UFES - Universidade Federal do Espírito Santo
Centro Tecnológico
Departamento de Informática

Engenharia de Software

Notas de Aula

2018

Curso: **Engenharia da Computação**

Monalessa Perini Barcellos

(monalessa@inf.ufes.br)

Este material é uma adaptação feita pela profa. Monalessa Perini Barcellos a partir do conteúdo das Notas de Aula das disciplinas Engenharia de Software, Engenharia de Requisitos e Projeto de Sistemas elaboradas pelo prof. Ricardo de Almeida Falbo para as referidas disciplinas ministradas no curso Ciência da Computação/UFES.

Sumário

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/Engenharia 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. Começa-se 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, pode-se 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 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 a pergunta for feita 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, pode-se 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, se está 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.

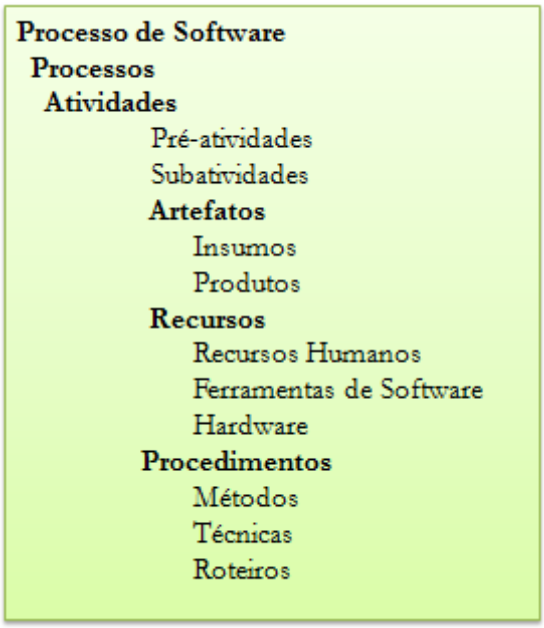

Figura 1.1 - Elementos que compõem um Processo de Software.

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.

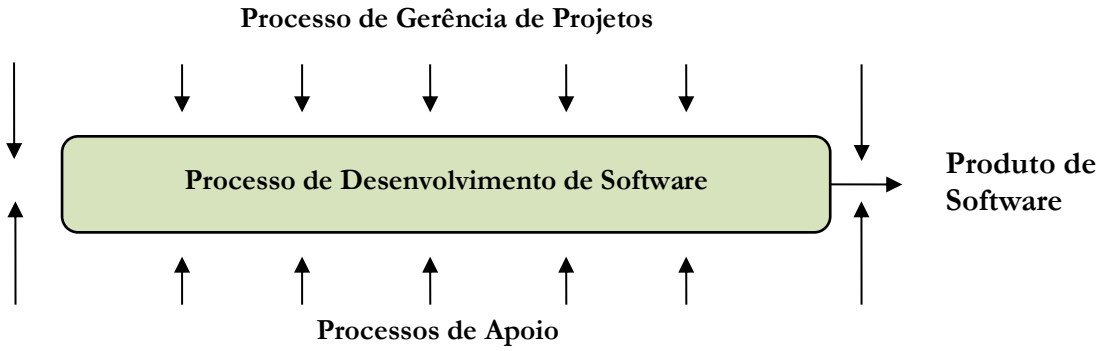

Figura 1.2 – Decomposição do Processo de Software.

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 duas partes: na Parte I são abordados os processos de desenvolvimento e manutenção de software e 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 de Software*: 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 – *Engenharia de Requisitos:* aborda requisitos e tipos de requisitos e procura dar uma visão geral da Engenharia de Requisitos.
- Capítulo 4 – *Levantamento Requisitos*: aborda técnicas e diretrizes para o levantamento e documentação de requisitos.

- Capítulo 5 – *Análise de Requisitos:* aborda aspectos, técnicas e métodos para análise de requisitos, com destaque para a Modelagem de Casos de Uso e a Modelagem de Classes.
- Capítulo 6 – *Projeto de Sistemas*: aborda os conceitos básicos de projeto de sistemas de software, tratando da arquitetura do sistema a ser desenvolvido e do projeto de seus componentes.
- Capítulo 7 – *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 8 – *Entrega e Manutenção*: discute as questões relacionadas à entrega do sistema para o cliente, tais como o treinamento e a documentação de entrega, e o processo de manutenção de software.

PARTE II: Gerência de Software

- Capítulo 9 – *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 10 – *Gerência de Projetos*: são abordadas as principais atividades do processo de gerência de projetos: planejamento, acompanhamento e encerramento de projetos.
- Capítulo 11 – *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.

Além dos 11 capítulos, este material inclui um anexo com a descrição detalhada da técnica Análise de Pontos de Função, a qual é introduzida no Capítulo 10.

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 estruturado adota uma visão de desenvolvimento baseada em um modelo entrada-processamento-saída, onde os dados são considerados separadamente das funções que os transformam e a decomposição funcional é usada intensamente. Por outro lado, 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. Neste texto, as atividades do processo de desenvolvimento de software são discutidas à luz do paradigma orientado a objetos.

PARTE I

Desenvolvimento e Manutenção de Software

Capítulo 2 – Visão Geral do Processo de Desenvolvimento de Software

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.

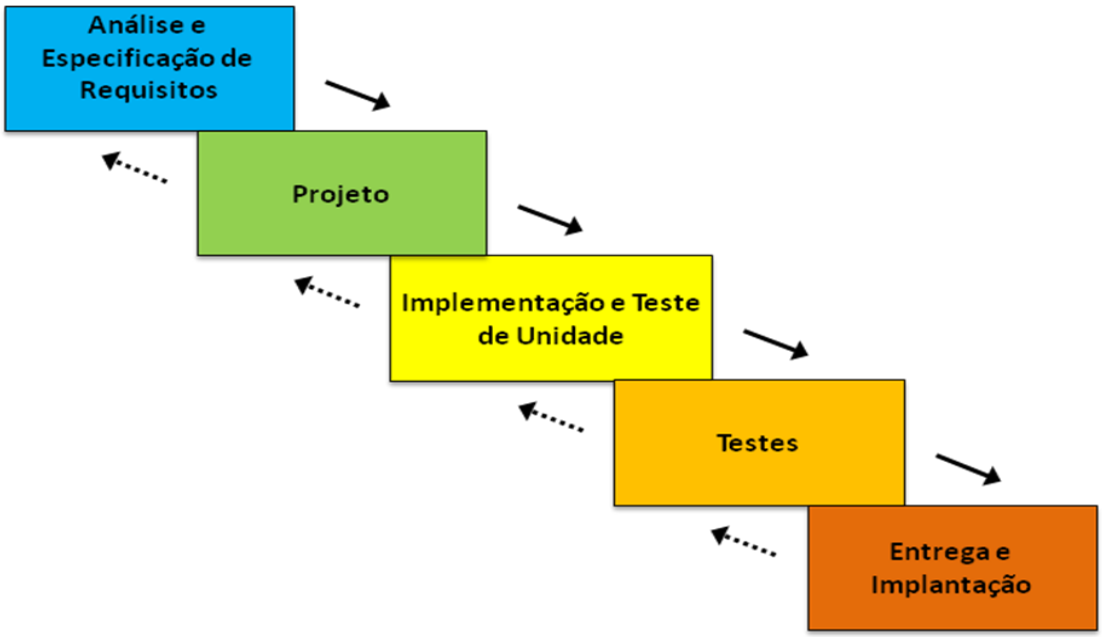

Figura 2.1 - O Modelo em Cascata.

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.

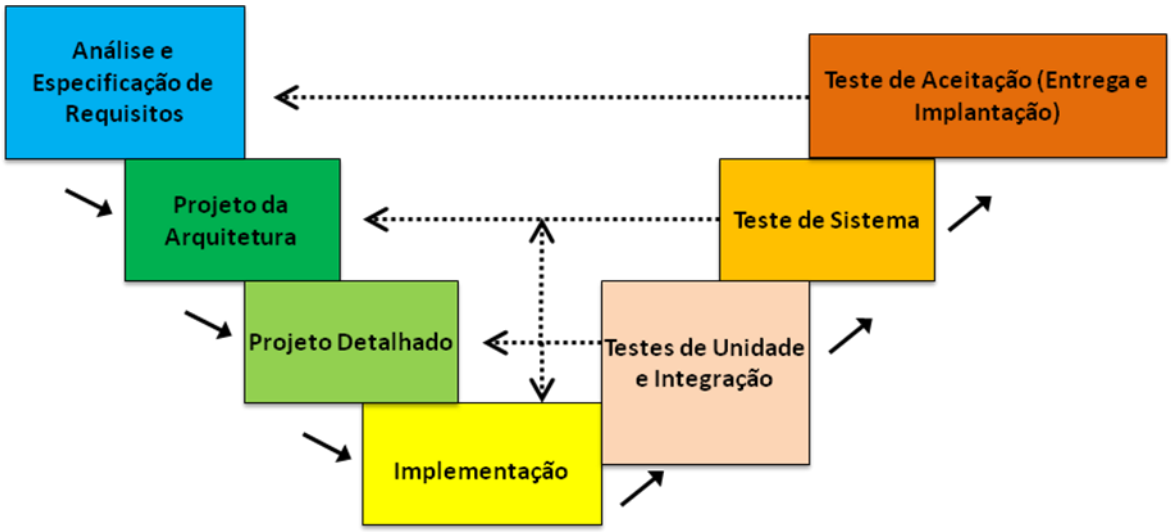

Figura 2.2 - O Modelo em V.

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).

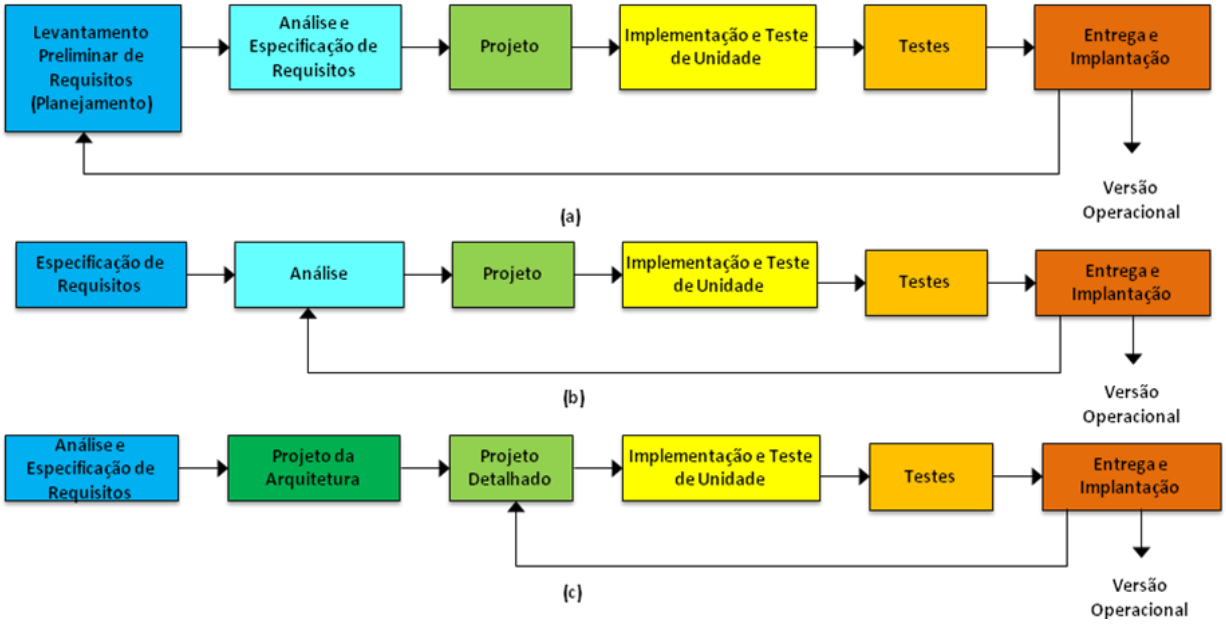

Figura 2.4 – Variações do Modelo Incremental.

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.

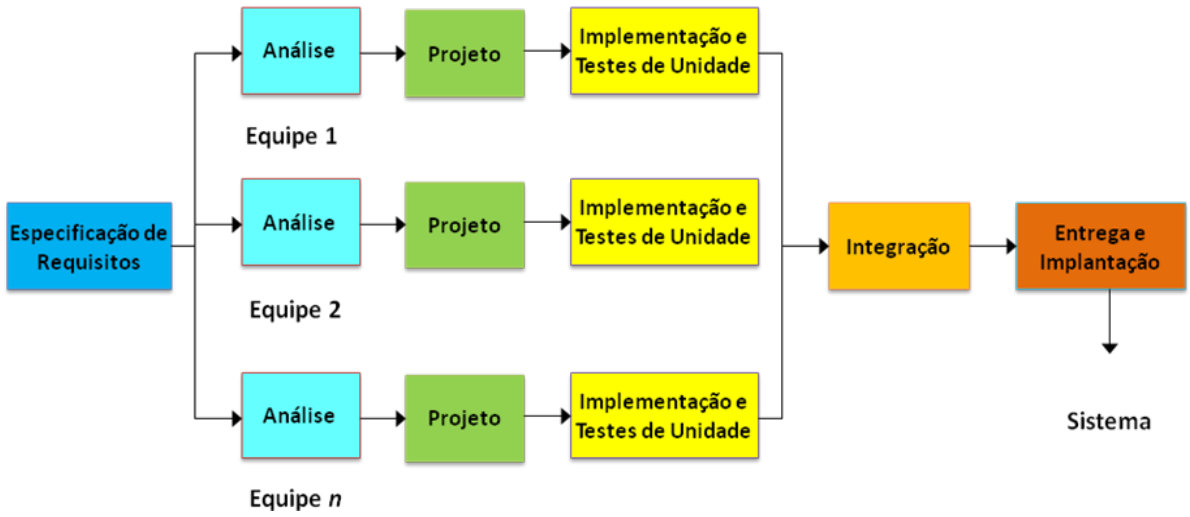

Figura 2.5 – O Modelo RAD.

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.

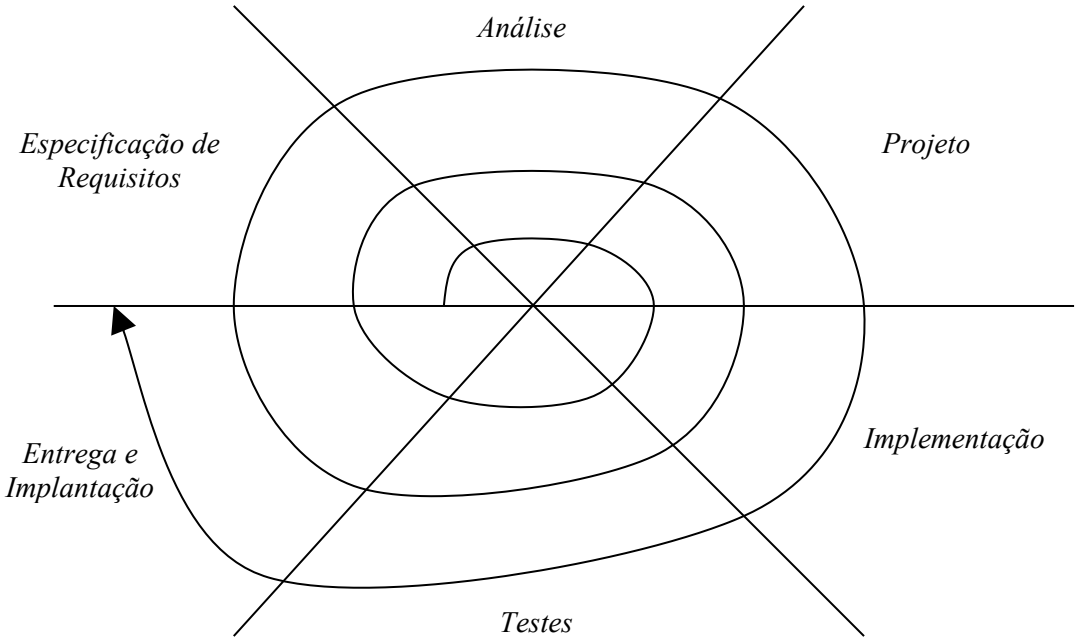

Figura 2.6 – O Modelo Espiral.

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).

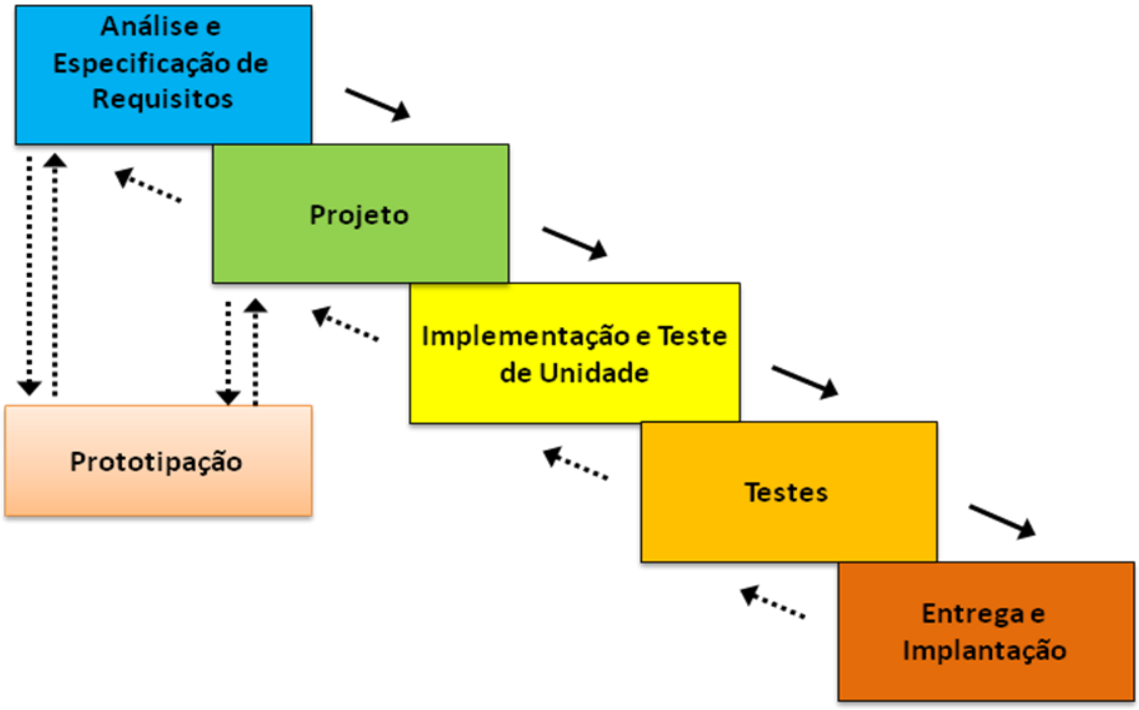

Figura 2.7 - O Modelo em Cascata com Prototipação.

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).

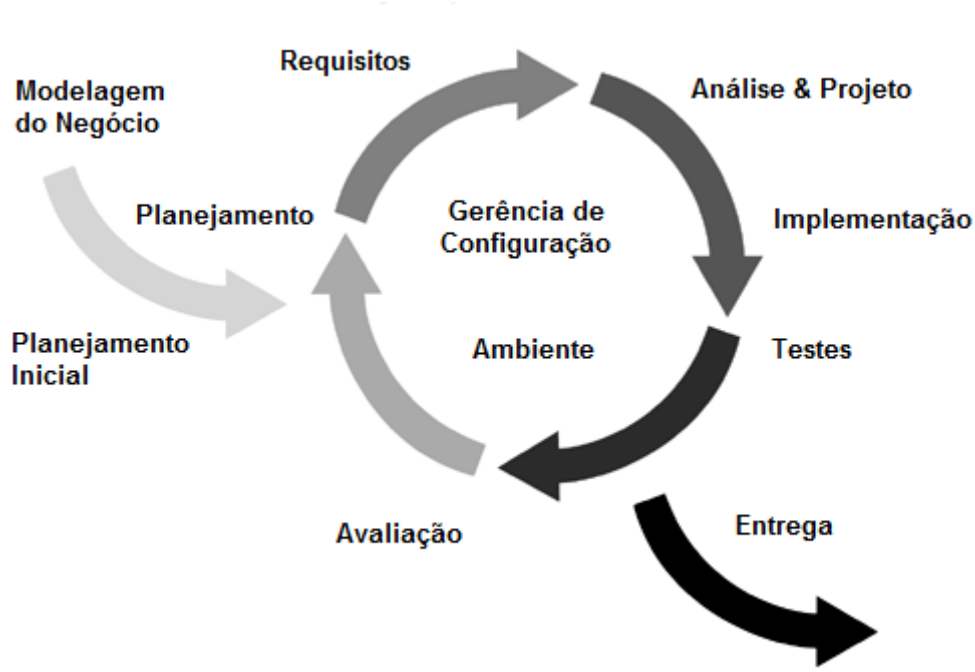

Figura 2.8 – Desenvolvimento Iterativo no RUP (adaptado de (KROLL; KRUCHTEN, 2003)).

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.

