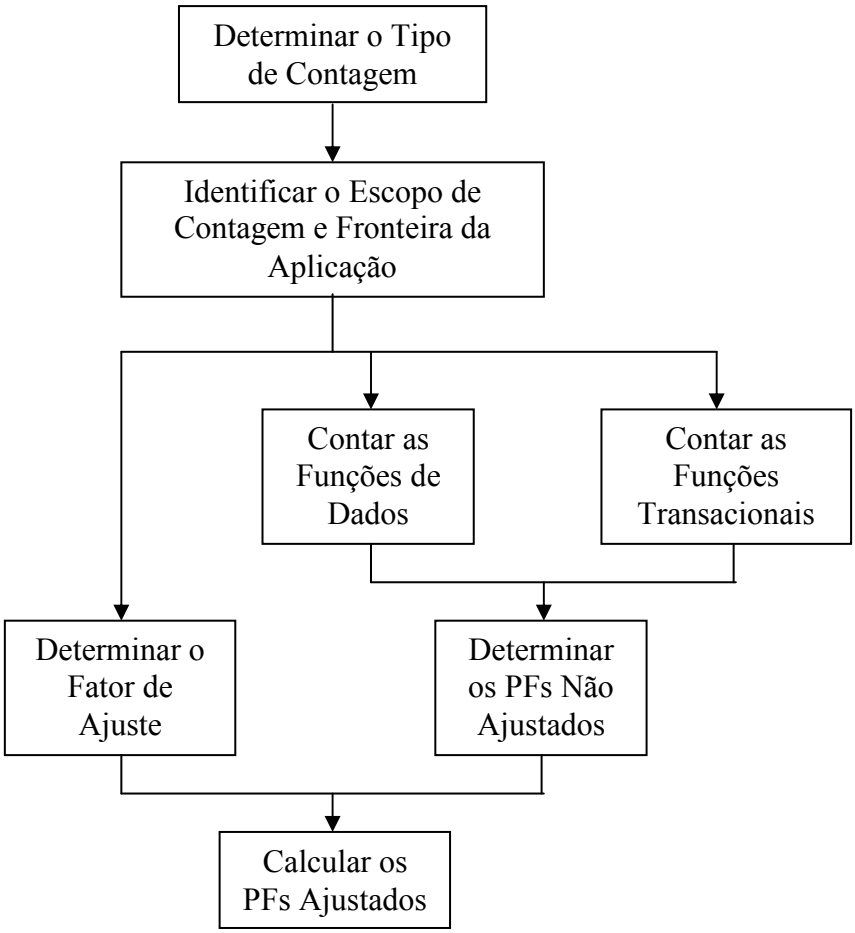


Figura A.1 – O Processo de Contagem de Pontos de Função.