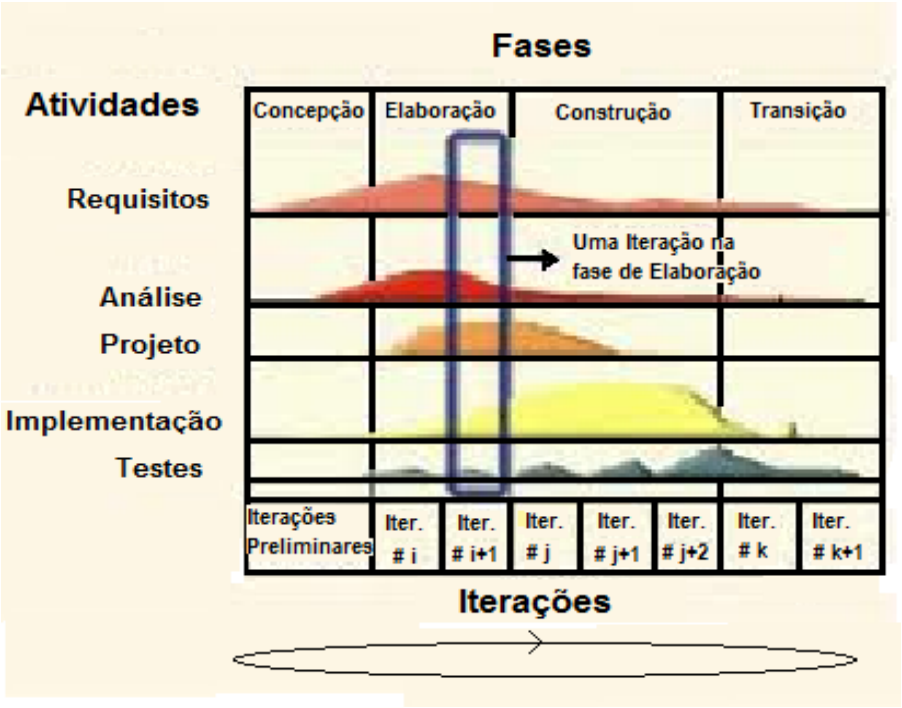

Figura 2.9 - O Modelo de Processo do RUP.

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.

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 – Engenharia de Requisitos

Requisitos têm um papel central no processo de software, sendo considerados um fator determinante para o sucesso ou fracasso de um projeto de software. O processo de levantar, analisar, documentar, gerenciar e controlar a qualidade dos requisitos é chamado de Engenharia de Requisitos.

Este capítulo dá uma visão geral da Engenharia de Requisitos, abordando, na seção 3.1, o conceito de requisito e seus tipos, e, na seção 3.2, o processo de engenharia de requisitos, discutindo cada uma de suas atividades.

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. Existem diversas definições para requisito de software na literatura, dentre elas:

- Requisitos são descrições dos serviços que devem ser providos pelo sistema e de suas restrições operacionais (SOMMERVILLE, 2007).
- Um requisito é uma característica do sistema ou a descrição de algo que o sistema é capaz de realizar para atingir seus objetivos (PFLEEGER, 2004).
- Um requisito é alguma coisa que o produto tem de fazer ou uma qualidade que ele precisa apresentar (ROBERTSON; ROBERTSON, 2006).

Com base nessas e em outras definições, pode-se dizer que os requisitos de um sistema incluem *especificações dos serviços que o sistema deve prover, restrições sob as quais ele deve operar, propriedades gerais do sistema e restrições que devem ser satisfeitas no seu processo de desenvolvimento*.

As várias definições acima apresentadas apontam para a existência de diferentes tipos de requisitos. Uma classificação amplamente aceita quanto ao tipo de informação documentada por um requisito faz a distinção entre requisitos funcionais e requisitos não funcionais.

- **Requisitos Funcionais**: são declarações de serviços que o sistema deve prover, descrevendo o que o sistema deve fazer (SOMMERVILLE, 2007). Um requisito funcional descreve uma interação entre o sistema e o seu ambiente (PFLEEGER, 2004), podendo descrever, ainda, como o sistema deve reagir a entradas específicas, como o sistema deve se comportar em
- **Requisitos Não Funcionais**: descrevem restrições sobre os serviços ou funções oferecidos pelo sistema (SOMMERVILLE, 2007), as quais limitam as opções para criar uma solução para o problema (PFLEEGER, 2004). Neste sentido, os requisitos não funcionais são muito importantes para a fase de projeto (design), servindo como base para a tomada de decisões nessa fase.

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.

Os requisitos não funcionais têm origem nas necessidades dos usuários, em restrições de orçamento, em políticas organizacionais, em necessidades de interoperabilidade com outros sistemas de software ou hardware ou em fatores externos como regulamentos e legislações (SOMMERVILLE, 2007). Assim, os requisitos não funcionais podem ser classificados quanto à sua origem. Existem diversas classificações de requisitos não funcionais. Sommerville (2007), por exemplo, classifica-os em:

- **Requisitos de produto**: especificam o comportamento do produto (sistema). Referem-se a atributos de qualidade que o sistema deve apresentar, tais como confiabilidade, usabilidade, eficiência, portabilidade, manutenibilidade e segurança.
- **Requisitos organizacionais**: são derivados de metas, políticas e procedimentos das organizações do cliente e do desenvolvedor. Incluem requisitos de processo (padrões de processo e modelos de documentos que devem ser usados), requisitos de implementação (tal como a linguagem de programação a ser adotada), restrições de entrega (tempo para chegar ao mercado - *time to market*, restrições de cronograma etc.), restrições orçamentárias (custo, custo-benefício) etc.
- **Requisitos externos**: referem-se a todos os requisitos derivados de fatores externos ao sistema e seu processo de desenvolvimento. Podem incluir requisitos de interoperabilidade com sistemas de outras organizações, requisitos legais (tais como requisitos de privacidade) e requisitos éticos.

No que se refere aos RNFs de produto, eles podem estar relacionados a propriedades emergentes do sistema como um todo, ou seja, propriedades que não podem ser atribuídas a uma parte específica do sistema, mas que, ao contrário, só aparecem após a integração de seus componentes, tal como confiabilidade (SOMMERVILLE, 2007). Contudo, algumas vezes, essas características podem estar associadas a uma função específica ou a um conjunto de funções. Por exemplo, uma certa função pode ter restrições severas de desempenho, enquanto outras funções do mesmo sistema podem não apresentar tal restrição.

Os requisitos devem ser redigidos de modo a serem passíveis de entendimento pelos diversos interessados (*stakeholders*). Clientes[1], usuários finais e desenvolvedores são todos interessados em requisitos, mas têm expectativas diferentes. Enquanto desenvolvedores e usuários finais têm interesse em detalhes técnicos, clientes requerem descrições mais abstratas. Assim, é útil apresentar requisitos em diferentes níveis de descrição. Sommerville (2007) sugere dois níveis de descrição de requisitos:

- **Requisitos de Cliente ou de Usuário:** são declarações em linguagem natural acompanhadas de diagramas intuitivos de quais serviços são esperados do sistema e das restrições sob as quais ele deve operar. Devem estar em um nível de abstração

[1] É importante notar a distinção que se faz aqui entre clientes e usuários finais. Consideram-se clientes aqueles que contratam o desenvolvimento do sistema e que, muitas vezes, não usarão diretamente o sistema. Eles estão mais interessados nos resultados da utilização do sistema pelos usuários do que no sistema em si. Usuários, por outro lado, são as pessoas que utilizarão o sistema em seu dia a dia. Ou seja, os usuários são as pessoas que vão operar ou interagir diretamente com o sistema.

mais alto, de modo que sejam compreensíveis pelos clientes e usuários do sistema que não possuem conhecimento técnico.

- **Requisitos de Sistema:** definem detalhadamente as funções, serviços e restrições do sistema. São versões expandidas dos requisitos de cliente usados pelos desenvolvedores para projetar, implementar e testar o sistema. Como requisitos de sistema são mais detalhados, as especificações em linguagem natural são insuficientes e para especificá-los, notações mais especializadas devem ser utilizadas.

Vale destacar que esses níveis de descrição de requisitos são aplicados em momentos diferentes e com propósitos distintos. Requisitos de cliente são elaborados nos estágios iniciais do desenvolvimento (levantamento preliminar de requisitos) e servem de base para um entendimento entre clientes e desenvolvedores acerca do que o sistema deve contemplar. Esses requisitos são, normalmente, usados como base para a contratação e o planejamento do projeto. Requisitos de sistema, por sua vez, são elaborados como parte dos esforços diretos para o desenvolvimento do sistema, capturando detalhes importantes para as fases técnicas posteriores do processo de desenvolvimento, a saber: projeto, implementação e testes.

Entretanto, não se deve perder de vista que requisitos de sistema são derivados dos requisitos de cliente. Os requisitos de sistema acrescentam detalhes, explicando os serviços e funções a serem providos pelo sistema em desenvolvimento. Os interessados nos requisitos de sistema necessitam conhecer mais precisamente o que o sistema fará, pois eles estão preocupados com o modo como o sistema apoiará os processos de negócio ou porque estão envolvidos na sua construção (SOMMERVILLE, 2007).

Uma vez que requisitos de cliente e de sistema têm propósitos e público alvo diferentes, é útil descrevê-los em documentos diferentes. Pfleeger (2004) sugere que dois tipos de documentos de requisitos sejam elaborados:

- **Documento de Definição de Requisitos**: deve ser escrito de maneira que o cliente possa entender, i.e., na forma de uma listagem do quê o cliente espera que o sistema proposto faça. Ele representa um consenso entre o cliente e o desenvolvedor sobre o quê o cliente quer.
- **Documento de Especificação de Requisitos**: refina os requisitos de cliente em termos mais técnicos, apropriados para o desenvolvimento de software, sendo produzido por analistas de requisitos.

Vale ressaltar que deve haver uma correspondência direta entre cada requisito de usuário listado no documento de requisitos e os requisitos de sistema tratados no documento de especificação de requisitos.

3.2 O Processo de 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*.

A Engenharia de Requisitos é fundamental, pois possibilita, dentre outros, estimar custo e tempo de maneira mais precisas e melhor gerenciar mudanças em requisitos. Dentre os

problemas de um processo de engenharia de requisitos ineficiente, podem-se citar (KOTONYA; SOMMERVILLE, 1998): (i) requisitos inconsistentes, (ii) produto final com custo maior do que o esperado, (iii) software instável e com altos custos de manutenção e (iv) clientes insatisfeitos.

De maneira geral, o processo de Engenharia de Requisitos envolve as atividades (KOTONYA; SOMMERVILLE, 1998) ilustradas na Figura 3.1. O processo começa pelo levantamento de requisitos, que deve levar em conta necessidades dos usuários e clientes, informações de domínio, sistemas existentes, regulamentos, leis etc. Uma vez identificados requisitos, é possível iniciar a atividade de análise, quando os requisitos levantados são usados como base para a modelagem do sistema. Tanto no levantamento quanto na análise de requisitos, é importante documentar requisitos e modelos. Conforme discutido anteriormente, para documentar requisitos, dois documentos são normalmente utilizados: o Documento de Definição de Requisitos, contendo uma lista dos requisitos de cliente identificados, e o Documento de Especificação de Requisitos, que registra os requisitos de sistema e os vários diagramas resultantes do trabalho de análise. Os documentos produzidos são, então, verificados e validados. Adicionalmente, um esforço de garantia da qualidade deve ser realizado, visando garantir conformidade em relação a padrões e ao processo estabelecidos pela organização. Caso clientes, usuários e desenvolvedores estejam de acordo com os requisitos, o processo de desenvolvimento pode avançar; caso contrário, deve-se retornar à atividade correspondente para resolver os problemas identificados. Em paralelo a todas as atividades anteriormente mencionadas, há a gerência de requisitos, que se ocupa em gerenciar mudanças nos requisitos.

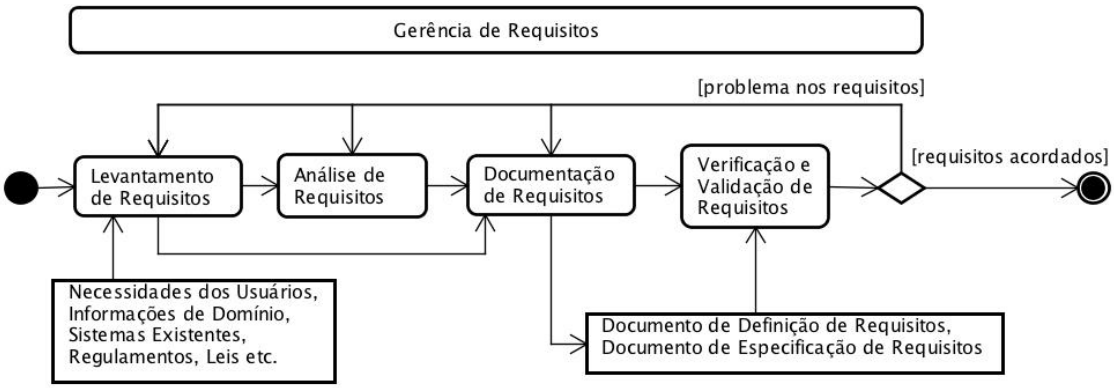

Figura 3.1 - O Processo da Engenharia de Requisitos (adaptado de (KOTONYA; SOMMERVILLE, 1998))**.**

- 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.

Vale destacar que não há limites bem definidos entre as atividades acima citadas. Na prática, elas são intercaladas e existe um alto grau de iteração e feedback entre elas. Idealmente, deve-se começar com um levantamento preliminar de requisitos que considere apenas requisitos de cliente. Esses requisitos devem ser documentados em um Documento de Definição de Requisitos e avaliados (verificação e validação). Esse documento deve ser usado como base para a contratação do projeto. Caso haja acordo em relação aos requisitos de cliente, pode-se iniciar um ciclo de levantamento detalhado de requisitos e análise. O processo é executado até que todos os usuários estejam satisfeitos e concordem com os requisitos ou até que a pressão do cronograma precipite o início da fase de projeto, o que é indesejável (KOTONYA; SOMMERVILLE, 1998).

Além disso, ao se adotar um modelo de ciclo de vida iterativo, essas atividades podem ser realizadas muitas vezes. Neste caso, uma vez contratado o projeto e, portanto, obtido um acordo em relação aos requisitos de cliente iniciais, pode-se iniciar um ciclo de levantamento detalhado de requisitos e análise, produzindo uma versão do Documento de Especificação de Requisitos. Havendo acordo em relação aos requisitos e modelos contidos nessa primeira versão do documento, o desenvolvimento pode prosseguir para a porção tratada nessa iteração (projeto, implementação e testes), enquanto um novo ciclo de levantamento e análise se inicia para tratar outros requisitos de cliente ainda não contemplados

Considerando o processo de software como um todo, das cinco atividades do processo de Engenharia de Requisitos anteriormente listadas, apenas as três primeiras fazem parte do processo de desenvolvimento propriamente dito (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.

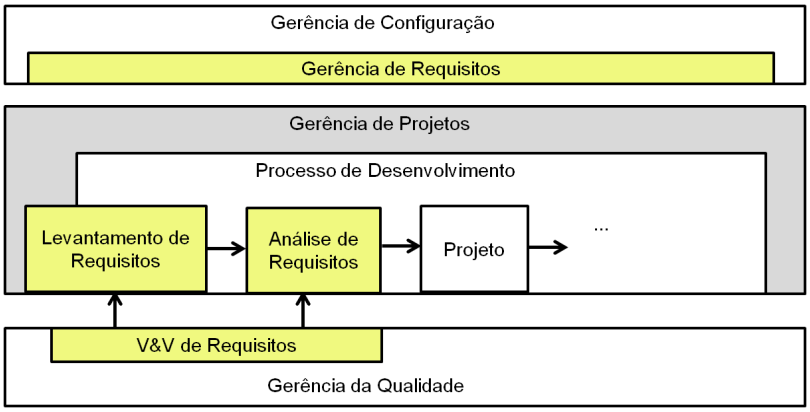

Figura 3.2 - As Atividades da Engenharia de Requisitos (destacadas em amarelo) no Processo de Software

A seguir é apresenta uma visão geral para cada uma das atividades da Engenharia de Requisitos. As atividades de levantamento e análise de requisitos são estudadas com maiores detalhes nos capítulos 4 e 5 destas notas de aula. As demais atividades ocorrem no contexto das atividades de gerência que são discutidas na Parte II destas notas de aula.

3.2.1 Levantamento de Requisitos

O levantamento de requisitos corresponde à fase inicial do processo de ER e envolve atividades de descoberta dos requisitos. Nessa fase, um esforço conjunto de clientes, usuários e especialistas de domínio é necessário, com o objetivo de entender a organização, seus processos, necessidades, deficiências dos sistemas de software atuais, possibilidades de melhorias, bem como restrições existentes. Trata-se de uma atividade complexa que não se resume somente a perguntar às pessoas o que elas desejam, mas sim analisar cuidadosamente a organização, o domínio da aplicação e os processos de negócio no qual o sistema será utilizado (KOTONYA; SOMMERVILLE, 1998).

Para levantar quais são os requisitos de um sistema, devem-se obter informações dos interessados (*stakeholders*), consultar documentos, obter conhecimentos do domínio e estudar o negócio da organização. Neste contexto, quatro dimensões devem ser consideradas utilizado (KOTONYA; SOMMERVILLE, 1998):

- **Entendimento do domínio da aplicação:** entendimento geral da área na qual o software a ser desenvolvido está inserido;
- **Entendimento do problema:** entendimento dos detalhes do problema específico a ser resolvido com o auxílio do sistema a ser desenvolvido;
- **Entendimento do negócio:** entender como o sistema afetará a organização e como contribuirá para que os objetivos do negócio e os objetivos gerais da organização sejam atingidos;
- **Entendimento das necessidades e das restrições dos interessados:** entender as demandas de apoio para a realização do trabalho de cada um dos interessados no sistema, entender os processos de trabalho a serem apoiados pelo sistema e o papel de eventuais sistemas existentes na execução e condução dos processos de trabalho. Consideram-se interessados no sistema, todas as pessoas que são afetadas pelo sistema de alguma maneira, dentre elas clientes, usuários finais e gerentes de departamentos onde o sistema será instalado.

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 dos 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).

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:* técnica amplamente utilizada, que consiste em conversas direcionadas com um propósito específico e com formato "pergunta-resposta". Seu objetivo é descobrir problemas a serem tratados, levantar procedimentos importantes e saber a opinião e as expectativas do entrevistado sobre o sistema.
- *Questionários:* o uso de questionários possibilita ao analista obter informações como postura, crenças, comportamentos e características de várias pessoas que serão afetas pelo sistema.
- *Observação:* consiste em observar o comportamento e o ambiente dos indivíduos de vários níveis organizacionais. Utilizando-se essa técnica, é possível capturar o que realmente é feito e qual tipo de suporte computacional é realmente necessário. Ajuda a confirmar ou refutar informações obtidas com outras técnicas e ajuda a identificar tarefas que podem ser automatizadas e que não foram identificadas pelos interessados.
- *Análise de documentos:* pela análise de documentos existentes na organização, analistas capturam informações e detalhes difíceis de conseguir por entrevista e observação. Documentos revelam um histórico da organização e sua direção.
- *Cenários:* com o uso desta técnica, um cenário de interação entre o usuário final e o sistema é montado e o usuário simula sua interação com o sistema nesse cenário, explicando ao analista o que ele está fazendo e de que informações ele precisa para realizar a tarefa descrita no cenário. O uso de cenários ajuda a entender requisitos, a expor o leque de possíveis interações e a revelar facilidades requeridas.
- *Prototipagem:* um protótipo é uma versão preliminar do sistema, muitas vezes não operacional e descartável, que é apresentada ao usuário para capturar informações específicas sobre seus requisitos de informação, observar reações iniciais e obter sugestões, inovações e informações para estabelecer prioridades e redirecionar planos.
- *Dinâmicas de Grupo:* há várias técnicas de levantamento de requisitos que procuram explorar dinâmicas de grupo para a descoberta e o desenvolvimento de requisitos, tais como *Brainstorming* e JAD (*Joint Application Development*). Na primeira, representantes de diferentes grupos de interessados engajam-se em uma discussão informal para rapidamente gerarem o maior número possível de ideias. Na segunda, interessados e analistas se reúnem para discutir problemas a serem solucionados e soluções possíveis. Com as diversas partes envolvidas

representadas, decisões podem ser tomadas e questões podem ser resolvidas mais rapidamente. A principal diferença entre JAD e *Brainstorming* é que, em JAD, tipicamente os objetivos do sistema já foram estabelecidos antes dos interessados participarem. Além disso, sessões JAD são normalmente bem estruturadas, com passos, ações e papéis de participantes definidos.

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 deve 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.

3.2.2 Análise de Requisitos

Uma vez preliminarmente identificados os requisitos, é possível iniciar a atividade de análise, quando os requisitos levantados devem ser refinados. Neste contexto, modelos são essenciais e diversos tipos de modelos podem ser utilizados. Esses modelos são representações gráficas que descrevem objetivos e processos de negócio, o problema a ser resolvido e o sistema a ser desenvolvido.

De maneira simples, um modelo é uma simplificação da realidade enfocando certos aspectos considerados relevantes segundo a perspectiva do modelo, e omitindo os demais. Modelos são construídos para se obter uma melhor compreensão da porção da realidade sendo modelada. 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:

- enfocar os aspectos chave, em detrimento de detalhes irrelevantes;
- 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;
- servir como base para a tomada de decisão
- formar a documentação do sistema.

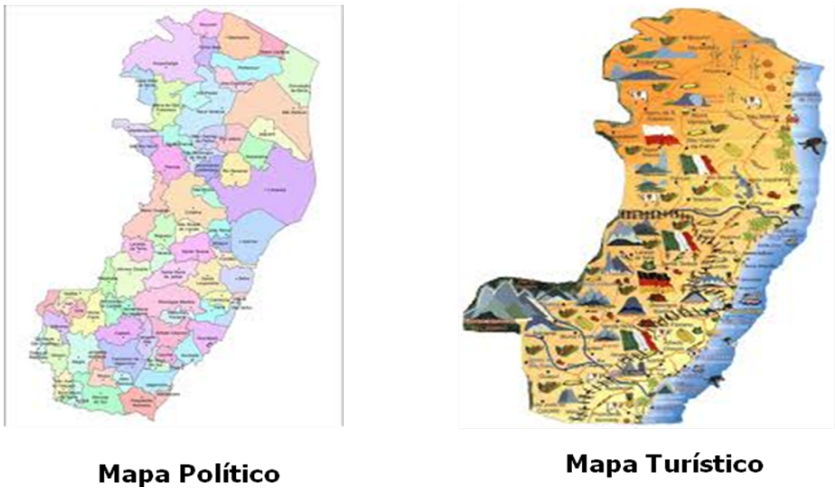

Figura 3.3 - Modelos como Abstrações da Realidade.

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

- *Perspectiva estrutural*: tem por objetivo descrever as informações que o sistema deve representar e gerenciar. Assim, busca modelar os conceitos, propriedades e relações do domínio que são relevantes para o sistema em desenvolvimento. Provê uma visão estática das informações que o sistema necessita tratar e, portanto, refere-se às representações que o sistema terá de prover para abstrair entidades do mundo real. Em outras palavras, esta perspectiva foca em "sobre o quê?" (quais informações?) o sistema opera. Diagramas de classes são usados para modelar esta perspectiva.
- *Perspectiva 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. Dessa forma, visa modelar o comportamento geral do sistema, de suas funcionalidades ou de uma entidade específica ao longo do tempo. Provê uma visão do comportamento do sistema ou de uma parcela do sistema. De maneira bem simples, o foco dessa perspectiva é no "quê" o sistema deve fazer (que funções ou serviços ele deve prover e como ele deve se comportar). Diagramas de casos de uso, diagramas de atividades, diagramas de estados e diagramas de interação são usados para modelar essa perspectiva.

Contudo, outras perspectivas podem ser alvo de modelos. A abordagem de Engenharia de Requisitos Baseada em Objetivos (*Goal-Oriented Requirements Engineering* - GORE), p.ex., assume que objetivos são uma perspectiva fundamental, pois estabelecem o "porquê" do sistema (e, portanto, dos elementos identificados em outras perspectivas). Na abordagem GORE, as razões para um novo sistema (ou uma nova versão de um sistema) precisam ser explicitadas em termos de objetivos a serem satisfeitos por ele (LAMSWEERDE, 2009) e, para tal, modelos de objetivos devem ser desenvolvidos.

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.

3.2.3 Documentação de Requisitos

Os requisitos e modelos capturados nas etapas anteriores devem ser descritos e apresentados em documentos. A documentação é, portanto, uma atividade de registro e oficialização dos resultados da Engenharia de Requisitos. Como resultado, um ou mais documentos devem ser produzidos.

Uma boa documentação fornece muitos benefícios, tais como (IEEE, 1998; NUSEIBEH; EASTERBROOK, 2000): (i) facilita a comunicação dos requisitos; (ii) reduz o esforço de desenvolvimento, pois sua preparação força usuários e clientes a considerar os requisitos atentamente, evitando retrabalho nas fases posteriores; (iii) fornece uma base realística para estimativas; (iv) fornece uma base para verificação e validação; (v) serve como base para futuras manutenções ou incremento de novas funcionalidades.

Diferentes interessados têm propósitos diferentes. Assim, pode ser útil ter mais do que um documento para registrar os resultados da engenharia de requisitos. Conforme discutido anteriormente, Pfleeger (2004) sugere que dois tipos de documentos de requisitos sejam elaborados: um Documento de Definição de Requisitos e um Documento de Especificação de Requisitos.

O Documento de Definição de Requisitos deve conter uma descrição do propósito do sistema, uma breve descrição do domínio do problema tratado pelo sistema e listas de requisitos funcionais e não funcionais, descritos em linguagem natural (requisitos de cliente). Para facilitar

a identificação e rastreamento dos requisitos, devem-se utilizar identificadores únicos para cada um dos requisitos listados. O público-alvo desse documento são clientes, usuários, gerentes (de cliente e de fornecedor) e desenvolvedores.

O Documento de Especificação de Requisitos deve conter os requisitos escritos a partir da perspectiva do desenvolvedor, devendo haver uma correspondência direta com os requisitos no Documento de Definição de Requisitos, de modo a se ter requisitos rastreáveis. Os vários modelos produzidos na fase de análise devem ser apresentados no Documento de Especificação de Requisitos, bem como glossários de termos usados e outras informações julgadas relevantes.

Deve-se observar que não há um padrão definido quanto à quantidade e ao nome dos documentos de requisitos. Há organizações que optam por ter apenas um documento de requisitos, contendo diversas seções, algumas tratando de requisitos de cliente, outras tratando de requisitos de sistema. Outras organizações, por sua vez, fazem uso de vários documentos distintos, capturando em documentos separados, por exemplo, requisitos funcionais, requisitos não funcionais, modelos de caso de uso e modelos estruturais e comportamentais. De fato, cabe a cada organização definir a quantidade, o nome e o conteúdo de cada documento. Para estruturar o conteúdo, é necessário que a organização defina seus modelos de documentos de requisitos.

3.2.4 Verificação e Validação de Requisitos

As atividades de Verificação & Validação (V&V) devem ser iniciadas o quanto antes no processo de desenvolvimento de software, pois quanto mais tarde os defeitos são encontrados, maiores os custos associados à sua correção (ROCHA; MALDONADO; WEBER, 2001). Uma vez que os requisitos são a base para o desenvolvimento, é fundamental que eles sejam cuidadosamente avaliados. Assim, os documentos produzidos durante a atividade de documentação de requisitos devem ser submetidos à verificação e à validação.

É importante realçar a diferença entre verificação e validação (voltaremos a essa discussão na Parte II destas notas de aula). O objetivo da verificação é assegurar que o software esteja sendo construído de forma correta. Deve-se verificar se os artefatos produzidos atendem aos requisitos estabelecidos e se os padrões organizacionais (de produto e processo) foram consistentemente aplicados. Por outro lado, o objetivo da validação é assegurar que o software que está sendo desenvolvido é o software correto, ou seja, assegurar que os requisitos, e o software deles derivado, atendem ao uso proposto (ROCHA; MALDONADO; WEBER, 2001).

No caso de requisitos, a verificação é feita, sobretudo, em relação à consistência entre requisitos e modelos e à conformidade com padrões organizacionais de documentação de requisitos. Já a validação tem de envolver a participação de usuários e clientes, pois somente eles são capazes de dizer se os requisitos atendem aos propósitos do sistema.

Nas atividades de V&V de requisitos, examinam-se os documentos de requisitos para assegurar que (PRESSMAN, 2006; KOTONYA; SOMMERVILLE, 1998; WIEGERS, 2003): (i) todos os requisitos do sistema tenham sido declarados de modo não-ambíguo, (ii) as inconsistências, conflitos, omissões e erros tenham sido detectados e corrigidos, (iii) os documentos estão em conformidade com os padrões estabelecidos e (iv) os requisitos realmente satisfazem às necessidades dos clientes e usuários. Em outras palavras, idealmente, um requisito, seja ele funcional ou não funcional, deve ser (WIEGERS, 2003; PFLEEGER, 2004):

- *Completo*: o requisito deve descrever completamente a funcionalidade a ser entregue (no caso de requisito funcional) ou a restrição a ser considerada (no caso

de requisito não funcional). Ele deve conter as informações necessárias para que o desenvolvedor possa projetar, implementar e testar essa funcionalidade ou restrição.

- *Correto*: cada requisito deve descrever exatamente a funcionalidade ou restrição a ser incorporada ao sistema.
- *Consistente*: o requisito não deve ser ambíguo ou conflitar com outro requisito.
- *Realista*: deve ser possível implementar o requisito com a capacidade e com as limitações do sistema e do ambiente de desenvolvimento.
- *Necessário*: o requisito deve descrever algo que o cliente realmente precisa ou que é requerido por algum fator externo ou padrão da organização.
- *Passível de ser priorizado*: os requisitos devem ter ordem de prioridade para facilitar o gerenciamento durante o desenvolvimento do sistema.
- *Verificável e passível de confirmação*: deve ser possível desenvolver testes para verificar se o requisito foi realmente implementado.
- *Rastreável*: deve ser possível identificar quais requisitos foram tratados em um determinado artefato, bem como identificar que produtos foram originados a partir de um requisito.

3.2.5 Gerência de Requisitos

Mudanças nos requisitos ocorrem ao longo de todo o processo de software, desde o levantamento e análise de requisitos até durante a operação do sistema. Elas são decorrentes de diversos fatores, tais como descoberta de erros, omissões, conflitos e inconsistências nos requisitos, melhor entendimento por parte dos usuários de suas necessidades, problemas técnicos, de cronograma ou de custo, mudança nas prioridades do cliente, mudanças no negócio, aparecimento de novos competidores, mudanças econômicas, mudanças na equipe, mudanças no ambiente onde o software será instalado e mudanças organizacionais ou legais. Para minimizar as dificuldades impostas por essas mudanças, é necessário gerenciar requisitos.

O processo de gerência de requisitos envolve as atividades que ajudam a equipe de desenvolvimento a identificar, controlar e rastrear requisitos e gerenciar mudanças de requisitos em qualquer momento ao longo do ciclo de vida do software (KOTONYA; SOMMERVILLE, 1998; PRESSMAN, 2006). Os principais objetivos desse processo são (KOTONYA; SOMMERVILLE, 1998):

- Gerenciar alterações nos requisitos acordados.
- Gerenciar relacionamentos entre requisitos.
- Gerenciar dependências entre requisitos e outros artefatos produzidos durante o processo de software.

Para apoiar a gerência de requisitos, matrizes de rastreabilidade podem ser produzidas. Elas relacionam os requisitos identificados entre si ou a um ou mais aspectos do sistema ou do seu ambiente, de modo que pode-se entender mais facilmente os impactos de uma modificação em um requisito nos diferentes aspectos do sistema.

Referências do Capítulo

IEEE, IEEE Recommended Practice for Software Requirements Specifications: IEEE Std 830-1998. New York: IEEE, 1998.

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

NUSEIBEH, B., EASTERBROOK, S., "Requirements engineering: a roadmap". In: Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 2000.

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

PRESSMAN, R.S., *Engenharia de Software*, McGraw-Hill, 6ª edição, 2006.

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

ROBERTSON, S., ROBERTSON, J., *Mastering the Requirements Process***.** 2nd Edition. Addison Wesley, 2006.

SOMMERVILLE, I., *Engenharia de Software*, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

van LAMSWEERDE, A., *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Willey, 2009.

WIEGERS, K.E., *Software Requirements: Practical techniques for gathering and managing requirements throughout the product development cycle*. 2nd Edition, Microsoft Press, Redmond, Washington, 2003.

Capítulo 4 – Levantamento de Requisitos

O levantamento de requisitos preocupa-se com o aprendizado e entendimento das necessidades dos usuários e patrocinadores do projeto, com o objetivo final de comunicar essas necessidades para os desenvolvedores do sistema. Uma parte substancial do levantamento de requisitos é dedicada a descobrir, extrair e aparar arestas dos desejos de potenciais interessados (AURUM; WOHLIN, 2005).

A fase de levantamento de requisitos envolve buscar, junto aos usuários, clientes e outros interessados, seus sistemas e documentos, todas as informações possíveis sobre as funções que o sistema deve executar (requisitos funcionais) e as restrições sob as quais ele deve operar (requisitos não funcionais). O produto principal dessa fase é o Documento de Definição de Requisitos.

O registro dos requisitos em um documento permite que possam ser verificados, validados e utilizados como base para outras atividades do processo de software. Para que sejam úteis, os requisitos têm de ser escritos em um formato compreensível por todos os interessados. Além disso, esses interessados devem interpretá-los uniformemente.

Uma vez que levantar requisitos não é tarefa fácil, é imprescindível que essa atividade seja cuidadosamente conduzida, fazendo uso de diversas técnicas. Dentre as diversas técnicas que podem ser aplicadas para o levantamento de requisitos, destacam-se: entrevistas, questionários, observação, investigação de documentos, prototipagem, cenários, abordagens em grupo, abordagens baseadas em objetivos e reutilização de requisitos.

Uma boa prática consiste em fazer o levantamento de requisitos de forma incremental. Inicialmente, em um levantamento preliminar de requisitos, apenas requisitos de cliente são capturados. Depois, em várias iterações, outros requisitos de cliente são capturados e requisitos de sistema vão sendo detalhados e especificados. Neste contexto, é importante realçar que o levantamento e a análise de requisitos são atividades estreitamente relacionadas e, portanto, devem ocorrer em paralelo. Assim, à medida que os requisitos vão sendo detalhados, eles devem ser modelados e especificados.

O levantamento preliminar de requisitos tem por objetivo prover uma visão do todo para se definir o que é mais importante e depois dividir o todo em partes para especificar os detalhes. Nessa fase, o levantamento é rápido e genérico, sendo feito em extensão e não em profundidade, i.e., o analista deve entender a extensão do que o sistema deve fazer, mas sem entrar em detalhes. Somente nos ciclos iterativos os requisitos serão detalhados, especificados e modelados (WAZLAWICK, 2004).

O levantamento de requisitos envolve um conjunto de atividades que deve permitir a comunicação, priorização, negociação e colaboração com todos os interessados relevantes. Deve prover, ainda, uma base para o aparecimento, descoberta e invenção de requisitos, como parte de um processo altamente interativo (AURUM; WOHLIN, 2005).

Zowghi e Coulin apud (AURUM; WOHLIN, 2005) indicam que as atividades do processo de levantamento de requisitos podem ser agrupadas em cinco tipos fundamentais de atividades, a saber:

- *Entendimento do Domínio de Aplicação*: é importante investigar e examinar em detalhes a porção do mundo real onde o sistema vai residir, dita o domínio de

aplicação. Aspectos sociais, políticos e organizacionais, bem como processos de trabalho existentes e problemas a serem resolvidos pelo sistema, precisam ser descritos em relação a metas e questões do negócio.

- *Identificação de Fontes de Requisitos*: requisitos podem estar espalhados em várias fontes e podem existir em vários formatos. Assim, podem existir muitas fontes de requisitos para um sistema e elas devem ser identificadas. Interessados representam a fonte de requisitos mais óbvia. Em especial, clientes, usuários e especialistas de domínio são os mais indicados para fornecerem informação detalhada sobre os problemas e as necessidades. Sistemas e processos existentes são também fontes de requisitos, especialmente quando o projeto envolve a substituição de um sistema existente. A documentação acerca desses sistemas e processos de negócio, incluindo manuais, formulários e relatórios, bem como de padrões da indústria, leis e regulamentações, provê informação útil sobre a organização e seu ambiente. Outras fontes incluem especificações de requisitos de sistemas (quando o sistema envolve hardware e software e existe uma especificação de mais alto nível), problemas reportados e solicitações de melhoria para sistemas correntes, e observação do dia a dia dos usuários. A necessidade de se obter requisitos a partir de múltiplas perspectivas e fontes ilustra bem a natureza de comunicação intensiva da Engenharia de Requisitos.
- *Análise de Interessados*: conforme citado anteriormente, interessados (*stakeholders*) são pessoas que têm interesse no sistema ou são afetadas de alguma maneira por ele e, portanto, precisam ser consultadas durante o levantamento de requisitos. Interessados incluem tanto pessoal interno quanto externo à organização. O cliente ou patrocinador do projeto é tipicamente o interessado mais aparente de um projeto. Contudo, os usuários são, na maioria das vezes, os interessados mais importantes. Outras partes cuja esfera de interesse pode ser afetada pela operação do sistema, tais como parceiros e clientes da organização, devem ser consideradas interessadas. Assim, um dos primeiros passos no processo de levantamento de requisitos consiste em analisar e envolver todos os interessados relevantes.
- *Seleção de Técnicas de Levantamento de Requisitos*: Nenhuma técnica individualmente é suficiente para levantar requisitos. Além disso, a escolha das técnicas a serem adotadas é fortemente dependente de características do projeto e de seus envolvidos. Diferentes técnicas devem ser empregadas, visando capturar diferentes tipos de informação e em diferentes estágios do processo de levantamento de requisitos. Assim, é importante selecionar adequadamente as técnicas a serem aplicadas.
- *Levantamento de Requisitos de Interessados e Outras Fontes*: Uma vez identificados as fontes de requisitos e os interessados relevantes, o levantamento de requisitos propriamente dito pode ser iniciado, aplicando-se as técnicas selecionadas.

Este capítulo aborda aspectos relacionados ao levantamento e documentação de requisitos. Na seção 4.1 são apresentadas diretrizes para a documentação de requisitos. A seção 4.2 trata especificamente de requisitos funcionais, enquanto que requisitos não funcionais são tratados na seção 4.3 e regras de negócio na seção 4.4.

4.1 Escrevendo e Documentando Requisitos

Os resultados do levantamento de requisitos têm de ser registrados em um documento, de modo que possam ser verificados, validados e utilizados como base para outras atividades do processo de software. Para que sejam úteis, os requisitos têm de ser escritos em um formato compreensível por todos os interessados. Além disso, esses interessados devem interpretá-los uniformemente.

Normalmente, requisitos são documentados usando alguma combinação de linguagem natural, modelos, tabelas e outros elementos. A linguagem natural é quase sempre imprescindível, uma vez que é a forma básica de comunicação compreensível por todos os interessados. Contudo, ela geralmente abre espaços para ambiguidades e má interpretação. Assim, é interessante procurar estruturar o uso da linguagem natural e complementar a descrição dos requisitos com outros elementos.

Diferentes abordagens podem ser usadas para documentar requisitos. Neste texto, em linha com o que foi discutido no Capítulo 3, sugerimos elaborar dois documentos: o Documento de Definição de Requisitos e o Documento de Especificação de Requisitos. O Documento de Definição de Requisitos é mais sucinto, escrito em um nível mais apropriado para o cliente e contempla apenas os requisitos de cliente. O Documento de Especificação de Requisitos é mais detalhado, escrito a partir da perspectiva dos desenvolvedores (PFLEEGER, 2004), normalmente contendo diversos modelos para descrever requisitos de sistema.

Os requisitos de cliente devem ser descritos de modo a serem compreensíveis pelos interessados no sistema que não possuem conhecimento técnico detalhado. Eles devem versar somente sobre o comportamento externo do sistema, em uma linguagem simples, direta e sem usar terminologia específica de software (SOMMERVILLE, 2007).

O Documento de Definição de Requisitos tem como propósito descrever os requisitos de cliente, tendo como público-alvo clientes, usuários, gerentes (de cliente e de fornecedor) e desenvolvedores. Há muitos formatos distintos propostos na literatura para documentos de requisitos. Neste texto, é proposta uma estrutura bastante simples para esse tipo de documento, contendo apenas quatro seções:

1. *Introdução:* breve introdução ao documento, descrevendo seu propósito e estrutura.
2. *Descrição do Propósito do Sistema:* descreve o propósito geral do sistema.
3. *Descrição do Minimundo:* apresenta, em um texto corrido, uma visão geral do domínio, do problema a ser resolvido e dos processos de negócio apoiados, bem como as principais ideias do cliente sobre o sistema a ser desenvolvido.
4. *Requisitos de Usuário:* apresenta os requisitos de usuário em linguagem natural. Três conjuntos de requisitos devem ser descritos nesta seção: requisitos funcionais, requisitos não funcionais e regras de negócio.

As três primeiras seções não têm nenhuma estrutura especial, sendo apresentadas na forma de um texto corrido. A introdução deve ser breve e basicamente descrever o propósito e a estrutura do documento, podendo seguir um padrão preestabelecido pela organização. A descrição do propósito do sistema deve ser direta e objetiva, tipicamente em um único parágrafo. Já a descrição do minimundo é um pouco maior, algo entre uma e duas páginas, descrevendo aspectos gerais e relevantes para um primeiro entendimento do domínio, do problema a ser resolvido e dos processos de negócio apoiados. Contém as principais ideias do

cliente sobre o sistema a ser desenvolvido, obtidas no levantamento preliminar e exploratório do sistema. Não se devem incluir detalhes.

A seção 4, por sua vez, não deve ter um formato livre. Ao contrário, deve seguir um formato estabelecido pela organização, contendo, dentre outros: identificador do requisito, descrição, tipo, origem, prioridade, responsável, interessados, dependências em relação a outros requisitos e requisitos conflitantes. A definição de padrões organizacionais para a definição de requisitos é essencial para garantir uniformidade e evitar omissão de informações importantes acerca dos requisitos (SOMMERVILLE, 2011; WIEGERS, 2003). Como consequência, o padrão pode ser usado como um guia para a verificação de requisitos. A Tabela 4.1 apresenta o padrão tabular sugerido neste texto. Sugerem-se agrupar requisitos de um mesmo tipo em diferentes tabelas. Assim, a informação do tipo do requisito não aparece explicitamente no padrão proposto. Além disso, informações complementares podem ser adicionadas em função do tipo de requisito. A seguir, discute-se como cada um dos itens da tabela pode ser tratado, segundo uma perspectiva geral. Na sequência, são tecidas considerações mais específicas sobre a especificação dos diferentes tipos de requisitos.

Tabela 4.1 – Tabela de Requisitos.

Identificador	Descrição	Origem	Prioridade	Responsável	Interessados	Dependências	Conflitos

Os requisitos devem possuir identificadores únicos para permitir a identificação e o rastreamento na gerência de requisitos. Há diversas propostas de esquemas de rotulagem de requisitos. Neste texto, recomenda-se usar um esquema de numeração sequencial por tipo de requisito, sendo usados os seguintes prefixos para designar os diferentes tipos de requisitos: *RF – requisitos funcionais*; *RNF – requisitos não funcionais*; *RN – regras de negócio*. Para outros esquemas de rotulagem, vide (WIEGERS, 2003). É importante destacar que, quando um requisito é eliminado, seu identificador não pode ser atribuído a outro requisito.

A descrição do requisito normalmente é feita na forma de uma sentença em linguagem natural. Ainda que expressa em linguagem natural, é importante adotar um estilo consistente e usar a terminologia do usuário ao invés do jargão típico da computação. Em relação ao estilo, recomenda-se utilizar sentenças em um dos seguintes formatos para descrever requisitos funcionais e não funcionais:

- *O sistema deve <verbo indicando ação, seguido de complemento>*: use o verbo *dever* para designar uma função ou característica requerida para o sistema, ou seja, para descrever um requisito obrigatório. Exemplos: O sistema deve efetuar o controle dos clientes da empresa. O sistema deve processar um pedido do cliente em um tempo inferior a cinco segundos, contado a partir da entrada de dados.
- *O sistema pode <verbo indicando ação, seguido de complemento>*: use o verbo *poder* para designar uma função ou característica desejável para o sistema, ou seja, para descrever um requisito desejável, mas não essencial. Exemplos: O sistema pode notificar usuários em débito. O sistema pode sugerir outros produtos para compra, com base em produtos colocados no carrinho de compras do usuário.

Em algumas situações, pode-se querer explicitar que alguma funcionalidade ou característica não deve ser tratada pelo sistema. Neste caso, indica-se uma sentença com a seguinte estrutura: *O sistema não deve <verbo indicando ação, seguido de complemento>*.