- **Determinar o tipo de contagem de pontos de função**: este é o primeiro passo no processo de contagem, sendo que existem três tipos de contagem: contagem de PF de projeto de desenvolvimento, de aplicações instaladas e de projetos de manutenção.
- **Identificar o escopo de contagem e a fronteira da aplicação**: neste passo, definem-se as funcionalidades que serão incluídas em uma contagem de PFs específica. A fronteira da aplicação é definida estabelecendo um limite lógico entre a aplicação que está sendo medida, o usuário e outras aplicações. O escopo de contagem define a parte do sistema (funcionalidades) a ser contada.
- **Determinar a contagem de pontos de função não ajustados**: os pontos de função não ajustados (PFNA) refletem as funcionalidades fornecidas pelo sistema para o usuário. Essa contagem leva em conta dois tipos de função: de dados e transacionais, bem como sua complexidade (simples, média ou complexa).
- **Contagem das funções de dados**: as funções de dados representam as funcionalidades relativas aos requisitos de dados internos e externos à aplicação. São elas os arquivos lógicos internos e os arquivos de interface externa. Ambos são grupos de dados logicamente relacionados ou informações de controle que foram identificados pelo usuário. A diferença está no fato de um **Arquivo Lógico Interno (ALI)** ser mantido dentro da fronteira da aplicação, isto é, armazenar os dados mantidos através de um ou mais processos elementares da aplicação, enquanto que um **Arquivo de Interface Externa (AIE)** é apenas referenciado pela aplicação, ou seja, ele é mantido dentro da fronteira de outra aplicação. Assim, o objetivo de um AIE é armazenar os dados referenciados por um ou mais processos elementares da aplicação sendo contada, mas que são mantidos por outras aplicações.
- **Contagem das funções transacionais**: as funções transacionais representam as funcionalidades de processamento de dados do sistema fornecidas para o usuário. São elas: as entradas externas, as saídas externas e as consultas externas. As **Entradas Externas (EEs)** são processos elementares que processam dados (ou informações de controle) que entram pela fronteira da aplicação. O objetivo principal de uma EE é manter um ou mais ALIs ou alterar o comportamento do sistema. As **Saídas Externas (SEs)** são processos elementares que enviam dados (ou informações de controle) para fora da fronteira da aplicação. Seu objetivo é mostrar informações recuperadas através de um processamento lógico (isto é, que envolva cálculos ou criação de dados derivados) e não apenas uma simples recuperação de dados. Uma SE pode, também, manter um ALI ou alterar o comportamento do sistema. Por fim, uma **Consulta Externa (CE)**, assim como uma SE, é um processo elementar que envia dados (ou informações de controle) para fora da fronteira da aplicação, mas sem realização de nenhum cálculo nem a criação de dados derivados. Seu objetivo é apresentar informação para o usuário, por meio apenas de uma recuperação das informações. Nenhum ALI é mantido durante sua realização, nem o comportamento do sistema é alterado.
- **Determinar o valor do fator de ajuste**: o fator de ajuste é baseado em 14 características gerais de sistemas, que avaliam a funcionalidade geral da aplicação que está sendo contada, e seus níveis de influência. O nível de influência de uma característica é determinado com base em uma escala de 0 (nenhuma influência) a 5 (forte influência). Assim, o fator de ajuste visa a ajustar os pontos de função não ajustados em ± 35%. Esse passo tornou-se opcional em 2002 para que o método da APF passasse a ser um padrão internacional de medição funcional (ISO/IEC 20926).

As principais críticas são a grande variação na interpretação das 14 características gerais de sistemas e a constatação que algumas delas estão desatualizadas.

- **Calcular os pontos de função ajustados**: finalmente, os PFs ajustados são calculados, considerando-se o tipo de contagem definido no primeiro passo.

A figura A.2 apresenta uma visão geral dos tipos de função que são considerados na contagem da APF.

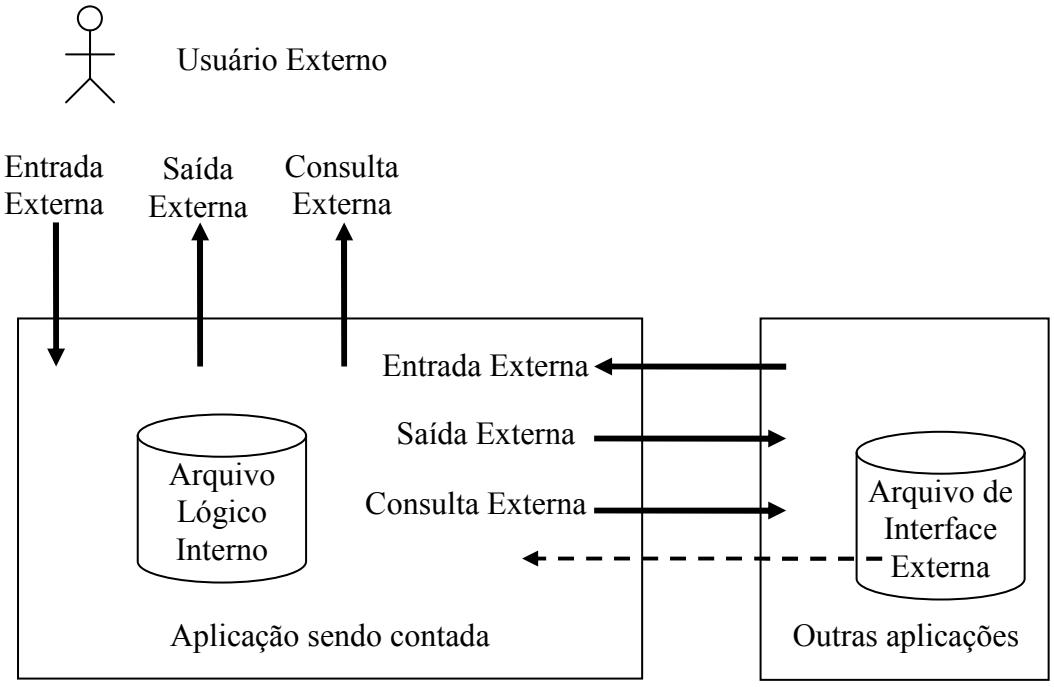


Figura A.2 – Visão Geral das Funções de uma Aplicação segundo a APF.

Contagem das Funções de Dados

Conforme discutido anteriormente, o primeiro passo para a contagem das funções de dados consiste em identificar arquivos lógicos internos (ALIs) e arquivos de interface externa (AIEs). Cada uma dessas funções de dados deve ser classificada segundo sua complexidade funcional. Essa complexidade é definida com base em dois conceitos: registros lógicos e itens de dados.

Registros Lógicos são subconjuntos de dados dentro de um ALI/AIE, que foram reconhecidos pelo usuário. Se o usuário não reconhecer subconjuntos de dados em um ALI/AIE, então se deve contar o ALI/AIE como um registro lógico.

Um Item de Dados, por sua vez, é um campo reconhecido pelo usuário como único e não repetido. Vale destacar que só devem ser contados os itens de dados utilizados pela aplicação em contagem.

Contando-se os registros lógicos e os itens de dados de um ALI/AIE, pode-se chegar à sua complexidade, utilizando a tabela A.1.

Tabela A.1 – Tabela de Identificação da Complexidade das Funções de Dados .

Número de Registros Lógicos	**Número de Itens de Dados Referenciados**		
	De 1 a 19	**De 20 a 50**	**51 ou mais**
Apenas 1	Simples	Simples	Média
De 2 a 5	Simples	Média	Complexa
6 ou mais	Média	Complexa	Complexa

Contagem das Funções Transacionais

De maneira análoga à contagem das funções de dados, a contagem das funções transacionais envolve a identificação de funções transacionais (entradas externas, saídas externas e consultas externas) e sua classificação de acordo com a complexidade funcional envolvida (simples, média ou complexa). A definição da complexidade funcional é feita com base no número de arquivos referenciados e dos itens de dados manipulados pela função, utilizando as tabelas A.2 para entradas externas e A.3 para saídas e consultas externas. Nessas tabelas, um arquivo referenciado pode ser um ALI lido ou mantido pela função transacional, ou um AIE lido pela função transacional. Já o número de itens de dados referenciados é calculado considerando apenas os itens de dados efetivamente referenciados pela função transacional em questão.

Tabela A.2 – Tabela de Identificação da Complexidade de Entradas Externas.

Número de Arquivos Referenciados	**Número de Itens de Dados Referenciados**		
	De 1 a 4	**De 5 a 15**	**16 ou mais**
0 ou 1	Simples	Simples	Média
2	Simples	Média	Complexa
3 ou mais	Média	Complexa	Complexa

Tabela A.3 – Tabela de Identificação da Complexidade de Saídas e Consultas Externas.

Número de Arquivos Referenciados	**Número de Itens de Dados Referenciados**		
	De 1 a 5	**De 6 a 19**	**20 ou mais**
0 ou 1	Simples	Simples	Média
2 ou 3	Simples	Média	Complexa
4 ou mais	Média	Complexa	Complexa

Cálculo dos Pontos de Função Não Ajustados

Uma vez contadas as funções de dados e as funções transacionais, é possível calcular os PFs não ajustados de uma aplicação. Esse cálculo é feito da seguinte forma:

1. Para cada um dos cinco tipos de função (ALI, AIE, EE, SE e CE), são computados os totais de pontos de função (NPF_i), segundo a seguinte expressão:

$$NPF_i = \sum_{j=1}^{3} NC_{i,j} * C_{i,j}$$

onde

- $NC_{i,j}$ = número funções do tipo *i* (*i* variando de 1 a 5, segundo os tipos de função existentes: ALI, AIE, EE, SE e CE) que foram classificados na complexidade *j* (*j* variando de 1 a 3, segundo os valores de complexidade: simples, média e complexa).
- $C_{i,j}$ = valor da contribuição da complexidade *j* no cálculo dos pontos da função *i*, dado pela tabela A.4.

Tabela A.4 – Contribuição das Funções na Contagem de PFs Não Ajustados.