Wiegers (2003) recomenda diversas diretrizes para a redação de requisitos, dentre elas:

- Escreva frases completas, com a gramática, ortografia e pontuação correta. Procure manter frases e parágrafos curtos e diretos.
- Use os termos consistentemente. Defina-os em um glossário.
- Prefira a voz ativa (o sistema deve fazer alguma coisa) à voz passiva (alguma coisa deve ser feita).
- Sempre que possível, identifique o tipo de usuário. Evite descrições genéricas como "o usuário deve [...]". Se o usuário no caso for, por exemplo, o caixa do banco, indique claramente "o caixa do banco deve [...]".
- Evite termos vagos, que conduzam a requisitos ambíguos e não testáveis, tais como "rápido", "adequado", "fácil de usar" etc.
- Escreva requisitos em um nível consistente de detalhe.
- O nível de especificação de um requisito deve ser tal que, se o requisito é satisfeito, a necessidade do cliente é atendida. Contudo, evite restringir desnecessariamente o projeto (*design*).
- Escreva requisitos individualmente testáveis. Um requisito bem escrito deve permitir a definição de um pequeno conjunto de testes para verificar se o requisito foi corretamente implementado.
- Evite longos parágrafos narrativos que contenham múltiplos requisitos. Divida um requisito desta natureza em vários menores.

Conforme apontado anteriormente, requisitos devem ser testáveis. Robertson e Robertson (2006) sugerem três maneiras de tornar os requisitos testáveis:

- Especificar uma descrição quantitativa de cada advérbio ou adjetivo, de modo que o significado dos qualificadores fique claro e não ambíguo.
- Trocar os pronomes pelos nomes das entidades.
- Garantir que todo nome (termo) importante seja definido em um glossário no documento de requisitos.

A origem de um requisito deve apontar a partir de que entidade (pessoa, documento, atividade) o requisito foi identificado. Um requisito identificado durante uma investigação de documentos, p.ex., tem como origem o(s) documento(s) inspecionado(s). Já um requisito levantado em uma entrevista com um certo usuário tem como origem o próprio usuário. A informação de origem é importante para se conseguir a rastrear requisitos para a sua origem, prática muito recomendada no contexto da gerência de requisitos.

Requisitos podem ter importância relativa diferente uns em relação aos outros. Assim, é importante que o cliente e outros interessados estabeleçam conjuntamente a prioridade de cada requisito.

É muito importante saber quem é o analista responsável por um requisito, bem como quem são os interessados (clientes, usuários etc.) naquele requisito. São eles que estarão envolvidos nas discussões relativas ao requisito, incluindo a tentativa de acabar com conflitos e a definição de prioridades. Assim, deve-se registrar o nome e o papel do responsável e dos interessados em cada requisito.

Um requisito pode depender de outros ou conflitar com outros. Quando as dependências e conflitos forem detectados, devem-se listar os respectivos identificadores nas colunas de dependências e conflitos.

4.2 Escrevendo Requisitos Funcionais

As diretrizes apresentadas anteriormente aplicam-se integralmente a requisitos funcionais. Assim, não há outras diretrizes específicas para os requisitos funcionais. Deve-se realçar apenas que, quando expressos como requisitos de usuário, requisitos funcionais são geralmente descritos de forma abstrata, não cabendo neste momento entrar em detalhes. Detalhes vão ser naturalmente adicionados quando esses mesmos requisitos forem descritos na forma de requisitos de sistema.

Uma alternativa largamente empregada para especificar requisitos funcionais no nível de requisitos de sistema é a modelagem. Uma das técnicas mais comumente utilizadas para descrever requisitos funcionais como requisitos de sistema é a modelagem de casos de uso.

Vale ressaltar que os processos de negócio a serem apoiados pelo sistema tipicamente dão origem a requisitos funcionais. Assim, a partir de uma descrição de minimundo ou em um relatório proveniente de alguma atividade de levantamento de requisitos, podem ser encontrados requisitos funcionais a partir da identificação dos processos a serem apoiados. Além disso, o controle de informações que o negócio precisa gerenciar para apoiar os processos de negócio também deve dar origem a requisitos funcionais representando atividades custodiais (cadastros).

4.3 Escrevendo Requisitos Não Funcionais

Clientes e usuários naturalmente enfocam a especificação de requisitos funcionais. Entretanto para um sistema ser bem sucedido, é necessário mais do que entregar a funcionalidade correta. Usuários também têm expectativas sobre quão bem o sistema vai funcionar. Características que entram nessas expectativas incluem: quão fácil é usar o sistema, quão rapidamente ele roda, com que frequência ele falha e como ele trata condições inesperadas. Essas características, coletivamente conhecidas como atributos de qualidade do produto de software, são parte dos requisitos não funcionais do sistema. Essas expectativas de qualidade do produto têm de ser exploradas durante o levantamento de requisitos (WIEGERS, 2003).

Clientes geralmente não apontam suas expectativas de qualidade explicitamente. Contudo, informações providas por eles durante o levantamento de requisitos fornecem algumas pistas sobre o que eles têm em mente. Assim, é necessário definir com precisão o que os usuários pensam quando eles dizem que o sistema deve ser amigável, rápido, confiável ou robusto (WIEGERS, 2003).

Há muitos atributos de qualidade que podem ser importantes para um sistema. Uma boa estratégia para levantar requisitos não funcionais de produto consiste em explorar uma lista de potenciais atributos de qualidade que a grande maioria dos sistemas deve apresentar em algum nível. Por exemplo, o modelo de qualidade externa e interna de produtos de software definido na norma ISO/IEC 25010, utilizado como referência para a avaliação de produtos de software, define oito características de qualidade, desdobradas em subcaracterísticas, a saber (ISO/IEC, 2011):

- *Aptidão Funcional* (*Functional Suitability*): grau em que o produto provê funções que satisfazem às necessidades explícitas e implícitas, quando usado em condições especificadas. Inclui subcaracterísticas que evidenciam a existência de um conjunto de funções e suas propriedades específicas, a saber:
  - ✓ *Completude Funcional*: capacidade do produto de software de prover um conjunto apropriado de funções para tarefas e objetivos do usuário especificados. Refere-se às necessidades declaradas. Um produto no qual falte alguma função requerida não apresenta este atributo de qualidade.
  - ✓ *Correção Funcional (ou Acurácia):* grau em que o produto de software fornece resultados corretos e precisos, conforme acordado. Um produto que apresente dados incorretos, ou com a precisão abaixo dos limites definidos como toleráveis, não apresenta este atributo de qualidade.
  - ✓ *Adequação Funcional* (*Functional Appropriateness*): capacidade do produto de software de facilitar a realização das tarefas e objetivos do usuário. Refere-se às necessidades implícitas.
- *Confiabilidade:* grau em que o produto executa as funções especificadas com um comportamento consistente com o esperado, por um período de tempo. A confiabilidade está relacionada com os defeitos que um produto apresenta e como este produto se comporta em situações consideradas fora do normal. Suas subcaracterísticas são:
  - ✓ *Maturidade*: é uma medida da frequência com que o produto de software apresenta defeitos ao longo de um período estabelecido de tempo. Refere-se, portanto, à capacidade do produto de software de evitar falhas decorrentes de defeitos no software, mantendo sua operação normal ao longo do tempo.
  - ✓ *Disponibilidade*: capacidade do produto de software de estar operacional e acessível quando seu uso for requerido.
  - ✓ *Tolerância a falhas*: capacidade do produto de software de operar em um nível de desempenho especificado em casos de defeitos no software ou no hardware. Esta subcaracterística tem a ver com a forma como o software reage quando ocorre uma situação externa fora do normal (p.ex., conexão com a Internet interrompida).
  - ✓ *Recuperabilidade:* capacidade do produto de software de se colocar novamente em operação, restabelecendo seu nível de desempenho especificado, e recuperar os dados diretamente afetados no caso de uma falha.
- *Usabilidade:* grau em que o produto apresenta atributos que permitem que o mesmo seja entendido, aprendido e usado, e que o tornem atrativo para o usuário. Tem como subcaracterísticas:
  - ✓ *Reconhecimento da Adequação*: grau em que os usuários reconhecem que o produto de software é adequado para suas necessidades.
  - ✓ *Capacidade de Aprendizado*: refere-se à facilidade de o usuário entender os conceitos chave do produto e aprender a utilizá-lo, tornando-se competente em seu uso.

- ✓ *Operabilidade*: refere-se à facilidade do usuário operar e controlar o produto de software.
- ✓ *Proteção contra Erros do Usuário*: capacidade do produto de software de evitar que o usuário cometa erros.
- ✓ *Estética da Interface com o Usuário*: capacidade do produto de software de ser atraente ao usuário, lhe oferecendo uma interface com interação agradável.
- ✓ *Acessibilidade:* capacidade do produto de software ser utilizado por um amplo espectro de pessoas, incluindo portadores de necessidades especiais e com limitações associadas à idade.

- *Eficiência de Desempenho:* capacidade de o produto manter um nível de desempenho apropriado em relação aos recursos utilizados em condições estabelecidas. Inclui subcaracterísticas que evidenciam o relacionamento entre o nível de desempenho do software e a quantidade de recursos utilizados, a saber:
  - ✓ *Comportamento em Relação ao Tempo*: capacidade do produto de software de fornecer tempos de resposta e de processamento apropriados, quando o software executa suas funções, sob condições estabelecidas.
  - ✓ *Utilização de Recursos*: capacidade do produto de software de usar tipos e quantidades apropriados de recursos, quando executa suas funções, sob condições estabelecidas.
  - ✓ *Capacidade*: refere-se ao grau em que os limites máximos dos parâmetros do produto de software (p.ex., itens que podem ser armazenados, número de usuários concorrentes etc.) atendem às condições especificadas.
- *Segurança:* grau em que informações e dados são protegidos contra acesso por pessoas ou sistemas não autorizados, bem como grau em que essas informações e dados são disponibilizados para as pessoas ou sistemas com acesso autorizado. Tem como subcaracterísticas:
  - ✓ *Confidencialidade*: grau em que o produto ou sistema garante que os dados estão acessíveis apenas para aqueles autorizados a acessá-los.
  - ✓ *Integridade*: grau em que o sistema, produto ou componente evita acesso não autorizado a, ou modificação de, programas ou dados.
  - ✓ *Não Repúdio:* grau em que se pode provar que ações ou eventos aconteceram, de modo que eles não possam ser repudiados posteriormente.
  - ✓ *Responsabilização:* grau em que as ações de uma entidade (pessoa ou sistema) podem ser rastreadas unicamente a essa entidade.
  - ✓ *Autenticidade:* grau em que se pode provar que a identidade de um sujeito ou recurso é aquela reivindicada.
- *Compatibilidade:* capacidade do produto de software de trocar informações com outras aplicações e/ou compartilhar o mesmo ambiente de hardware ou software. Suas subcaracterísticas são:

- ✓ *Coexistência:* grau em que um produto pode desempenhar eficientemente suas funções requeridas, ao mesmo tempo em que compartilha um ambiente e recursos comuns com outros produtos, sem prejuízo para qualquer outro.
- ✓ *Interoperabilidade:* capacidade do produto de software de interagir com outros sistemas especificados, trocando e usando as informações trocadas.

- *Manutenibilidade:* capacidade do produto de software de ser modificado. Tem como subcaracterísticas:
  - ✓ *Modularidade:* grau em que um sistema ou programa é composto por componentes discretos (coesos e fracamente acoplados), de modo que mudanças em um componente tenham impacto mínimo sobre os outros.
  - ✓ *Reusabilidade:* capacidade dos componentes do produto de software serem utilizados na construção de outros componentes ou sistemas.
  - ✓ *Analisabilidade:* grau de eficácia e eficiência com que é possível: avaliar o impacto de uma alteração pretendida no produto ou sistema; diagnosticar deficiências ou causas de falhas no produto, ou identificar partes a serem modificadas.
  - ✓ *Modificabilidade:* grau em que um produto ou sistema pode ser eficaz e eficientemente modificado sem introduzir defeitos ou degradar sua qualidade.
  - ✓ *Testabilidade:* grau de eficácia e eficiência com que critérios de teste podem ser estabelecidos para um sistema, produto ou componente, e testes podem ser realizados para determinar se esses critérios foram satisfeitos.
- *Portabilidade:* refere-se à capacidade do software ser transferido de um ambiente de hardware, software ou operacional para outro. Tem como subcaracterísticas:
  - ✓ *Adaptabilidade:* grau em que um produto ou sistema pode ser eficaz e eficientemente adaptado para ambientes diferentes, ou em evolução, de hardware, software ou operacional.
  - ✓ *Capacidade para ser instalado (instalabilidade):* grau de eficácia e eficiência com que um produto ou sistema pode ser instalado e/ou desinstalado com sucesso em um ambiente especificado.
  - ✓ *Capacidade de substituição (substituibilidade):* grau em que um produto pode substituir outro produto especificado para a mesma finalidade, no mesmo ambiente. Considera, também, a facilidade de atualização para novas versões.

A característica de aptidão funcional e suas subcaracterísticas estão claramente relacionadas a requisitos funcionais. As demais, contudo, podem ser vistas como requisitos não funcionais que quaisquer produtos de software terão de tratar em alguma extensão.

Outro ponto importante a destacar é que diferentes autores listam diferentes características de qualidade, usando classificações próprias. Por exemplo, Bass, Clements e Kazman (2003) consideram, dentre outros, os seguintes atributos de qualidade:

- *Disponibilidade:* refere-se a falhas do sistema e suas consequências associadas. Uma falha ocorre quando o sistema não entrega mais um serviço consistente com sua especificação.
- *Modificabilidade:* diz respeito ao custo de modificação do sistema.
- *Desempenho:* refere-se a tempo.
- *Segurança:* está relacionada à habilidade do sistema impedir o uso não autorizado, enquanto ainda provê seus serviços para os usuários legítimos.
- *Testabilidade:* refere-se ao quão fácil é testar o software.
- *Usabilidade:* diz respeito a quão fácil é para o usuário realizar uma tarefa e o tipo de suporte ao usuário que o sistema provê.

Além das características de qualidade que se aplicam diretamente ao sistema, ditas características de qualidade de produto, Bass, Clements e Kazman (2003) listam outras características relacionadas a metas de negócio, dentre elas: tempo para chegar ao mercado (*time to market*), custo-benefício, tempo de vida projetado para o sistema, mercado alvo, cronograma de implementação e integração com sistemas legados.

Wiegers (2003), por sua vez, agrupa atributos de qualidade do produto em duas categorias principais: atributos importantes para os usuários e atributos importantes para os desenvolvedores. Como atributos importantes para os usuários são apontados os seguintes: disponibilidade, eficiência, flexibilidade, integridade, interoperabilidade, confiabilidade, robustez e usabilidade. Como atributos importantes para os desenvolvedores, são enumerados os seguintes: manutenibilidade, portabilidade, reusabilidade e testabilidade.

Uma vez que não há um consenso sobre quais atributos de qualidade considerar, cada organização deve definir as categorias de requisitos não funcionais a serem consideradas em seus projetos de software. Além disso, essa informação deve ser adicionada à tabela de requisitos não funcionais e, portanto, a Tabela 3.4, quando usada para descrever requisitos não funcionais, deve ter uma coluna adicional para indicar a categoria do requisito não funcional.

Em um mundo ideal, todo sistema deveria exibir os valores máximos para todos os atributos de qualidade. Contudo, como no mundo real isso não é possível, é fundamental definir quais atributos são mais importantes para o sucesso do projeto. Uma abordagem para tratar essa questão consiste em pedir para que diferentes representantes de usuários classifiquem cada atributo em uma escala de 1 (sem importância) a 5 (muito importante). As respostas ajudam o analista a determinar quais atributos são mais importantes. Obviamente, conflitos podem surgir e precisam ser resolvidos (WIEGERS, 2003).

Um aspecto a considerar é que diferentes partes do produto podem requerer diferentes combinações de atributos de qualidade. Assim, é importante também diferenciar características que se aplicam ao produto por inteiro daquelas que são necessárias para certas partes do sistema (WIEGERS, 2003). Para capturar essa diferenciação, é importante introduzir mais uma coluna na tabela de requisitos não funcionais, escopo, podendo assumir dois valores: funcionalidade, quando a característica se aplica apenas a algumas funcionalidades; e sistema, quando a característica se aplica ao sistema como um todo. Quando o escopo de um RNF for de funcionalidade, os requisitos funcionais que precisam incorporar essa característica de qualidade deverão incluir em sua coluna de dependências o identificador desse RNF.

Uma vez priorizados os atributos de qualidade, o analista deve passar, então, a trabalhar com os usuários no sentido de especificar, para cada atributo considerado importante, requisitos mensuráveis e, por conseguinte, testáveis. Se os atributos de qualidade são especificados de maneira não passível de verificação, não é possível dizer posteriormente se eles foram atingidos ou não. Quando apropriado, devem-se indicar escalas ou unidades de medida para cada atributo e os valores mínimo, alvo e máximo. Se não for possível quantificar todos os atributos importantes, deve-se, pelo menos, definir suas prioridades. Pode-se, ainda, perguntar aos usuários o que constituiria um valor inaceitável para cada atributo e definir testes que tentem forçar o sistema a demonstrar tais características (WIEGERS, 2003). É importante destacar, contudo, que essa especificação mais detalhada dos RNFs não deve ser feita no levantamento inicial de requisitos. Ao contrário, ela é considerada como parte da análise de requisitos (corresponde à especificação dos RNFs) e muitas vezes, é até postergada para a fase de projeto (ou transição para a fase de projeto), uma vez que RNFs guardam estreita relação com aspectos tecnológicos, os quais serão tratados na fase de projeto.

Robertson e Robertson (2006) sugerem definir critérios de adequação ou ajuste (*fit criteria*) para permitir quantificar requisitos (tanto funcionais como não funcionais) e associá-los à descrição dos requisitos. À primeira vista, alguns requisitos não funcionais podem parecer difíceis de quantificar. Entretanto, deve ser possível atribuir números a eles. Se não se consegue quantificar e medir um requisito, então é provável que o requisito não seja de fato um requisito. Ele pode ser, por exemplo, vários requisitos descritos em um só.

Seja a seguinte situação. No desenvolvimento de um sistema para uma biblioteca, o usuário coloca o seguinte requisito não funcional: "O sistema deve ser amigável ao usuário". Esse requisito é vago, ambíguo e não passível de ser expresso em números. Como quantificar se o sistema é "amigável ao usuário"? Primeiro, é preciso entender o que o usuário quer dizer com "amigável ao usuário". Significa ser fácil de compreender? Fácil de aprender? Fácil de operar? Atrativo? Clareando a intenção do usuário, é possível sugerir uma escala de medição. Suponha que o usuário diga que ser "amigável ao usuário" significa que os usuários serão capazes de aprender rapidamente a usar o sistema. Uma vez definido que se está falando sobre facilidade de aprender (inteligibilidade), é possível definir como escala de medição o tempo gasto para dominar a execução de certas tarefas. A partir disso, pode-se estabelecer como critério de aceitação o seguinte: "Novos bibliotecários devem ser capazes de efetuar empréstimos após a quarta tentativa de realizar essa tarefa usando o sistema".

Determinar critérios de aceitação ajuda a clarear um requisito. Ao se estabelecer uma escala de medição e os valores aceitáveis, o requisito é transformado de uma intenção vaga, e até certo ponto ambígua, em um requisito mensurável e bem formado. Estabelecida uma escala, pode-se perguntar ao usuário o que ele considera uma falha em atender ao requisito, de modo a definir o critério de aceitação. Contudo, pode ser difícil, senão impossível, obter um requisito completo e mensurável em primeira instância (ROBERTSON; ROBERTSON, 2006). Assim, na descrição de requisitos de cliente é suficiente capturar a intenção e depois, na especificação de requisitos de sistema, transformar essa intenção em um requisito mensurável, adicionando a ele um critério de aceitação. É muito comum que, neste processo, um requisito não funcional de usuário dê origem a vários requisitos não funcionais de sistema.

4.4 Escrevendo Regras de Negócio

Toda organização opera de acordo com um extenso conjunto de políticas corporativas, leis, padrões industriais e regulamentações governamentais. Tais princípios de controle são

coletivamente designados por regras de negócio. Uma regra de negócio é uma declaração que define ou restringe algum aspecto do negócio, com o propósito de estabelecer sua estrutura ou controlar ou influenciar o comportamento do negócio (WIEGERS, 2003).

Sistemas de informação tipicamente precisam fazer cumprir as regras de negócio. Ao contrário dos requisitos funcionais e não funcionais, a maioria das regras de negócio origina-se fora do contexto de um sistema específico. Assim, as regras a serem tratadas pelo sistema precisam ser identificadas, documentadas e associadas aos requisitos do sistema em questão (WIEGERS, 2003).

Wiegers identifica cinco tipos principais de regras de negócio, cada um deles apresentando uma forma típica de ser escrito:

- *Fatos ou invariantes:* declarações que são verdade sobre o negócio. Geralmente descrevem associações ou relacionamentos entre importantes termos do negócio. Ex.: Todo pedido tem uma taxa de remessa.
- *Restrições:* como o próprio nome indica, restringem as ações que o sistema ou seus usuários podem realizar. Algumas palavras ou frases sugerem a descrição de uma restrição, tais como *deve*, *não deve*, *não pode* e *somente*. Ex.: Um aluno só pode tomar emprestado, concomitantemente, de um a três livros.
- *Ativadores de Ações:* são regras que disparam alguma ação sob condições específicas. Uma declaração na forma "Se <alguma condição é verdadeira ou algum evento ocorre>, então <algo acontece>" é indicada para descrever ativadores de ações. Ex.: Se a data para retirada do livro é ultrapassada e o livro não é retirado, então a reserva é cancelada. Quando as condições que levam às ações são uma complexa combinação de múltiplas condições individuais, então o uso de tabelas de decisão ou árvores de decisão é indicado. As figuras 4.1 e 4.2 ilustram uma mesma regra de ativação de ações descrita por meio de uma árvore de decisão e de uma tabela de decisão, respectivamente.