Função	Complexidade		
	Simples	**Média**	**Complexa**
ALI	7	10	15
AIE	5	7	10
EE	3	4	6
SE	4	5	7
CE	3	4	6

2. O total de pontos de função não ajustados (PFNA) é dado pelo somatório dos pontos das tabelas de função:

$$PFNA = \sum_{i=1}^{5} NPF_i$$

sendo que *i* varia de 1 a 5, segundo os tipos de função existentes (AIL, AIE, EE, SE e CE).

Uma boa opção consiste em preencher a tabela mostrada na Figura A.3.

Função	*Itens Contados por Complexidade*	*Contribuição*	*Total por Complexidade*	*Total de PFNA da Função*
ALI	__ Baixa __ Média __ Alta	x 7 x 10 x 15		
AIE	__ Baixa __ Média __ Alta	x 5 x 7 x 10		
EE	__ Baixa __ Média __ Alta	x 3 x 4 x 6		
CE	__ Baixa __ Média __ Alta	x 3 x 4 x 6		
SE	__ Baixa __ Média __ Alta	x 4 x 5 x 7		

Total de Pontos de Função Não Ajustados: ____

Figura A.3 – Tabela para Cálculo de Pontos de Função Não Ajustados.

Determinação do Fator de Ajuste

O fator de ajuste influencia os pontos de função não ajustados em ± 35%, obtendo-se o número de PFs ajustados. Para se calcular o fator de ajuste, são usadas 14 características gerais dos sistemas, a saber:

1. Comunicação de Dados
2. Processamento de Dados Distribuído
3. Desempenho
4. Utilização do Equipamento (Restrições de Recursos Computacionais)
5. Volume de Transações
6. Entrada de Dados On-line
7. Eficiência do Usuário Final (Usabilidade)
8. Atualização On-line
9. Processamento Complexo
10. Reusabilidade
11. Facilidade de Implantação
12. Facilidade Operacional (Processos Operacionais, tais como Inicialização, Cópia de Segurança, Recuperação etc)
13. Múltiplos Locais e Organizações do Usuário
14. Facilidade de Mudanças (Manutenibilidade)

Para cada uma dessas 14 características deve-se atribuir um valor de 0 (nenhuma influência) a 5 (forte influência), dito grau ou nível de influência, que indica o quanto determinada característica tem influência no sistema. Os 14 graus de influência (GIs) informados são somados, resultando no nível de influência total (NIT):

$$NIT = \sum_{i=1}^{14} GI_i$$

Finalmente, o valor do fator de ajuste (VFA) é determinado, então, pela fórmula:

$$VFA = (NIT * 0,01) + 0,65$$

Cálculo dos Pontos de Função Ajustados

Uma vez calculados os PF não ajustados e o fator de ajuste, é possível calcular os PFs ajustados. Esse cálculo é feito de formas diferentes para cada tipo de contagem (projeto de desenvolvimento, projeto de manutenção ou aplicações instaladas). Para projetos de desenvolvimento, o cálculo é dado por:

$$PF = PFNA * VFA$$

onde PFNA = Número de PFs não ajustados e
VFA = valor do fator de ajuste

Referência

C. Hazan. "Medição da Qualidade e Produtividade em Software", In: Qualidade e Produtividade em Software, 4ª edição, K.C. Weber, A.R.C. Rocha, C.J. Nascimento (organizadores), Makron Books, 2001, p. 25 – 41.

As 14 Características Gerais e seus Graus de Influência (Dias, 2004)

Grau	Descrição
0	Nenhuma influência
1	Influência mínima
2	Influência moderada
3	Influência média
4	Influência significante
5	Influência forte

1. **Comunicação de dados**: os aspectos relacionados aos recursos utilizados para a comunicação de dados do sistema deverão ser descritos de forma global. Descrever se a aplicação utiliza protocolos[2] diferentes para recebimento/envio das informações do sistema.
    0. Aplicação *batch* ou funciona *stand-alone*;
    1. Aplicação *batch*, mas utiliza entrada de dados ou impressão remota;
    2. Aplicação *batch*, mas utiliza entrada de dados e impressão remota;
    3. Aplicação com entrada de dados *on-line* para alimentar processamento *batch* ou sistema de consulta;
    4. Aplicação com entrada de dados *on-line*, mas suporta apenas um tipo de protocolo de comunicação;
    5. Aplicação com entrada de dados *on-line* e suporta mais de um tipo de protocolo de comunicação.