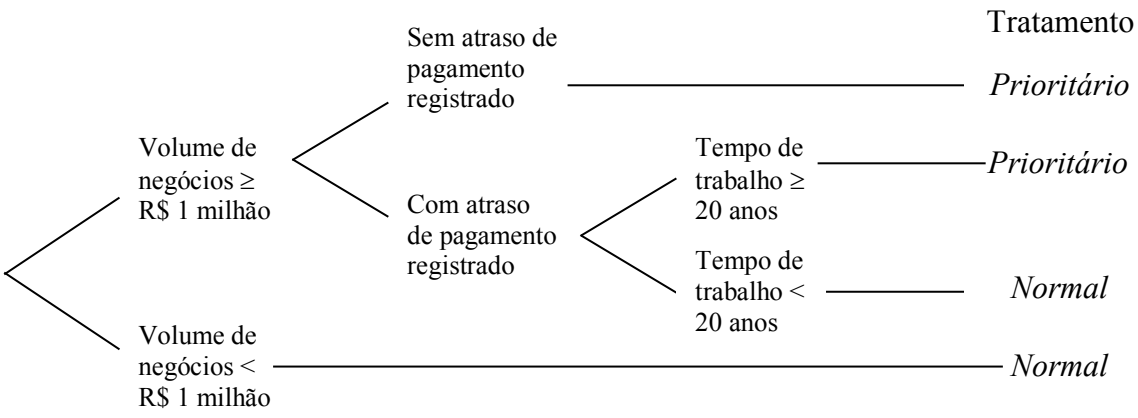

Figura 4.1 – Exemplo de Árvore de Decisão.

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

Figura 4.2 – Exemplo de Tabela de Decisão.

- *Inferências:* são regras que derivam novos fatos a partir de outros fatos ou cálculos. São normalmente escritas no padrão "se / então", como as regras ativadoras de ação, mas a cláusula então implica um fato ou nova informação e não uma ação a ser tomada. Ex.: Se o usuário não devolve um livro dentro do prazo estabelecido, então ele torna-se um usuário inadimplente.
- *Computações:* são regras de negócio que definem cálculos a serem realizados usando algoritmos ou fórmulas matemáticas específicos. Podem ser expressas como fórmulas matemáticas, descrição textual, tabelas, etc. Ex.: Aplica-se um desconto progressivo se mais do que 10 unidades forem adquiridas. De 10 a 19 unidades, o desconto é de 10%. A compra de 20 ou mais unidades tem um desconto de 25%. A Figura 4.3 mostra essa mesma regra expressa no formato de uma tabela.

Número de Itens Adquiridos	**Percentual de Desconto**
1 a 9	0
10 a 19	10%
20 ou mais	25%

Figura 4.3 – Exemplo de Regra de Cálculo.

Ao contrário de requisitos funcionais e não funcionais, regras de negócio não são passíveis de serem capturadas por meio de perguntas simples e diretas, tal como "Quais são suas regras de negócio?". Regras de negócio emergem durante a discussão de requisitos, sobretudo quando se procura entender a base lógica por detrás de requisitos e restrições apontados pelos interessados (WIEGERS, 2003). Assim, não se deve pensar que será possível levantar muitas regras de negócio em um levantamento preliminar de requisitos. Pelo contrário, as regras de negócio vão surgir principalmente durante o levantamento detalhado dos requisitos. Wiegers (2003) aponta diversas potenciais origens para regras de negócio e sugere tipos de questões que o analista pode fazer para tentar capturar regras advindas dessas origens:

- *Políticas:* Por que é necessário fazer isso desse jeito?
- *Regulamentações:* O que o governo requer?
- *Fórmulas:* Como este valor é calculado?
- *Modelos de Dados:* Como essas entidades de dados estão relacionadas?
- *Ciclo de Vida de Objetos:* O que causa uma mudança no estado desse objeto?
- *Decisões de Atores:* O que o usuário pode fazer a seguir?

- *Decisões de Sistema:* Como o sistema sabe o que fazer a seguir?
- *Eventos:* O que pode (e não pode) acontecer?

Regras de negócio normalmente têm estreita relação com requisitos funcionais. Uma regra de negócio pode ser tratada no contexto de uma certa funcionalidade. Neste caso, a regra de negócio deve ser listada na coluna de dependências do requisito funcional (vide Tabela 5.1). Há casos em que uma regra de negócio conduz a um requisito funcional para fazer cumprir a regra. Neste caso, a regra de negócio é considerada a origem do requisito funcional (WIEGERS, 2003).

É importante destacar a importância das regras (restrições) obtidas a partir de modelos de dados, ditas *restrições de integridade*. Elas complementam as informações do modelo de dados e capturam restrições entre elementos de um modelo de dados que não são passíveis de serem capturadas pelas notações gráficas utilizadas em modelos desse tipo. Tais regras devem ser documentadas junto com os modelos de dados. Seja o exemplo do modelo de dados da Figura 3.10. Esse fragmento de modelo indica que: (i) um aluno cursa um curso; (ii) um aluno pode se matricular em nenhuma ou várias turmas; (iii) um curso possui um conjunto de disciplinas em sua matriz curricular; (iv) uma turma é de uma disciplina específica. Contudo, nada diz sobre restrições entre o estabelecimento dessas várias relações. Suponha que o negócio indique que a seguinte restrição deve ser considerada: Um aluno só pode ser matricular em turmas de disciplinas que compõem a grade curricular do curso que esse aluno cursa. Essa restrição tem de ser escrita para complementar o modelo.

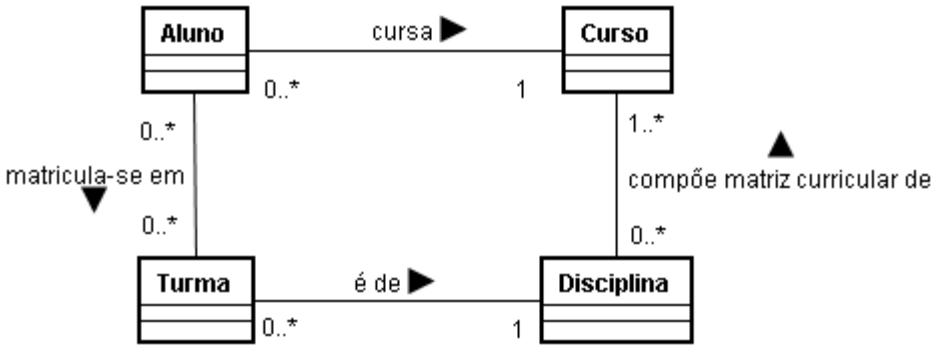

Figura 4.4 – Exemplo de Fragmento de Modelo de Dados com Restrição de Integridade.

Outro tipo de restrição importante são as regras que impõem restrições sobre funcionalidades de inserção, atualização ou exclusão de dados, devido a relacionamentos de existentes entre as entidades. Voltando ao exemplo da Figura 3.10, a exclusão de disciplinas não deve ser livre, uma vez que turmas são existencialmente dependentes de disciplinas. Assim, a seguinte regra de integridade referencial de dados deve ser considerada: Ao excluir uma disciplina, devem-se excluir todas as turmas a ela relacionadas. Essas regras são denominadas neste texto como *restrições de processamento* e, uma vez que dizem respeito a funcionalidades, devem ser documentadas junto com a descrição do caso de uso que detalha a respectiva funcionalidade.

Referências do Capítulo

AURUM, A., WOHLIN, C., *Engineering and Managing Software Requirements*, Springer-Verlag, 2005.

BASS, L., CLEMENTS, P., KAZMAN, R., *Software Architecture in Practice*, Second edition, Addison Wesley, 2003.

ISO/IEC 25010, System and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.

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

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

ROBERTSON, S., ROBERTSON, J. *Mastering the Requirements Process***.** 2nd Edition. Addison Wesley, 2006.

SOMMERVILLE, I., *Engenharia de Software*, 8ª Edição. São Paulo: Pearson – Addison Wesley, 2007.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.

WIEGERS, K.E., Software Requirements: Practical techniques for gathering and managing requirements throughout the product development cycle. 2nd Edition, Microsoft Press, Redmond, Washington, 2003.

Capítulo 5 – Análise de Requisitos

A investigação do domínio do problema e do negócio a ser apoiado nos leva a perceber que o desenvolvimento de um novo sistema de informação é um agente de mudanças no negócio e na prática dos profissionais envolvidos. Podemos pensar que estamos diante de duas versões de um sistema: o sistema como ele é (*as-is*) e o sistema que se pretende obter (*to-be*). Note que quando falamos do sistema neste contexto não estamos falando de um sistema computacional, mas sim de sistema em uma acepção mais geral da palavra: um conjunto intelectualmente organizado de elementos (pessoas, produtos de software, dispositivos etc.). O sistema *as-is* apresenta problemas, limitações e deficiências. O sistema *to-be* tem a intenção de tratar esses problemas usando oportunidades oferecidas pela tecnologia. Assim, para que o sistema *to-be* seja bem-sucedido, o sistema de software a ser desenvolvido (software *to-be*) precisa cooperar efetivamente com seu ambiente (pessoas, dispositivos e outros sistemas de software existentes). Assim, podemos estruturar o trabalho de ER em termos de três dimensões, como mostra a Figura 5.1 (LAMSWEERDE, 2009):

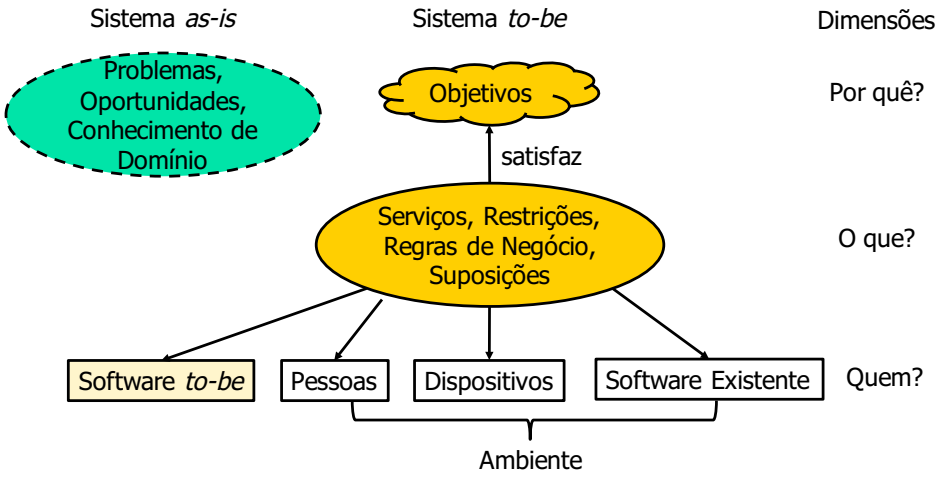

Figura 5.1 – As dimensões da Engenharia de Requisitos (adaptado de (LAMSWEERDE, 2009)).

- **Por quê?** As razões para se desenvolver um novo sistema computacional (software *to-be*) devem ser explicitadas em termos de objetivos a serem satisfeitos.
- **O que?** Esta dimensão preocupa-se com os serviços (requisitos funcionais) que o novo sistema computacional deve prover para satisfazer os objetivos. Esses serviços devem satisfazer restrições (requisitos não funcionais), garantir regras de negócio e apoiam-se em suposições.
- **Quem?** Alguns serviços serão implementados pelo software *to-be*, enquanto outros serão realizados por procedimentos manuais, operações de dispositivos ou por outros produtos de software já existentes.

Para responder a essas perguntas, é necessário um entendimento mais profundo do domínio do problema, do negócio a ser apoiado, das limitações do sistema atual e dos requisitos para o novo sistema. Requisitos de cliente são insuficientes para isso. Eles resultam diretamente da atividade de levantamento de requisitos, sendo tipicamente descritos em linguagem natural, em um baixo nível de detalhes. Assim, uma vez identificados os requisitos de cliente, é

necessário detalhá-los, colocando-os no nível de descrição de requisitos de sistema. Este é o propósito da Análise de Requisitos.

Requisitos de sistema resultam dos esforços dos analistas de organizar e detalhar requisitos de cliente, envolvendo a elaboração de um conjunto de modelos abstratos do sistema, agora em um nível alto de detalhes. A correta derivação de requisitos de sistema a partir de requisitos de cliente é fundamental para assegurar que equívocos não serão arbitrariamente introduzidos pelos engenheiros de requisitos na especificação de requisitos (AURUM; WOHLIN, 2005). Durante a análise de requisitos, requisitos funcionais e não funcionais devem ser expressos em um nível de detalhes que permita guiar as etapas subsequentes do desenvolvimento de software (projeto, implementação e testes). Além disso, atenção especial deve ser dada às regras de negócio. Elas têm de ser capturadas e incorporadas às funcionalidades do sistema. Neste contexto, vale destacar que a aplicação de técnicas de levantamento de requisitos é imprescindível e, portanto, o levantamento detalhado de requisitos ocorre em paralelo com a análise de requisitos.

Para descrever requisitos funcionais detalhadamente, é necessário produzir modelos, cada um deles capturando uma perspectiva diferente do sistema. A linguagem natural ainda é utilizada, mas em uma escala reduzida, circunscrita a descrições de certos modelos ou às definições em glossários ou dicionários de dados. Grande parte das informações tratadas na análise de requisitos funcionais é melhor comunicada por meio de diagramas do que por meio de texto. Assim, a modelagem é uma atividade essencial da análise de requisitos.

A modelagem de objetivos pode ser empregada para tratar a dimensão "Por quê?". Um modelo de objetivo procura capturar precisamente os objetivos a serem satisfeitos pelo sistema *to-be*, suas ramificações em termos de subobjetivos, como esses objetivos interagem (conflitos e sinergias) e como eles estão alinhados a objetivos de negócio (LAMSWEERDE, 2009).

A modelagem conceitual visa definir em detalhes as funções requeridas pelo sistema e o conhecimento necessário para realizá-las. O produto principal da modelagem conceitual é o esquema conceitual do sistema (OLIVÉ, 2007). O esquema ou modelo conceitual[2] de um sistema captura as funções e informações que o sistema deve prover e gerenciar. Ele deve ser concebido com foco no domínio do problema e não no domínio da solução, mas deve tratar tanto uma visão externa do sistema (como o sistema é percebido pelos usuários) quanto uma visão interna do mesmo (como as abstrações do domínio são representadas e relacionadas).

Os termos análise de sistemas e análise de requisitos são muitas vezes empregados para designar as atividades de modelagem conceitual (análise de requisitos funcionais). Assim, a maioria dos métodos de análise de sistemas concentra-se na análise de requisitos funcionais, nada falando sobre a análise de requisitos não funcionais. Entretanto, durante a atividade de análise de requisitos, tanto requisitos funcionais quanto requisitos não funcionais devem ser especificados em detalhes.

O produto de trabalho principal da análise de requisitos é o Documento de Especificação de Requisitos. Esse documento deve conter os requisitos funcionais e não funcionais descritos

[2] Olivé (2007) utiliza o termo "esquema conceitual" para designar o conjunto de artefatos que capturam o conhecimento que um sistema de informação necessita para realizar suas funções. Contudo, a maioria dos textos, tais como (WAZLAWICK, 2004) e (BLAHA; RUMBAUGH, 2006), utiliza o termo "modelo conceitual" para designar esse conjunto de artefatos. Olivé utiliza o termo "modelo conceitual" para designar tipos de modelos, tais como modelos de objetos, modelos de casos de uso etc. Nestas notas de aula, o termo "modelo conceitual" é usado como na maioria dos textos, não sendo feita uma distinção precisa entre as duas acepções do termo. De maneira geral, o contexto indica a que se refere o termo.

em nível de requisitos de sistema. Conforme citado anteriormente, os requisitos funcionais de sistema são descritos por um conjunto de modelos inter-relacionados. Os requisitos não funcionais de sistema detalham os requisitos não funcionais de usuário, adicionando a eles critérios de aceitação.

É importante apontar que há uma forte dependência entre os métodos e técnicas usados na modelagem conceitual e o paradigma de desenvolvimento adotado. Isso ocorre porque um modelo é uma representação do sistema segundo um particular metamodelo. Esse metamodelo corresponde ao conjunto de elementos de modelagem (estruturais e comportamentais) e regras de uso desses elementos, o qual permite construir modelos segundo o respectivo paradigma (AURUM; WOHLIN, 2005). Assim, para a modelagem conceitual de requisitos funcionais é necessário escolher um paradigma de desenvolvimento (e o correspondente metamodelo subjacente a ele) a partir do qual os modelos serão construídos. O paradigma orientado a objetos, por exemplo, fornece um conjunto de elementos de modelagem que permite modelar um sistema como sendo composto de objetos organizados em classes que se comunicam entre si por meio de troca de mensagens. Uma classe define as propriedades (atributos, relacionamentos e operações) que todos os objetos dela podem possuir. Assim, classes, objetos, associações, atributos e operações, dentre outros, são elementos do metamodelo subjacente ao paradigma orientado a objetos. Neste texto, o paradigma adotado é o da orientação a objetos.

Este capítulo é bastante extenso e está assim organizado: a seção 5.1 introduz a linguagem de modelagem UML, que será usada para a criação dos modelos durante a análise de requisitos; a seção 5.2 trata de aspectos fundamentais do paradigma orientado a objetos, necessários para a condução da análise de requisitos segundo esse paradigma; a seção 5.3 apresenta um método para a análise de requisitos funcionais; a seção 5.4 aborda requisitos não funcionais no contexto da análise; a seção 5.5 trata do Documento de Especificação de Requisitos; as seções 5.6, 5.7 e 5.8 tratam das atividades de modelagem que são realizadas no âmbito da análise de requisitos, sendo, respectivamente: modelagem de casos de uso, modelagem conceitual estrutural e modelagem dinâmica.

5.1 A Linguagem de Modelagem Unificada

A Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML) é uma linguagem gráfica padrão para especificar, visualizar, documentar e construir artefatos de sistemas de software (BOOCH; RUMBAUGH; JACOBSON, 2006).

A UML foi criada na década de 1990, a partir de uma tentativa de se unificar dois dos principais métodos orientados a objetos utilizados até então: a Técnica de Modelagem de Objetos (*Object Modeling Technique* – OMT) (RUMBAUGH et al., 1994) e o Método de Booch (BOOCH, 1994). Inicialmente, buscava-se criar um método unificado. A este esforço juntou-se também Ivar Jacobson, fundindo também seu método OOSE (JACOBSON, 1992). Contudo, percebeu-se que não era possível estabelecer um único método adequado para todo e qualquer projeto de desenvolvimento. De fato, um método é composto por uma linguagem estabelecendo a notação a ser usada na elaboração dos artefatos a serem produzidos e de um processo descrevendo que artefatos construir e como construí-los. A linguagem pode ser unificada, mas a decisão de quais artefatos produzir e que passos seguir não é passível de padronização, já que pode variar de projeto para projeto. Assim, ao invés de criarem um método unificado, Rumbaugh, Booch e Jacobson propuseram a UML, incorporando as principais notações para os artefatos de seus métodos e de vários outros, com a colaboração de várias empresas e autores. A UML foi aprovada em novembro de 1997 pelo *Object Management*

Group (OMG) pondo fim a uma guerra de métodos orientados a objeto. Sua versão mais recente, a UML 2.5, foi publicada 2015 e é resultado de um esforço de diversos colaboradores, envolvendo empresas e pesquisadores sob a coordenação do OMG.

Vale realçar que a UML é somente uma linguagem e, portanto, é apenas parte de um método de desenvolvimento de software. Ela é independente do processo de software a ser usado, ainda que seja mais adequada a processos de desenvolvimento orientados a objetos. Ela se destina a visualizar, especificar, construir e documentar artefatos de software (BOOCH; RUMBAUGH; JACOBSON, 2006). No contexto da Engenharia de Requisitos, a UML provê diversos diagramas que podem ser usados na modelagem de requisitos, tanto segundo a perspectiva estrutural quanto segundo a perspectiva comportamental. Para a modelagem conceitual estrutural, os diagramas de classes são amplamente utilizados. Para a modelagem comportamental, podem ser usados diagramas de casos de uso, de interação, de estados e de atividades. A seguir, é apresentada uma descrição sucinta de cada um desses diagramas, baseada em (BOOCH; RUMBAUGH; JACOBSON, 2006):

- *Diagrama de Classes:* modela um conjunto de classes e seus relacionamentos. Diagramas de classes proveem uma visão estática da estrutura de um sistema e, portanto, são usados na modelagem conceitual estrutural.
- *Diagrama de Casos de Uso:* mostra um conjunto de casos de uso e atores e seus relacionamentos. Os casos de uso descrevem a funcionalidade do sistema percebida pelos atores externos. Um ator interage com o sistema, podendo ser um usuário humano, dispositivo de hardware ou outro sistema. Diagramas de casos de uso proveem uma visão das funcionalidades do sistema.
- *Diagrama de Gráfico de Estados* (ou simplesmente Diagrama de Estados): mostra os estados pelos quais os objetos de uma classe específica podem passar ao longo de suas vidas, em resposta a estímulos recebidos, juntamente com suas ações. Os diagramas de estados proveem uma visão dinâmica dos objetos de uma classe, sendo importantes para modelar o comportamento de objetos de uma classe em resposta à ocorrência de eventos.
- *Diagrama de Atividades*: mostra a estrutura de um processo. Provê uma visão dinâmica do sistema (ou de uma porção do sistema) e pode ser usado tanto para modelar processos de negócio quanto para modelar funções do sistema. Um diagrama de atividades dá ênfase ao fluxo de controle entre objetos.
- *Diagrama de Interação*: mostra um conjunto de objetos interagindo, incluindo as mensagens que podem ser trocadas entre eles. Provê uma visão dinâmica do comportamento de um sistema ou de uma porção do sistema. Há dois tipos de diagramas de interação:
- *Diagrama de Sequência*: dá ênfase à ordenação temporal das mensagens.
- *Diagrama de Comunicação*: tem o mesmo propósito do diagrama de sequência, apresentando, contudo, ênfase na organização estrutural dos objetos que enviam ou recebem mensagens.

No contexto da Engenharia de Requisitos, além dos diagramas anteriormente citados, merecem atenção também os *diagramas de pacotes*. Na UML, pacote é um mecanismo de propósito geral usado para organizar elementos de modelagem em grupos. Assim, um diagrama

de pacotes mostra a decomposição de um modelo em unidades menores e suas dependências (BOOCH; RUMBAUGH; JACOBSON, 2006).