2. **Processamento de Dados Distribuído**: Esta característica refere-se a sistemas que utilizam dados ou processamento distribuído, valendo-se de diversas CPUs.
    0. Aplicação não auxilia na transferência de dados ou funções entre os processadores da empresa;
    1. Aplicação prepara dados para o usuário final utilizar em outro processador (do usuário final), tal como planilhas;
    2. Aplicação prepara dados para transferência, transfere-os para serem processados em outro equipamento da empresa (não pelo usuário final);
    3. Processamento é distribuído e a transferência de dados é *on-line* e apenas em uma direção;
    4. Processamento é distribuído e a transferência de dados é *on-line* e em ambas as direções;
    5. As funções de processamento são dinamicamente executadas no equipamento (CPU) mais apropriada;

[2] Protocolo é um conjunto de informações que reconhecem e traduzem para um determinado padrão, informações entre dois sistemas ou periféricos, permitindo intercâmbio das informações.

3. **Desempenho**: Trata-se de parâmetros estabelecidos pelo usuário como aceitáveis, relativos a tempo de resposta.

0. Nenhum requisito especial de desempenho foi solicitado pelo usuário;
1. Requisitos de desempenho foram estabelecidos e revistos, mas nenhuma ação especial foi requerida;
2. Tempo de resposta e volume de processamento são itens críticos durante horários de pico de processamento. Nenhuma determinação especial para a utilização do processador foi estabelecida. A data limite para a disponibilidade de processamento é sempre o próximo dia útil;
3. Tempo de resposta e volume de processamento são itens críticos durante todo o horário comercial. Nenhuma determinação especial para a utilização do processador foi estabelecida. A data-limite necessária para a comunicação com outros sistemas é limitante.
4. Os requisitos de desempenho estabelecidos requerem tarefas de análise de desempenho na fase de planejamento e análise da aplicação.
5. Além do descrito no item anterior, ferramentas de análise de desempenho foram usadas nas fases de planejamento, desenvolvimento e/ou implementação para atingir os requisitos de desempenho estabelecidos pelos usuários.

4. **Utilização do Equipamento**: Trata-se de observações quanto ao nível de utilização de equipamentos requerido para a execução do sistema. Este aspecto é observado com vista a planejamento de capacidades e custos.

0. Nenhuma restrição operacional explícita ou mesmo implícita foi incluída.
1. Existem restrições operacionais leves. Não é necessário esforço especial para atender às restrições.
2. Algumas considerações de ajuste de desempenho e segurança são necessárias.
3. São necessárias especificações especiais de processador para um módulo específico da aplicação.
4. Restrições operacionais requerem cuidados especiais no processador central ou no processador dedicado para executar a aplicação.
5. Além das características do item anterior, há considerações especiais que exigem utilização de ferramentas de análise de desempenho, para a distribuição do sistema e seus componentes, nas unidades processadoras.

5. **Volume de transações**: Consiste na avaliação do nível de influência do volume de transações no projeto, desenvolvimento, implantação e manutenção do sistema.

0. Não estão previstos períodos de picos de volume de transação.
1. Estão previstos picos de transações mensalmente, trimestralmente, anualmente ou em certo período do ano.
2. São previstos picos semanais.
3. São previstos picos diários.
4. Alto volume de transações foi estabelecido pelo usuário, ou o tempo de resposta necessário atinge nível alto o suficiente para requerer análise de desempenho na fase de projeto.
5. Além do descrito no item anterior, é necessário utilizar ferramentas de análise de desempenho nas fases de projeto, desenvolvimento e/ou implantação.

6. **Entrada de dados *on-line***: A análise desta característica permite quantificar o nível de influência exercida pela utilização de entrada de dados no modo *on-line* no sistema.

0. Todas as transações são processadas em modo *batch*.
1. De 1% a 7% das transações são entradas de dados *on-line*.
2. De 8% a 15% das transações são entradas de dados *on-line*.
3. De 16% a 23% das transações são entradas de dados *on-line*.
4. De 24% a 30% das transações são entradas de dados *on-line*.
5. Mais de 30% das transações são entradas de dados *on-line*.

7. **Usabilidade**: a análise desta característica permite quantificar o grau de influência relativo aos recursos implementados com vista a tornar o sistema amigável, permitindo incrementos na eficiência e satisfação do usuário final, tais como:

- Auxílio à navegação (teclas de função, acesso direto e menus dinâmicos)
- Menus Documentação e *help on-line*
- Movimento automático do cursor.
- Movimento horizontal e vertical de tela.
- Impressão remota (via transações *on-line*)
- Teclas de função preestabelecidas.
- Processos batch submetidos a partir de transações *on-line*
- Utilização intensa de campos com vídeo reverso, intensificados, sublinhados, coloridos e outros indicadores.
- Impressão da documentação das transações on-line através de *hard copy*
- Utilização de mouse
- Menus *pop-up*
- O menor número possível de telas para executar as funções de negócio.
- Suporte bilingue (contar como 4 itens)
- Suporte multilíngue. (contar como 6 itens)

Pontuação:

0. Nenhum dos itens descritos.
1. De um a três itens descritos.
2. De quatro a cinco dos itens descritos.
3. Mais de cinco dos itens descritos, mas não há requisitos específicos do usuário quanto à usabilidade do sistema.
4. Mais de cinco dos itens descritos e foram estabelecidos requisitos quanto à usabilidade fortes o suficiente para gerarem atividades específicas envolvendo fatores, tais como minimização da digitação, para mostrar inicialmente os valores utilizados com mais frequência.
5. Mais de cinco dos itens descritos e foram estabelecidos requisitos quanto à usabilidade fortes o suficiente para requerer ferramentas e processos especiais para demonstrar antecipadamente que os objetivos foram alcançados.

8. **Atualizações *on-line***: Mede a influência no desenvolvimento do sistema face à utilização de recursos que visem a atualização dos Arquivos Lógicos Internos, no modo *on-line*.

0. Nenhuma.
1. Atualização *on-line* de um a três arquivos lógicos internos. O volume de atualização é baixo e a recuperação de dados é simples.
2. Atualização *on-line* de mais de três arquivos lógicos internos. O volume de atualização é baixo e a recuperação dos dados é simples.
3. Atualização *on-line* da maioria dos arquivos lógicos internos.
4. Em adição ao item anterior, é necessário proteção contra perdas de dados que foi projetada e programada no sistema.
5. Além do item anterior, altos volumes trazem considerações de custo no processo de recuperação. Processos para automatizar a recuperação foram incluídos minimizando a intervenção do operador.

9. **Processamento complexo**: a complexidade de processamento influencia no dimensionamento do sistema, e, portanto, deve ser quantificado o seu grau de influência, com base nas seguintes categorias:

- Processamento especial de auditoria e/ou processamento especial de segurança foram considerados na aplicação;
- Processamento lógico extensivo;
- Processamento matemático extensivo;
- Processamento gerando muitas exceções, resultando em transações incompletas que devem ser processadas novamente. Exemplo: transações de auto-atendimento bancário interrompidas por problemas de comunicação ou com dados incompletos;
- Processamento complexo para manusear múltiplas possibilidades de entrada/saída. Exemplo: multimídia.

Pontuação

0. Nenhum dos itens descritos.
1. Apenas um dos itens descritos.
2. Dois dos itens descritos.
3. Três dos itens descritos.
4. Quatro dos itens descritos.
5. Todos os cinco itens descritos.

10. **Reusabilidade**: a preocupação com o reaproveitamento de parte dos programas de uma aplicação em outras aplicações implica em cuidados com padronização. O grau de influência no dimensionamento do sistema é quantificado observando-se os seguintes aspectos:

0. Nenhuma preocupação com reutilização de código.
1. Código reutilizado foi usado somente dentro da aplicação.
2. Menos de 10% da aplicação foi projetada prevendo utilização posterior do código por outra aplicação.
3. 10% ou mais da aplicação foi projetada prevendo utilização posterior do código por outra aplicação.
4. A aplicação foi especificamente projetada e/ou documentada para ter seu código reutilizado por outra aplicação e a aplicação é customizada pelo usuário em nível de código fonte.
5. A aplicação foi especificamente projetada e/ou documentada para ter seu código facilmente reutilizado por outra aplicação e a aplicação é customizada para uso através de parâmetros que podem ser alterados pelo usuário.

11. **Facilidade de implantação**: a quantificação do grau de influência desta característica é feita, observando-se o plano de conversão e implantação e/ou ferramentas utilizadas durante a fase de testes do sistema.