Vale ressaltar mais uma vez, que a UML é uma linguagem de modelagem, não um método de desenvolvimento OO. Os métodos consistem, pelo menos em princípio, de uma linguagem de modelagem e um processo de uso dessa linguagem. A UML não prescreve esse processo de utilização e, portanto, deve ser aplicada no contexto de um processo, lembrando que projetos diferentes podem requerer processos diferentes. Sendo assim, na próxima seção é apresentado um método geral de análise de requisitos funcionais que aponta quais diagramas da UML adotar e um processo de elaboração dos mesmos.

5.2 O Paradigma Orientado a Objetos

Um dos paradigmas de desenvolvimento mais adotados atualmente é a orientação a objetos. Segundo esse paradigma, o mundo é visto como sendo composto por *objetos*, onde um objeto é uma entidade que combina estrutura de dados e comportamento funcional. No paradigma orientado a objetos, os sistemas são modelados como um número de objetos que interagem.

A orientação a objetos oferece um número de conceitos bastante apropriados para a modelagem de sistemas. Os modelos baseados em objetos são úteis para a compreensão de problemas, para a comunicação com os especialistas e usuários das aplicações, e para a realização das tarefas ao longo do processo de desenvolvimento de software. Em um sistema construído segundo o paradigma orientado a objetos (OO), componentes são partes encapsuladas de dados e funções, que podem herdar atributos e comportamento de outros componentes da mesma natureza e cujos componentes comunicam-se entre si por meio de mensagens (YOURDON, 1994). O paradigma OO utiliza uma perspectiva humana de observação da realidade, incluindo objetos, classificação e compreensão hierárquica.

Para fazer bom uso do paradigma OO, é importante conhecer os princípios adotados por ele na administração da complexidade, bem como seus principais conceitos.

5.2.1 Princípios para Administração da Complexidade

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

a) Abstração

Uma das principais formas do ser humano lidar com a complexidade é através do uso de abstrações. As pessoas tipicamente tentam compreender o mundo, construindo modelos mentais de partes dele. Tais modelos são uma visão simplificada de algo, onde apenas os elementos relevantes são considerados. Modelos, portanto, são mais simples do que os complexos sistemas que eles modelam.

Seja o exemplo de um mapa representando um modelo de um território. Um mapa é útil porque abstrai apenas as características do território que se deseja modelar. Se um mapa

incluísse todos os detalhes do território, provavelmente teria o mesmo tamanho do território e, portanto, não serviria a seu propósito.

Da mesma forma que um mapa precisa ser significativamente menor que o território que mapeia, incluindo apenas as informações selecionadas, um modelo deve abstrair apenas as características relevantes de um sistema para seu entendimento. Assim, pode-se definir abstração como sendo o princípio de ignorar aspectos não relevantes de um assunto, segundo a perspectiva de um observador, tornando possível uma concentração maior nos aspectos principais do mesmo. De fato, a abstração consiste na seleção que um observador faz de alguns aspectos de um assunto, em detrimento de outros que não demonstram ser relevantes para o propósito em questão. A Figura 5.1 ilustra o conceito de abstração.

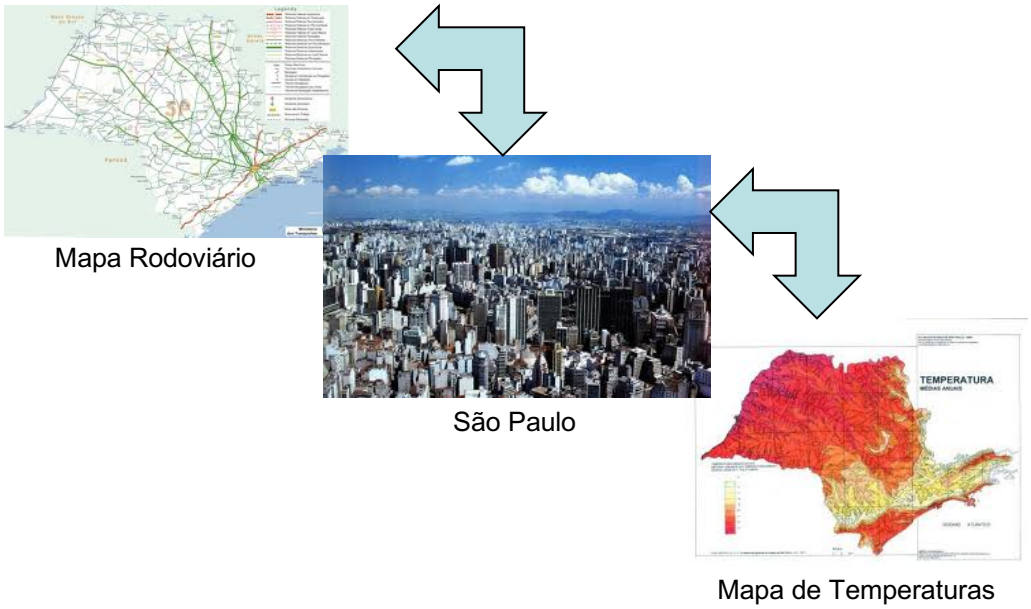

Figura 5.1 – Ilustração do Conceito de Abstração.

Um conjunto de abstrações pode formar uma hierarquia e, pela identificação dessas hierarquias, é possível simplificar significativamente o entendimento sobre um problema (BOOCH, 1994). Assim, hierarquia é uma forma interessante de arrumar as abstrações.

b) Encapsulamento

No mundo real, um objeto pode interagir com outro sem conhecer seu funcionamento interno. Uma pessoa, por exemplo, utiliza um forno de microondas sem saber efetivamente qual a sua estrutura interna ou como seus mecanismos internos são ativados. Para utilizá-lo, basta saber usar seu painel de controle (a interface do aparelho) para realizar as operações de ligar/desligar, informar o tempo de cozimento etc. Como essas operações produzem os resultados, não interessa ao cozinheiro. A Figura 5.2 ilustra o conceito de encapsulamento.

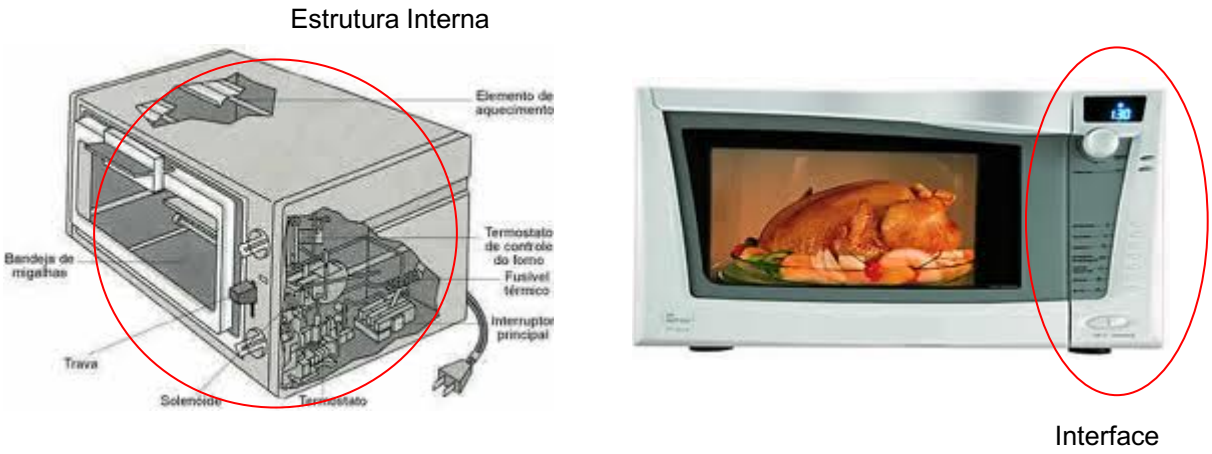

Figura 5.2 – Ilustração do Conceito de Encapsulamento.

O encapsulamento consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, de seus detalhes internos de implementação, que ficam ocultos dos demais objetos (BLAHA; RUMBAUGH, 2006). A interface de um objeto deve ser definida de forma a revelar o menos possível sobre o seu funcionamento interno.

Abstração e encapsulamento são conceitos complementares: enquanto a abstração enfoca o comportamento observável de um objeto, o encapsulamento oculta a implementação que origina esse comportamento. Encapsulamento é frequentemente conseguido através da ocultação de informação, isto é, escondendo detalhes que não contribuem para suas características essenciais. Tipicamente, em um sistema orientado a objetos, a estrutura de um objeto e a implementação de seus métodos são encapsuladas (BOOCH, 1994). Assim, o encapsulamento serve para separar a interface contratual de uma abstração e sua implementação. Os usuários têm conhecimento apenas das operações que podem ser requisitadas e precisam estar cientes apenas do *quê* as operações realizam e não *como* elas estão implementadas.

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

c) Modularidade

Os métodos de desenvolvimento de software buscam obter sistemas modulares, isto é, construídos a partir de elementos que sejam autônomos e conectados por uma estrutura simples e coerente. Modularidade visa à obtenção de sistemas decompostos em um conjunto de módulos coesos e fracamente acoplados e é crucial para se obter reusabilidade e facilidade de extensão.

Abstração, encapsulamento e modularidade são princípios sinergéticos, isto é, ao se trabalhar bem com um deles, estão-se aperfeiçoando os outros também.

5.2.2 Principais Conceitos da Orientação a Objetos

De maneira simples, o paradigma OO traz uma visão de mundo em que os fenômenos e domínios são vistos como coleções de objetos interagindo entre si. Essa forma simples de se colocar a visão do paradigma OO esconde conceitos importantes da orientação a objetos que são a base para o desenvolvimento OO, tais como classes, associações, generalização etc. A seguir os principais conceitos da orientação a objetos são discutidos.

a) Objetos

O mundo real é povoado por entidades que interagem entre si, onde cada uma delas desempenha um papel específico. Na orientação a objetos, essas entidades são ditas *objetos*. Objetos podem ser coisas concretas ou abstratas, tais como um carro, uma reserva de passagem aérea, uma organização etc.

Do ponto de vista da modelagem de sistemas, um objeto é uma entidade que incorpora uma abstração relevante no contexto de uma aplicação. Um objeto possui um estado (informação), exibe um comportamento bem definido (dado por um número de operações para examinar ou alterar seu estado) e tem identidade única.

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

A abstração incorporada por um objeto é caracterizada por um conjunto de serviços ou operações que outros objetos, ditos clientes, podem requisitar. Operações são usadas para recuperar ou manipular a informação de estado de um objeto e se referem apenas às estruturas de dados do próprio objeto, não devendo acessar diretamente estruturas de outros objetos. Caso a informação necessária para a realização de uma operação não esteja disponível, o objeto terá de colaborar com outros objetos.

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

Cada objeto tem uma identidade própria, que lhe é inerente. Todos os objetos têm existência própria, ou seja, dois objetos são distintos, mesmo se seus estados e comportamentos forem iguais. A identidade de um objeto transcende os valores correntes de suas propriedades.

b) Classes

No mundo real, diferentes objetos desempenham um mesmo papel. Seja o caso de duas cadeiras. Apesar de serem objetos diferentes, elas compartilham uma mesma estrutura e um mesmo comportamento. Entretanto, não há necessidade de se despender tempo modelando cada cadeira. Basta definir, em um único lugar, um modelo descrevendo a estrutura e o comportamento desses objetos. A esse modelo dá-se o nome de *classe*. Uma classe descreve um conjunto de objetos com as mesmas propriedades (atributos e associações), o mesmo

comportamento (operações) e a mesma semântica. Objetos que se comportam da maneira especificada pela classe são ditos *instâncias* dessa classe.

Todo objeto pertence a uma classe, ou seja, é instância de uma classe. De fato, a orientação a objetos norteia o processo de desenvolvimento através da *classificação de objetos*, isto é, objetos são agrupados em classes, em função de exibirem facetas similares, sem, no entanto, perda de sua individualidade, como ilustra a Figura 5.3. Assim, do ponto de vista estrutural, a modelagem orientada a objetos consiste, basicamente, na definição de classes. O comportamento e a estrutura de informação de uma instância são definidos pela sua classe.

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

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

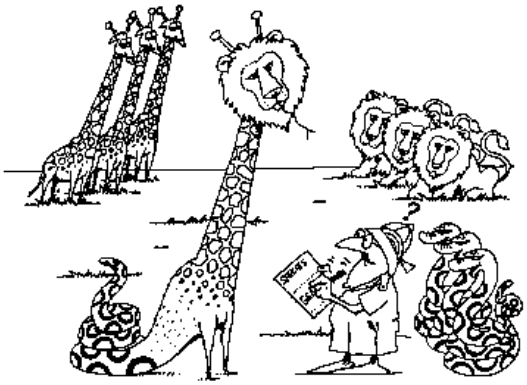

Figura 5.3 – Ilustração do conceito de Classificação (BOOCH, 1994).

c) Ligações e Associações

Em qualquer sistema, objetos relacionam-se uns com os outros. Por exemplo, em "*o empregado João trabalha no Departamento de Pessoal*", temos um relacionamento entre o objeto *empregado João* e o objeto *Departamento de Pessoal*.

Ligações e associações são meios de se representar relacionamentos entre objetos e entre classes, respectivamente. Uma ligação é uma conexão entre objetos. No exemplo anterior, há uma ligação entre os objetos *João* e *Departamento de Pessoal*. Uma associação, por sua vez, descreve um conjunto de ligações com estrutura e semântica comuns. No exemplo anterior, há uma associação entre as classes *Empregado* e *Departamento*. Todas as ligações de uma associação interligam objetos das mesmas classes e, assim, uma associação descreve um conjunto de potenciais ligações da mesma maneira que uma classe descreve um conjunto de potenciais objetos (BLAHA; RUMBAUGH, 2006).

Uma associação comum entre duas classes representa um relacionamento estrutural entre pares, significando que essas duas classes estão em um mesmo nível, sem que uma seja mais importante do que a outra. Além das associações comuns, a UML considera dois tipos de

associações especiais entre objetos: *composição* e *agregação*. Ambos representam relações todo-parte. A agregação é uma forma especial de associação que especifica um relacionamento entre um objeto agregado (o todo) e seus componentes (as partes). A composição, por sua vez, é uma forma de agregação na qual o tempo de vida entre todo e partes é coincidente. As partes podem até ser criadas após a criação do todo, mas uma vez criadas, vivem e morrem com o todo. Uma parte pode ainda ser removida explicitamente antes da morte do todo (BOOCH; RUMBAUGH; JACOBSON, 2006).

d) Generalização / Especialização

Muitas vezes, um conceito geral pode ser especializado, adicionando-se novas características. Seja o exemplo do conceito de *estudante* no contexto de uma universidade. De modo geral, há características que são intrínsecas a quaisquer estudantes da universidade. Entretanto, é possível especializar este conceito para mostrar especificidades de subtipos de estudantes, tais como estudantes de graduação e estudantes de pós-graduação.

Da maneira inversa, pode-se extrair de um conjunto de conceitos, características comuns que, quando generalizadas, formam um conceito geral. Por exemplo, ao se avaliar os conceitos de carros, motos, caminhões e ônibus, pode-se notar que eles têm características comuns que podem ser generalizadas em um supertipo *veículo automotor terrestre*.

As abstrações de especialização e generalização são muito úteis para a estruturação de sistemas. Com elas, é possível construir hierarquias de classes. A *herança* é um mecanismo para modelar similaridades entre classes, representando as abstrações de generalização e especialização. Através da herança, é possível tornar explícitos propriedades e operações comuns em uma hierarquia de classes. O mecanismo de herança possibilita reutilização, captura explícita de características comuns e definição incremental de classes.

No que se refere à definição incremental de classes, a herança permite conceber uma nova classe como um refinamento de outras classes. A nova classe pode herdar as similaridades e definir apenas as novas características.

A herança é, portanto, um relacionamento entre classes (em contraposição às associações que representam relacionamentos entre objetos das classes), no qual uma classe compartilha a estrutura e comportamento definido em uma ou mais outras classes. A classe que herda características[3] é chamada *subclasse* e a que fornece as características, *superclasse*. Desta forma, a herança representa uma hierarquia de abstrações na qual uma subclasse herda de uma ou mais superclasses.

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

A generalização permite abstrair, a partir de um conjunto de classes, uma classe mais geral contendo todas as características comuns. A especialização é a operação inversa e, portanto, permite especializar uma classe em um número de subclasses, explicitando as diferenças entre as novas subclasses. Deste modo é possível compor a hierarquia de classes. Esses tipos de relacionamento são conhecidos também como relacionamentos "*é um tipo de*",

[3] O termo *característica* é usado aqui para designar estrutura (atributos e associações) e comportamento (operações).

onde um objeto da subclasse também "*é um tipo de*" objeto da superclasse. Neste caso uma instância da subclasse é dita uma *instância indireta* da superclasse.

e) Mensagens e Métodos

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

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

f) Classes e Operações Abstratas

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

Classes abstratas podem ser projetadas de duas maneiras distintas. Primeiro, elas podem prover implementações completamente funcionais do comportamento que pretendem capturar. Alternativamente, elas podem prover apenas definição de um protocolo para uma operação sem apresentar um método correspondente. Tal operação é dita uma *operação genérica ou abstrata*. Neste caso, a classe abstrata não é completamente implementada e todas as suas subclasses concretas são obrigadas a prover uma implementação para suas operações abstratas. Assim, diz-se que uma operação abstrata define apenas a assinatura[4] a ser usada nas implementações que as subclasses deverão prover, garantindo, assim, uma interface consistente. Métodos que implementam uma operação genérica têm a mesma semântica.

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

5.3 Um Método de Análise de Requisitos Funcionais

Uma vez que tipicamente diversos modelos do sistema são produzidos, surgem algumas importantes questões: Que modelos produzir? Em que sequência? Quais as relações existentes

[4] nome da operação, parâmetros e retorno.

entre esses modelos? Estas questões podem ser parcialmente respondidas pela adoção de um método de análise.

Um método é composto por uma linguagem estabelecendo a notação a ser usada na elaboração dos artefatos a serem produzidos e de um processo descrevendo que artefatos construir e como construí-los.

O método sugerido neste texto adota a Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML) (BOOCH; RUMBAUGH; JACOBSON, 2006) como linguagem de modelagem e prescreve o processo ilustrado na Figura 5.4. Nessa figura, as atividades de modelagem propriamente ditas estão destacadas em amarelo. As demais atividades correspondem a atividades de documentação e verificação e validação do Documento de Especificação de Requisitos.

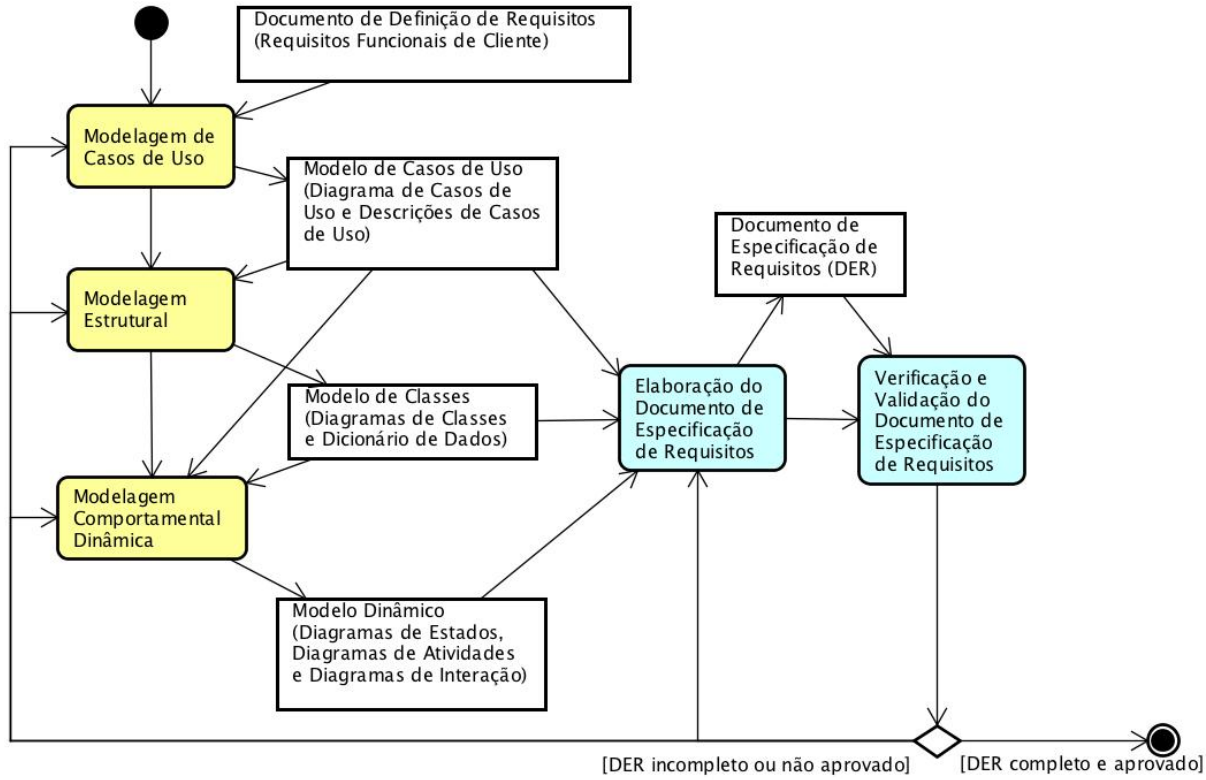

Figura 5.4 – O Método de Análise de Requisitos Funcionais Proposto.

O primeiro modelo a ser construído é o modelo de casos de uso. Sua escolha como primeiro modelo deve-se ao fato do modelo de casos de uso ser muito simples e, portanto, passível de compreensão tanto por desenvolvedores – analistas, projetistas, programadores e testadores – como pela comunidade usuária – clientes, usuários e demais interessados. O modelo de casos de uso é um modelo comportamental, mostrando as funções do sistema de maneira estática. Ele é composto de dois artefatos: os diagramas de casos de uso e as descrições de casos de uso. O diagrama de casos de uso descreve graficamente o sistema, seu ambiente e como sistema e ambiente se relacionam. Assim, ele descreve o sistema segundo uma perspectiva externa. As descrições dos casos de uso descrevem o passo a passo para a realização dos casos de uso e são essencialmente textuais. Elas tratam de como os casos de uso são realizados internamente, complementando os diagramas. Casos de uso são tratados na seção 5.6.

Tomando por base casos de uso e suas descrições, é possível passar à modelagem conceitual estrutural, quando os conceitos e relacionamentos envolvidos no domínio são capturados em um conjunto de diagramas de classes. Neste momento é importante definir, também, o significado dos conceitos e de suas propriedades, bem como restrições sobre eles. Essas definições são documentadas em um dicionário de dados do projeto. A modelagem conceitual estrutural é o foco da seção 5.7.

Algumas classes do modelo estrutural apresentam um comportamento dependente de seu estado. Para essas classes, é útil elaborar diagramas de estados, mostrando os estados pelos quais um objeto da classe pode passar ao longo de sua existência e os eventos que provocam transições de estados. Além disso, uma vez que casos de uso são representados apenas de maneira estática, pode ser útil também mostrar a dinâmica de um caso de uso, representando como objetos do diagrama de classes interagem para realizar um caso de uso. Os diagramas de interação são elaborados com este propósito. Finalmente, pode ser útil representar os passos de um caso de uso na forma de um diagrama mostrando objetos gerados e os atores envolvidos em cada passo. Para tal, diagramas de atividades podem ser elaborados. No âmbito da modelagem dinâmica, nestas notas de aula, apenas os diagramas de estados serão detalhados (seção 5.8).

Deve-se frisar que, apesar da Figura 5.4 sugerir que os passos do método são sequenciais, na prática, isso não ocorre. Paralelamente à modelagem de casos de uso, pode-se iniciar a modelagem conceitual estrutural. Os diagramas de atividade podem também ser elaborados em conjunto com a definição dos casos de uso, de maneira a complementar a descrição de casos de uso específicos. Diagramas de estado e de interação requerem a definição preliminar de casos de uso e classes. Contudo, podem ter sido definidos apenas alguns casos de uso e classes (e não todos eles) para já iniciar a elaboração desses diagramas. Assim, as atividades mostradas na Figura 5.4 são fortemente paralelas e iterativas.

Paralelamente a todas as atividades de modelagem, o documento de especificação de requisitos deve ir sendo elaborado. A verificação e a validação também podem ser feitas por partes e não para o documento como um todo. Por exemplo, é bastante comum validar primeiro somente os casos de uso. Verificações de consistência, tais como as feitas entre casos de uso e classes, ou casos de uso, diagramas de interação e diagramas de classes, podem ser feitas separadamente uma das outras. Assim, as atividades de documentação, verificação e validação são atividades contínuas que ocorrem paralelamente à modelagem conceitual.

5.4 Especificação de Requisitos Não Funcionais

Assim como os requisitos funcionais precisam ser especificados em detalhes, o mesmo acontece com os requisitos não funcionais. Para os atributos de qualidade considerados prioritários, o analista deve trabalhar no sentido de especificá-los de modo que eles se tornem mensuráveis e, por conseguinte, testáveis.

Para cada atributo de qualidade, devem-se definir as medidas a serem usadas, indicando a unidade da medida e sua escala, e os valores mínimo, alvo e máximo. Pode-se, ainda, perguntar aos usuários o que constituiria um valor inaceitável para o atributo e definir testes que tentem forçar o sistema a demonstrar tais características (WIEGERS, 2003). Ao se estabelecer uma escala de medição e os valores aceitáveis, o requisito é transformado de uma intenção vaga, e até certo ponto ambígua, em um requisito mensurável e bem formado. Estabelecida uma escala, pode-se perguntar ao usuário o que é considerado uma falha em atender ao requisito, de modo a definir o critério de aceitação do mesmo (ROBERTSON;

ROBERTSON, 2006). Assim, na especificação de requisitos de sistema, é importante transformar um requisito de usuário em um requisito mensurável, adicionando a ele um critério de aceitação.

A ISO/IEC 25023 – Medidas de Qualidade de Produtos de Software e Sistemas (ISO/IEC, 2016), ou a sua antecessora, a ISO/IEC 9126, pode ser uma boa fonte de medidas. As partes 2 (Medidas Externas) (ISO/IEC, 2003a) e 3 (Medidas Internas) (ISO/IEC, 2003b) da ISO/IEC 9126 apresentam diversas medidas que podem ser usadas para especificar objetivamente os requisitos não funcionais. Essas medidas foram revistas e atualizadas na ISO/IEC 25023. Nessas normas, medidas são sugeridas para as diversas subcaracterísticas de qualidade externa e interna descritas na atual ISO/IEC 25010 (ISO/IEC, 2011), indicando, dentre outros, nome e propósito da medida, método de aplicação e fórmula, e como interpretar os valores da medida.

Seja o exemplo de um sistema que tem como requisito não funcional ser fácil de aprender. Esse requisito poderia ser especificado conforme mostrado na Tabela 5.1.

Tabela 5.1 – Especificação de Requisito Não Funcional.

RNF01 – A funcionalidade "Efetuar Locação de Item" deve ser fácil de aprender.		
Medida: Facilidade de Aprendizagem de função (*Ease of function learn*) (ISO/IEC, 2003a)	Descrição:	Facilidade de aprender a realizar uma tarefa em uso.
	Propósito:	Quanto tempo o usuário leva para aprender a realizar uma tarefa especificada eficientemente?
	Método de Aplicação:	Observar o comportamento do usuário desde quando ele começa a aprender até quando ele começa a operar eficientemente.
	Medição:	T = soma do tempo de operação do usuário até que ele consiga realizar a tarefa em um tempo especificado (tempo requerido para aprender a operação para realizar a tarefa).
Critério de Aceitação:	T <= 15 minutos, considerando que o usuário está operando o sistema eficientemente quando a tarefa "Efetuar Locação" é realizada em um tempo inferior a 2 minutos.	

5.5 O Documento de Especificação de Requisitos

Os requisitos de sistema, assim como foi o caso dos requisitos de usuário, têm de ser especificados em um documento, de modo a poderem ser verificados e validados e posteriormente usados como base para as atividades subsequentes do desenvolvimento de software. O Documento de Especificação de Requisitos tem como propósito registrar os requisitos escritos a partir da perspectiva do desenvolvedor e, portanto, deve incluir os vários modelos conceituais desenvolvidos, bem como a especificação dos requisitos não funcionais detalhados usando critérios de aceitação ou cenários de atributos de qualidade.

Diferentes formatos podem ser propostos para documentos de especificação requisitos, bem como mais de um documento pode ser usado para documentar os requisitos de sistema. Neste texto, propõe-se o uso de um único documento, contendo as seguintes informações:

1. *Introdução:* breve introdução ao documento, descrevendo seu propósito e estrutura.
2. *Modelo de Casos de Uso:* apresenta o modelo de casos de uso do sistema, incluindo os diagramas de casos de uso e as descrições de casos de uso associadas.
3. *Modelo Estrutural:* apresenta o modelo estrutural do sistema, incluindo os diagramas de classes do sistema.

4. *Modelo Dinâmico:* apresenta os modelos comportamentais dinâmicos do sistema, incluindo os diagramas de estados, diagramas de interação e diagramas de atividades[5].
5. *Dicionário do Projeto:* apresenta as definições dos principais conceitos capturados pelos diversos modelos e restrições de integridade a serem consideradas, servindo como um glossário do projeto.
6. *Especificação dos Requisitos Não Funcionais:* apresenta os requisitos não funcionais descritos no nível de sistema, o que inclui critérios de aceitação e cenários de atributos de qualidade.

É importante frisar que dificilmente um sistema é simples o bastante para ser modelado como um todo. Quase sempre é útil dividir um sistema em unidades menores, mais fáceis de serem gerenciáveis, ditas subsistemas. É útil organizar a especificação de requisitos por subsistemas e, portanto, cada uma das seções propostas acima pode ser subdividida por subsistemas.

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

A UML (*Unified Modeling Language*) provê um tipo principal de item de agrupamento, denominado pacote, que é um mecanismo de propósito geral para a organização de elementos da modelagem em grupos. Um diagrama de pacotes mostra a decomposição de um modelo em unidades menores e suas dependências, como ilustra a Figura 5.5. A linha pontilhada direcionada indica que o pacote origem (no exemplo, o pacote Atendimento a Cliente) depende do pacote destino (no exemplo, o pacote Controle de Acervo).

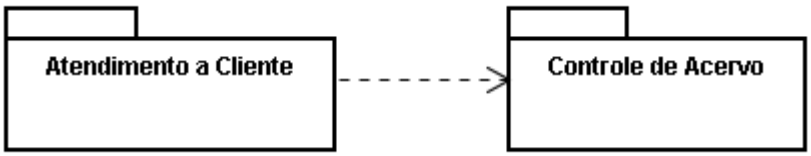

Figura 5.5– Exemplo de um Diagrama de Pacotes.

Nas próximas seções são realizadas as discussões necessárias para que sejam elaborados os Modelos de Casos de Uso, os Modelos Estruturais e os Diagramas de Estados (modelo dinâmico).

5.6 Modelagem de Casos de Uso

Modelos de caso de uso (*use cases*) são modelos passíveis de compreensão tanto por desenvolvedores – analistas, projetistas, programadores e testadores – como pela comunidade usuária – clientes e usuários. Como o próprio nome sugere, um caso de uso é uma maneira de usar o sistema. Usuários interagem com o sistema, interagindo com seus casos de uso. Tomados

[5] Neste material, no que diz respeito à modelagem dinâmica, serão abordados apenas os diagramas de estados.

em conjunto, os casos de uso de um sistema representam a sua funcionalidade. Casos de uso são, portanto, os "itens" que o desenvolvedor negocia com seus clientes.

O propósito do modelo de casos de uso é capturar e descrever a funcionalidade que um sistema deve prover. Um sistema geralmente serve a vários atores, para os quais ele provê diferentes serviços. Tipicamente, a funcionalidade a ser provida por um sistema é muito grande para ser analisada como uma única unidade e, portanto, é importante ter um mecanismo de dividir essa funcionalidade em partes menores e mais gerenciáveis. O conceito de caso de uso é muito útil para esse propósito (OLIVÉ, 2007).

É importante ter em mente que casos de uso são fundamentalmente uma ferramenta textual. Ainda que casos de uso sejam também descritos graficamente (p.ex., fluxogramas ou algum diagrama da UML, dentre eles diagramas de casos de uso, diagramas de sequência e diagramas de atividades), não se deve perder de vista a natureza textual dos casos de uso. Olhando casos de uso apenas a partir da UML, que não trata do conteúdo ou da escrita de casos de uso, pode-se pensar, equivocadamente, que casos de uso são uma construção gráfica ao invés de textual.

Em essência, casos de uso servem como um meio de comunicação entre pessoas, algumas delas sem nenhum treinamento especial e, portanto, o uso de texto para especificar casos de uso é geralmente a melhor escolha. Casos de uso são amplamente usados no desenvolvimento de sistemas, porque, por meio sobretudo de suas descrições textuais, usuários e clientes conseguem visualizar qual a funcionalidade a ser provida pelo sistema, conseguindo reagir mais rapidamente no sentido de refinar, alterar ou rejeitar as funções previstas para o sistema (COCKBURN, 2005). Assim, um modelo de casos de uso inclui duas partes principais: (i) os diagramas de casos de uso e (ii) as descrições de atores e de casos de uso, sendo que essas últimas podem ser complementadas com outros diagramas associados, tais como os diagramas de atividade e de sequência da UML.

Outro aspecto a ser realçado é que os modelos de caso de uso são independentes do método de análise a ser usado e até mesmo do paradigma de desenvolvimento. Assim, pode-se utilizar a modelagem de casos de uso tanto no contexto do desenvolvimento orientado a objetos (foco deste texto), como em projetos desenvolvidos segundo o paradigma estruturado. De fato, o uso de modelos de caso de uso pode ser ainda mais amplo. Casos de uso podem ser usados, por exemplo, para documentar processos de negócio de uma organização. Contudo, neste texto, explora-se a utilização de casos de uso para modelar e documentar requisitos funcionais de sistemas. Assim, geralmente são interessados[6] (*stakeholders*) nos casos de uso: as pessoas que usarão o sistema (usuários), o cliente que requer o sistema, outros sistemas com os quais o sistema em questão terá de interagir e outros membros da organização (ou até mesmo de fora dela) que têm restrições que o sistema precisa garantir.

Esta seção enfoca a modelagem de casos de uso e está estruturada conforme descrito a seguir. A subseção 5.6.1 discute os dois principais conceitos empregados na modelagem de casos de uso: atores e casos de uso. A subseção 5.6.2 aborda os diagramas de casos de uso e sua notação segundo a UML. A subseção 5.6.3 trata da especificação de casos de uso. A subseção 5.6.4 discute os tipos de relacionamentos que podem ser estabelecidos entre casos de uso, a saber: inclusão, extensão e generalização / especialização. Finalmente, a subseção 5.6.5 discute como trabalhar com casos de uso.

[6] Alguém ou algo com interesse no comportamento do sistema sob discussão (COCKBURN, 2005).

5.6.1 Atores e Casos de Uso

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

Segundo Cockburn (2005), um caso de uso captura um contrato entre os interessados (*stakeholders*) em um sistema sobre o seu comportamento. Um caso de uso descreve o comportamento do sistema sob certas condições, em resposta a uma requisição feita por um interessado, dito o ator primário do caso de uso. Assim, os dois principais conceitos da modelagem de casos de uso são atores e casos de uso.

Atores

Dá-se nome de ator a um papel desempenhado por entidades físicas (pessoas, organizações ou outros sistemas) que interagem com o sistema em questão da mesma maneira, procurando atingir os mesmos objetivos. Uma mesma entidade física pode desempenhar diferentes papéis no mesmo sistema, bem como um dado papel pode ser desempenhado por diferentes entidades (OLIVÉ, 2007).

Atores são externos ao sistema. Um ator se comunica diretamente com o sistema, mas não é parte dele. A modelagem dos atores ajuda a definir as fronteiras do sistema, isto é, o conjunto de atores de um sistema delimita o ambiente externo desse sistema, representando o conjunto completo de entidades para as quais o sistema pode servir (BLAHA; RUMBAUGH, 2006; OLIVÉ, 2007).

Uma dúvida que sempre passa pela cabeça de um iniciante em modelagem de casos de uso é saber se o ator é a pessoa que efetivamente opera o sistema (p.ex., o atendente de uma locadora de automóveis) ou se é a pessoa interessada no resultado do processo (p.ex., o cliente que efetivamente loca o automóvel e é atendido pelo atendente). Essa definição depende, em essência, da fronteira estabelecida para o sistema. Sistemas de informação podem ter diferentes níveis de automatização. Por exemplo, se um sistema roda na Internet, seu nível de automatização é maior do que se ele requer um operador. Assim, é importante capturar qual o nível de automatização requerido e levar em conta o real limite do sistema (WAZLAWICK, 2004). Se o caso de uso roda na Internet (p.ex., um caso de uso de reserva de automóvel), então o cliente é o ator efetivamente. Se o caso de uso requer um operador (p.ex., um caso de uso de locação de automóvel, disponível apenas na locadora e para ser usado por atendentes), então o operador é o ator.

Quando se for considerar um sistema como sendo um ator, deve-se tomar o cuidado para não confundir a ideia de sistema externo (ator) com produtos usados na implementação do sistema em desenvolvimento. Para que um sistema possa ser considerado um ator, ele deve ser um sistema de informação completo (e não apenas uma biblioteca de classes, por exemplo). Além disso, ele deve estar fora do escopo do desenvolvimento do sistema atual. O analista não terá a oportunidade de alterar as funções do sistema externo, devendo adequar a comunicação às características do mesmo (WAZLAWICK, 2004).

Um ator primário é um ator que possui metas a serem cumpridas através do uso de serviços do sistema e que, tipicamente, inicia a interação com o sistema (OLIVÉ, 2007). Um ator secundário é um ator externo que interage com o sistema para prover um serviço para este último. A identificação de atores secundários é importante, uma vez que ela permite identificar interfaces externas que o sistema usará e os protocolos que regem as interações ocorrendo através delas (COCKBURN, 2005).

De maneira geral, o ator primário é o usuário direto do sistema ou outro sistema computacional que requisita um serviço do sistema em desenvolvimento. O sistema responde à requisição procurando atendê-la, ao mesmo tempo em que protege os interesses de todos os demais interessados no caso de uso. Entretanto, há situações em que o iniciador do caso de uso não é o ator primário. O tempo, por exemplo, pode ser o acionador de um caso de uso. Um caso de uso que roda todo dia à meia noite ou ao final do mês tem o tempo como acionador. Mas o caso de uso ainda visa atingir um objetivo de um ator e esse ator é considerado o ator primário do caso de uso, ainda que ele não interaja efetivamente com o sistema (COCKBURN, 2005).

Para nomear atores, recomenda-se o uso de substantivos no singular, iniciados com letra maiúscula, possivelmente combinados com adjetivos. Exemplos: Cliente, Bibliotecário, Correntista, Correntista Titular etc.

Casos de Uso

Um caso de uso é uma porção coerente da funcionalidade que um sistema pode fornecer para atores interagindo com ele (BLAHA; RUMBAUGH, 2006). Um caso de uso corresponde a um conjunto de ações realizadas pelo sistema (ou por meio da interação com o sistema), que produz um resultado observável, com valor para um ou mais atores do sistema. Geralmente, esse valor é a realização de uma meta de negócio ou tarefa (OLIVÉ, 2007). Assim, um caso de uso captura alguma função visível ao ator e, em especial, busca atingir uma meta desse ator.

Deve-se considerar que um caso de uso corresponde a uma transação completa, ou seja, um usuário poderia ativar o sistema, executar o caso de uso e desativar o sistema logo em seguida, e a operação estaria completa e consistente e atenderia a uma meta desse usuário (WAZLAWICK, 2004).

Ser uma transação completa é uma característica essencial de um caso de uso[7], pois somente transações completas são capazes de atingir um objetivo do usuário. Casos de uso que necessitam de múltiplas seções não passam nesse critério e devem ser divididos em casos de uso menores. Seja o exemplo de um caso de uso de concessão de empréstimo. Inicialmente, um atendente interagindo com um cliente informa os dados necessários para a avaliação do pedido de empréstimo. O pedido de empréstimo é, então, enviado para análise por um analista de crédito. Uma vez analisado e aprovado, o empréstimo é concedido, quando o dinheiro é entregue ao cliente e um contrato é assinado, dentre outros. Esse processo pode levar vários dias e não é realizado em uma seção única. Assim, o caso de uso de concessão de empréstimo deveria ser subdividido em casos de uso menores, tais como casos de uso para efetuar pedido de empréstimo, analisar pedido de empréstimo e formalizar concessão de empréstimo.

[7] Esta regra tem como exceção os casos de uso de inclusão e extensão, conforme discutido mais adiante na seção que trata de relacionamentos entre casos de uso.

Por outro lado, casos de uso muito pequenos, que não caracterizam uma transação completa, devem ser considerados passos de um caso de uso maior[8]. Seja o exemplo de uma biblioteca a qual cobra multa na devolução de livros em atraso. Um caso de uso específico para apenas calcular o valor da multa não é relevante, pois não caracteriza uma transação completa capaz de atingir um objetivo do usuário. O objetivo do usuário é efetuar a devolução e, neste contexto, uma regra de negócio (a que estabelece a multa) tem de ser levada em conta. Assim, calcular a multa é apenas um passo do caso de uso que efetua a devolução, o qual captura uma ação do sistema para garantir a regra de negócio e, portanto, satisfazer um interesse da biblioteca como organização.

Um caso de uso reúne todo o comportamento relevante de uma parte da funcionalidade do sistema. Isso inclui o comportamento principal normal, as variações de comportamento normais, as condições de exceção e o cancelamento de uma requisição. O conjunto de casos de uso captura a funcionalidade completa do sistema (BLAHA; RUMBAUGH, 2006).

Casos de uso fornecem uma abordagem para os desenvolvedores chegarem a uma compreensão comum com os usuários finais e especialistas do domínio, acerca da funcionalidade a ser provida pelo sistema (BOOCH; RUMBAUGH; JACOBSON, 2006).

Os objetivos dos atores são um bom ponto de partida para a identificação de casos de uso. Pode-se propor um caso de uso para satisfazer cada um dos objetivos de cada um dos atores. A partir desses objetivos, podem-se estudar as possíveis interações do ator com o sistema e refinar o modelo de casos de uso.

Cada caso de uso tem um nome. Esse nome deve capturar a essência do caso de uso. Para nomear casos de uso sugere-se usar frases iniciadas com verbos no infinitivo, seguidos de complementos, que representem a meta ou tarefa a ser realizada com o caso de uso. As primeiras letras (exceto preposições) de cada palavra devem ser grafadas em letra maiúscula. Exemplos: Cadastrar Cliente, Devolver Livro, Efetuar Pagamento de Fatura etc.

Um caso de uso pode ser visto como um tipo cujas instâncias são *cenários*. Um cenário é uma execução de um caso de uso com entidades físicas particulares desempenhando os papéis dos atores e em um particular estado do domínio de informação. Um cenário, portanto, exercita um certo caminho dentro do conjunto de ações de um caso de uso (OLIVÉ, 2007).

Alguns cenários mostram o objetivo do caso de uso sendo alcançado; outros terminam com o caso de uso sendo abandonado (COCKBURN, 2005). Mesmo quando o objetivo de um caso de uso é alcançado, ele pode ser atingido seguindo diferentes caminhos. Assim, um caso de uso deve comportar todas essas situações. Para tal, um caso de uso é normalmente descrito por um conjunto de fluxos de eventos, capturando o fluxo de eventos principal, i.e., o fluxo de eventos típico que conduz ao objetivo do caso de uso, e fluxos de eventos alternativos, descrevendo exceções ou variantes do fluxo principal.