0. Nenhuma consideração especial foi estabelecida pelo usuário e nenhum procedimento especial é requerido na implantação.
1. Nenhuma consideração especial foi estabelecida pelo usuário, mas procedimentos especiais são necessários na implementação.
2. Requisitos de conversão e implantação foram estabelecidos pelo usuário e roteiro de conversão e implantação foram providos e testados. O impacto da conversão no projeto não é considerado importante.
3. Requisitos de conversão e implantação foram estabelecidos pelo usuário e roteiro de conversão e implantação foram providos e testados. O impacto da conversão no projeto é considerado importante.
4. Além do item 2, conversão automática e ferramentas de implantação foram providas e testadas.
5. Além do item 3, conversão automática e ferramentas de implantação foram providas e testadas.

12. **Facilidade operacional**: a análise desta característica permite quantificar o nível de influência na aplicação, com relação a procedimentos operacionais automáticos que reduzem os procedimentos manuais, bem como mecanismos de inicialização, salvamento e recuperação, verificados durante os testes do sistema.

0. Nenhuma consideração especial de operação, além do processo normal de salvamento foi estabelecida pelo usuário.

1-4. Verifique quais das seguintes afirmativas podem ser identificadas na aplicação. Selecione as que forem aplicadas. Cada item vale um ponto, exceto se definido explicitamente:

- Foram desenvolvidos processos de inicialização, salvamento e recuperação, mas a intervenção do operador é necessária.
- Foram estabelecidos processos de inicialização, salvamento e recuperação, e nenhuma intervenção do operador é necessária (conte como dois itens)
- A aplicação minimiza a necessidade de montar fitas magnéticas.
- A aplicação minimiza a necessidade de manuseio de papel.

5. A aplicação foi desenhada para trabalhar sem operador, nenhuma intervenção do operador é necessária para operar o sistema além de executar e encerrar a aplicação. A aplicação possui rotinas automáticas para recuperação em caso de erro.

13. **Múltiplos Locais e Organizações do Usuário**: consiste na análise da arquitetura do projeto, observando-se a necessidade de instalação do sistema em diversos lugares.

0. Os requisitos do usuário não consideraram a necessidade de instalação em mais de um local.
1. A necessidade de múltiplos locais foi considerada no projeto e a aplicação foi desenhada para operar apenas em ambientes de software e hardware idênticos.
2. A necessidade de múltiplos locais foi considerada no projeto e a aplicação está preparada para trabalhar apenas em ambientes similares de software e hardware.
3. A necessidade de múltiplos locais foi considerada no projeto e a aplicação está preparada para trabalhar em diferentes ambientes de hardware e/ou software.
4. Plano de documentação e manutenção foram providos e testados para suportar a aplicação em múltiplos locais, além disso, os itens 1 ou 2 caracterizam a aplicação.
5. Plano de documentação e manutenção foram providos e testados para suportar a aplicação em múltiplos locais, além disso, o item 3 caracteriza a aplicação.

14. **Facilidade de mudanças**: focaliza a preocupação com a influência da manutenção no desenvolvimento do sistema. Esta influência deve ser quantificada baseando na observação de atributos, tais como:

- disponibilidade de facilidades como consultas e relatórios flexíveis para atender necessidades simples (conte como 1 item);
- disponibilidade de facilidades como consultas e relatórios flexíveis para atender necessidades de complexidade média (conte como 2 itens);
- disponibilidade de facilidades como consultas e relatórios flexíveis para atender necessidades complexas (conte 3 itens);
- se os dados de controle são armazenados em tabelas que são mantidas pelo usuário através de processos on-line, mas mudanças têm efeitos somente no dia seguinte;
- se os dados de controle são armazenados em tabelas que são mantidas pelo usuário através de processos on-line, as mudanças têm efeito imediatamente (conte como 2 itens).

Pontuação

0. Nenhum dos itens descritos.
1. Um dos itens descritos.
2. Dois dos itens descritos.
3. Três dos itens descritos.
4. Quatro dos itens descritos.
5. Todos os cinco itens descritos.

Referência:

R. Dias, "Análise por Pontos de Função: Uma Técnica para Dimensionamento de Sistemas de Informação", on-line. Disponível em: www.presidentekennedy.br/resi/edicao03/artigo02.pdf. Último acesso: 13.05.2004.