5.6.2 Diagramas de Casos de Uso

Basicamente, um diagrama de casos de uso mostra um conjunto de casos de uso e atores e seus relacionamentos, sendo utilizado para ilustrar uma visão estática das maneiras possíveis de se usar o sistema (BOOCH; RUMBAUGH; JACOBSON, 2006).

[8] As mesmas exceções da nota anterior se aplicam aqui, conforme discutido mais adiante.

Os diagramas de casos de uso da UML podem conter os seguintes elementos de modelo, ilustrados na Figura 5.6 (BOOCH; RUMBAUGH; JACOBSON, 2006):

- *Assunto:* o assunto delimita a fronteira de um diagrama de casos de uso, sendo normalmente o sistema ou um subsistema. Os casos de uso de um assunto descrevem o comportamento completo do assunto. O assunto é exibido em um diagrama de casos de uso como um retângulo envolvendo os casos de uso que o compõem. O nome do assunto (sistema ou subsistema) pode ser mostrado dentro do retângulo.
- *Ator:* representa um conjunto coerente de papéis que os usuários ou outros sistemas desempenham quando interagem com os casos de uso. Tipicamente, um ator representa um papel que um ser humano, um dispositivo de hardware ou outro sistema desempenha com o sistema em questão. Atores não são parte do sistema. Eles residem fora do sistema. Atores são representados por um ícone de homem, com o nome colocado abaixo do ícone.
- *Caso de Uso:* representa uma funcionalidade que o sistema deve prover. Casos de uso são parte do sistema e, portanto, residem dentro dele. Um caso de uso é representado por uma elipse com o nome do caso de uso dentro ou abaixo dela.
- *Relacionamentos de Dependência, Generalização e Associação*: são usados para estabelecer relacionamentos entre atores, entre atores e casos de uso, e entre casos de uso.

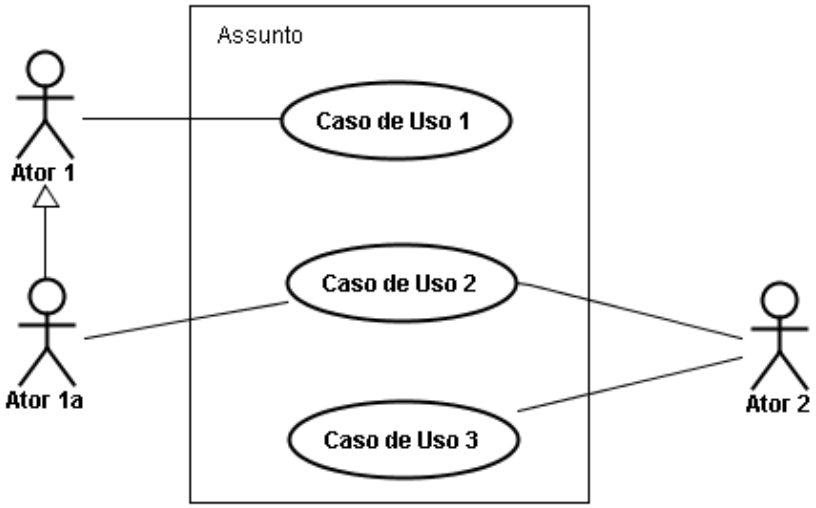

Figura 5.6 - Diagrama de Casos de Uso – Conceitos e Notação.

Atores só podem estar conectados a casos de uso por meio de *associações*. Uma associação entre um ator e um caso de uso significa que estímulos podem ser enviados entre atores e casos de uso. 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).

Atores podem ser organizados em hierarquias de generalização / especialização, de modo a capturar que um ator filho herda o significado e as associações com casos de uso de seu pai, especializando esse significado e potencialmente adicionando outras associações como outros casos de uso.

A Figura 5.7 mostra um diagrama de casos de uso para um sistema de caixa automático. Nesse diagrama, o assunto é o sistema como um todo. Os atores são: os clientes do banco, o sistema bancário e os responsáveis pela manutenção do numerário no caixa eletrônico. Cliente e mantenedor são atores primários, uma vez que têm objetivos a serem atingidos pelo uso do sistema. O sistema bancário é um ator, pois o sistema do caixa automático precisa interagir com o sistema bancário para realizar os casos de uso *Efetuar Saque*, *Emitir Extrato* e *Efetuar Pagamento*.

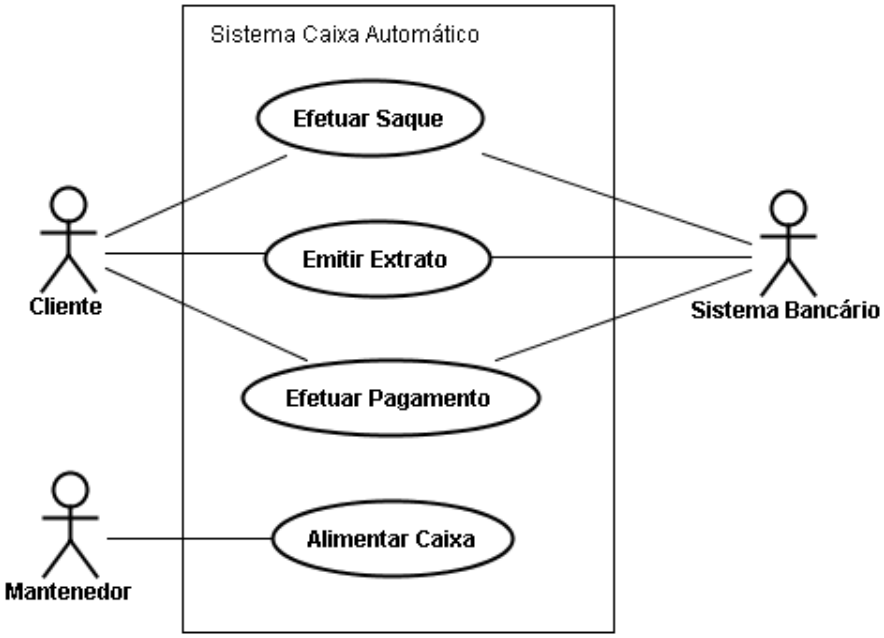

Figura 5.7 - Diagrama de Casos de Uso – Caixa Automático.

Um caso de uso descreve o que um sistema deve fazer. O diagrama de casos de uso provê uma visão apenas parcial disso, uma vez que mostra as funcionalidades por perspectiva externa. É necessário, ainda, capturar uma visão interna de cada caso de uso, especificando o comportamento do caso de uso pela descrição do fluxo de eventos que ocorre internamente (passos do caso de uso). Assim, uma parte fundamental do modelo de casos de uso é a descrição dos casos de uso.

5.6.3 Descrevendo Casos de Uso

Um caso de uso deve descrever *o que* um sistema faz. Exceto para situações muito simples, um diagrama de casos de uso é insuficiente para este propósito. Assim, deve-se especificar o comportamento de um caso de uso pela descrição textual de seu fluxo de eventos, de modo que outros interessados possam compreendê-lo.

A especificação ou descrição de um caso de uso deve conter, dentre outras informações, um conjunto de sentenças, cada uma delas escrita em uma forma gramatical designando um passo simples, de modo que aprender a ler um caso de uso não requeira mais do que uns poucos minutos. Dependendo da situação, diferentes estilos de escrita podem ser adotados (COCKBURN, 2005).

Cada passo do fluxo de eventos de um caso de uso tipicamente descreve uma das seguintes situações: (i) uma interação entre um ator e o sistema, (ii) uma ação que o sistema

realiza para atingir o objetivo do ator primário ou (iii) uma ação que o sistema realiza para proteger os interesses de um interessado. Essas ações podem incluir validações e mudanças do estado interno do sistema (COCKBURN, 2005).

Não há um padrão definido para especificar casos de uso. Diferentes autores propõem diferentes estruturas, formatos e conteúdos para descrições de casos de uso, alguns mais indicados para casos de uso essenciais e mais complexos, outros para casos de uso cadastrais e mais simples. Mais além, pode ser útil utilizar mais de um formato dentro do mesmo projeto, em função das peculiaridades de cada caso de uso. De todo modo, é recomendável que a organização defina modelos de descrição de casos de uso a serem adotados em seus projetos, devendo definir tantos modelos quantos julgar necessários.

As seguintes informações são um bom ponto de partida para a definição de um modelo de descrição de casos de uso:

- *Nome:* nome do caso de uso, capturando a sua essência
- *Escopo:* diz respeito ao que está sendo documentado pelo caso de uso. Tipicamente pode ser um processo de negócio, um sistema ou um subsistema. Vale lembrar que este texto não aborda a utilização de casos de uso para a modelagem de processos de negócio. Assim, o escopo vai apontar o sistema / subsistema do qual o caso de uso faz parte.
- *Descrição do Propósito:* uma descrição sucinta do caso de uso, na forma de um único parágrafo, procurando descrever o objetivo do caso de uso.
- *Ator Primário:* nome do ator primário, ou seja, o interessado que tem um objetivo em relação ao sistema, o qual pode ser atingido pela execução do caso de uso.
- *Interessados e Interesses:* um interessado é alguém ou algo (um outro sistema) que tem um interesse no comportamento do caso de uso sendo descrito. Nesta seção são descritos cada um dos interessados no sistema e qual o seu interesse no caso de uso, incluindo o ator primário.
- *Pré-condições:* o que deve ser verdadeiro antes da execução do caso de uso. Se as pré-condições não forem satisfeitas, o caso de uso não pode ser realizado.
- *Pós-condições:* o que deve ser verdadeiro após a execução do caso de uso, considerando que o fluxo de eventos normal é realizado com sucesso.
- *Fluxo de Eventos Normal:* descreve os passos do caso de uso realizados em situações normais, considerando que nada acontece de errado e levando em conta a maneira mais comum do caso de uso ser realizado.
- *Fluxo de Eventos Alternativos:* descreve formas alternativas de realizar certos passos do caso de uso. Há duas formas alternativas principais: fluxos variantes, que são considerados dentro da normalidade do caso de uso; e fluxos de exceção, que se referem ao tratamento de erros durante a execução de um passo do fluxo normal (ou de um fluxo variante ou até mesmo de um outro fluxo de exceção).
- *Requisitos Relacionados:* listagem dos identificadores dos requisitos (funcionais, não funcionais e regras de negócio) tratados pelo caso de uso sendo descrito, de modo a permitir rastrear os requisitos. Casos de uso podem ser usados para conectar vários requisitos, de tipos diferentes. Assim, essa listagem ajuda a manter um rastro

entre requisitos funcionais, não funcionais e regras de negócio, além de permitir verificar se algum requisito deixou de ser tratado.

- *Classes / Entidades:* classes (no paradigma orientado a objetos) ou entidades (no paradigma estruturado) necessárias para tratar o caso de uso sendo descrito. Esta seção é normalmente preenchida durante a modelagem conceitual estrutural e é igualmente importante para permitir rastrear requisitos para as etapas subsequentes do desenvolvimento (projeto e implementação, sobretudo).

Descrevendo os Fluxos de Eventos

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. O fluxo de eventos normal (ou principal) é uma informação essencial na descrição de um caso de uso e não pode ser omitido em nenhuma circunstância. O fluxo de eventos normal é, portanto, a principal seção de uma descrição de caso de uso, a qual descreve o processo quando tudo dá certo, ou seja, sem a ocorrência de nenhuma exceção (WAZLAWICK, 2004).

Variantes do curso básico de eventos e tratamento de exceções que possam vir a ocorrer devem ser descritos em cursos alternativos. Normalmente, um caso de uso possui apenas um único curso básico, mas diversos cursos alternativos. Seja o exemplo de um sistema de caixa automático de banco, cujo diagrama de casos de uso é mostrado na Figura 5.8. O caso de uso *Efetuar Saque* poderia ser descrito como mostrado na Figura 5.9.

Como visto nesse exemplo, um caso de uso pode ter um número de cursos alternativos que podem levar o caso de uso por diferentes caminhos. Tanto quanto possível, esses cursos alternativos, muitos deles cursos de exceção, devem ser identificados durante a especificação do fluxo de eventos normal de um caso do uso.

Vale realçar que uma exceção não é necessariamente um evento que ocorre muito raramente, mas sim um evento capaz de impedir o prosseguimento do caso de uso, se não for devidamente tratado. Uma exceção também não é algo que impede o caso de uso de ser iniciado, mas algo que impede a sua conclusão. Condições que impedem um caso de uso de ser iniciado devem ser tratadas como pré-condições. As pré-condições nunca devem ser testadas durante o processo do caso de uso, pois, por definição, elas impedem que o caso de uso seja iniciado. Logo, seria inconsistente imaginar que elas pudessem ocorrer durante a execução do caso de uso. Se uma pré-condição é falsa, então o caso de uso não pode ser iniciado (WAZLAWICK, 2004).

Observa-se que a maioria das exceções ocorre nos passos em que alguma informação é passada dos atores para o sistema. Isso porque, quando uma informação é passada para o sistema, muitas vezes ele realiza validações. Quando uma dessas validações falha, tipicamente ocorre uma exceção (WAZLAWICK, 2004).

Nome: Efetuar Saque

Escopo: Sistema de Caixa Automático

Descrição do Propósito: Este caso de uso permite que um cliente do banco efetue um saque, retirando dinheiro de sua conta bancária.

Ator Primário: Cliente
Interessados e Interesses:

- Cliente: deseja efetuar um saque.
- Banco: garantir que apenas o próprio cliente efetuará saques e que os valores dos saques sejam compatíveis com o limite de crédito do cliente.

Pré-condições: O caixa automático deve estar conectado ao sistema bancário.

Pós-condições: O saque é efetuado, debitando o valor da conta do cliente e entregando o mesmo valor para o cliente em espécie.

Fluxo de Eventos Normal

O cliente insere seu cartão no caixa automático, que analisa o cartão e verifica se ele é aceitável. Se o cartão é aceitável, o caixa automático solicita que o cliente informe a senha. O cliente informa a senha. O caixa automático envia os dados do cartão e da senha para o sistema bancário para validação. Se a senha estiver correta, o caixa solicita que o cliente informe o tipo de transação a ser efetuada. O cliente seleciona a opção saque e o caixa solicita que seja informada a quantia. O cliente informa a quantia a ser sacada. O caixa envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada. Se o saque é autorizado, as notas são preparadas e liberadas.

Fluxos de Eventos de Exceção

- O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada.
- Senha incorreta: Se a senha informada está incorreta, uma mensagem é mostrada para o cliente que poderá entrar com a senha novamente. Caso o cliente informe três vezes senha incorreta, o cartão deverá ser bloqueado.
- Saque não autorizado: Se o saque não for aceito pelo sistema bancário, uma mensagem de erro é exibida e a operação é abortada.
- Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.
- Cancelamento: O cliente pode cancelar a transação a qualquer momento, enquanto o saque não for autorizado pelo sistema bancário.

Requisitos Relacionados: RF01, RN01, RNF01, RNF02[9]

Classes: Cliente, Conta, Cartão, Transação, Saque.

Figura 5.8 – Descrição do Caso de Uso *Efetuar Saque*.

[9] São as seguintes as descrições dos requisitos listados: RF01 – O sistema de caixa automático deve permitir que clientes efetuem saques em dinheiro; RN01 – Não devem ser permitidas transações que deixem a conta do cliente com saldo inferior ao de seu limite de crédito; RNF01 – O sistema de caixa automático deve estar integrado ao sistema bancário; RNF02 – As operações realizadas no caixa automático devem dar respostas em até 10s a partir da entrada de dados.

Em sistemas de médio a grande porte, pode ser útil considerar a fusão de casos de uso fortemente relacionados em um único caso de uso, contendo mais de um fluxo de eventos normal. Em muitos sistemas é necessário dar ao usuário a possibilidade de cancelar ou alterar dados de uma transação efetuada anteriormente com sucesso. Se cada uma dessas possibilidades for considerada como um caso de uso isolado, o número de casos de uso pode crescer demasiadamente, aumentando desnecessariamente a complexidade do modelo de casos de uso. Além disso, o fluxo de eventos normal de um caso de uso desse tipo tende a ser muito simples, não justificando documentar todo um conjunto de informações para adicionar apenas duas ou três linhas descrevendo os passos do caso de uso. Assim, em situações dessa natureza, é interessante considerar apenas um caso de uso, contendo diversos fluxos de eventos principais. Essa abordagem é bastante recomendada para casos de uso cadastrais, em que um único caso de uso inclui fluxos de eventos normais para criar, alterar, consultar e excluir entidades.

Fluxos de eventos normais podem ser descritos de diferentes maneiras, dependendo do nível de formalidade que se deseja para as descrições. Dentre os formatos possíveis, há dois principais:

- *Livre:* o fluxo de eventos normal é escrito na forma de um texto corrido, como no exemplo da Figura 5.8.
- *Enumerado:* cada passo do fluxo de eventos normal é numerado, de modo que possa ser referenciado nos fluxos de eventos alternativos ou em outros pontos do fluxo de eventos normal. A Figura 5.9 reapresenta o exemplo da Figura 5.8 neste formato. As seções iniciais foram omitidas por serem iguais às da Figura 5.8. Neste texto, advoga-se em favor do uso do formato enumerado.

Cada exceção deve ser tratada por um *fluxo alternativo de exceção*. Fluxos alternativos de exceção devem ser descritos contendo as seguintes informações (WAZLAWICK, 2004): um identificador, uma descrição sucinta da exceção que ocorreu, os passos para tratar a exceção (ações corretivas) e uma indicação de como o caso de uso retorna ao fluxo principal (se for o caso) após a execução das ações corretivas.

Quando um formato de descrição enumerado é utilizado, não é necessário colocar uma verificação como uma condicional no fluxo principal. Por exemplo, no caso da Figura 3.15, o passo 3 não deve ser escrito como "3. Se o cartão é válido, o caixa automático solicita que o cliente informe a senha.". Basta o fluxo alternativo, no exemplo, o fluxo 2a.

Ainda quando o formato de descrição enumerado é utilizado, o identificador da exceção deve conter a linha do fluxo de eventos principal (ou eventualmente de algum outro fluxo de eventos alternativo) no qual a exceção ocorreu e uma letra para identificar a própria exceção (WAZLAWICK, 2004), como ilustra o exemplo da Figura 5.9.

Uma informação que precisa estar presente na descrição de um fluxo de eventos de exceção diz respeito a como finalizar o tratamento de uma exceção. Wazlawick (2004) aponta quatro formas básicas para finalizar o tratamento de uma exceção:

- Voltar ao início do caso de uso, o que não é muito comum nem prático.
- Voltar ao início do passo em que ocorreu a exceção e executá-lo novamente. Esta é a situação mais comum.
- Voltar para algum um passo posterior. Esta situação ocorre quando as ações corretivas realizam o trabalho que o passo (ou a sequência de passos) posterior

deveria executar. Neste caso, é importante verificar se novas exceções não poderiam ocorrer.

- Abortar o caso de uso. Neste caso, não se retorna ao fluxo principal e o caso de uso não atinge seus objetivos.

Nome: Efetuar Saque

Fluxo de Eventos Normal

1. O cliente insere seu cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. O caixa automático solicita que o cliente informe a senha.
4. O cliente informa a senha.
5. O caixa automático envia os dados do cartão e da senha para o sistema bancário para validação.
6. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.
7. O cliente seleciona a opção saque.
8. O caixa automático solicita que seja informada a quantia.
9. O cliente informa a quantia a ser sacada.
10. O caixa automático envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.
11. As notas são preparadas e liberadas.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Senha incorreta:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

10a - Saque não autorizado: Uma mensagem de erro é exibida e a operação é abortada.

11a - Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.

1 a 9: Cancelamento: O cliente pode cancelar a transação, enquanto o saque não for autorizado pelo sistema bancário. A transação é abortada.

Figura 5.9 – Descrição do Caso de Uso *Efetuar Saque* – Formato Enumerado.

Além dos fluxos de exceção, há outro tipo de fluxo de eventos alternativo: os *fluxos variantes*. Fluxos variantes são considerados dentro da normalidade do caso de uso e indicam formas diferentes, mas igualmente normais, de se realizar uma certa porção de um caso de uso. Seja o caso de um sistema de um supermercado, mais especificamente um caso de uso para efetuar uma compra. Um passo importante desse caso de uso é a realização do pagamento, o qual pode se dar de três maneiras distintas: pagamento em dinheiro, pagamento em cheque, pagamento em cartão. Nenhuma dessas formas de pagamento constitui uma exceção. São todas maneiras diferentes, mas normais, de realizar um certo passo do caso de uso e, portanto, pode-se dizer que o fluxo principal possui três variações. A descrição de um fluxo variante deve

conter: um identificador, uma descrição sucinta do passo especializado e os passos enumerados, como ilustra a Figura 5.10.

Nome: Efetuar Compra

Fluxo de Eventos Normal

...
1. De posse do valor a ser pago, o atendente informa a forma de pagamento.
2. Efetuar o pagamento:
   2a. Em dinheiro
   2b. Em cheque
   2c. Em cartão
3. O pagamento é registrado.

Fluxos de Eventos Variantes

2a – Pagamento em Dinheiro:
   2a.1 – O atendente informa a quantia em dinheiro entregue pelo cliente.
   2a.2 – O sistema informa o valor do troco a ser dado ao cliente.

2b – Pagamento em Cheque:
   2b.1 – O atendente informa os dados do cheque, a saber: banco, agência, conta e valor.

2c – Pagamento em Cartão:
   2c.1 – O atendente informa os dados do cartão e o valor da compra.
   2.c.2 – O sistema envia os dados informados no passo anterior, junto com a identificação da loja para o serviço de autorização do Sistema de Operadoras de Cartão de Crédito.
   2c.3 – O Sistema de Operadoras de Cartão de Crédito autoriza a compra e envia o código da autorização.

Figura 5.10 – Descrição Parcial do Caso de Uso *Efetuar Compra* – com Variantes.

Por fim, em diversas situações, pode ser desnecessariamente trabalhoso especificar casos de uso segundo um formato completo, seja usando uma descrição dos fluxos de eventos no formato livre seja no formato enumerado. Para esses casos, um formato simplificado, na forma de uma tabela, pode ser usado. O formato tabular é normalmente empregado para casos de uso que possuem uma estrutura de interação simples, seguindo uma mesma estrutura geral, tais como casos de uso cadastrais (ou CRUD[10]) e consultas. Casos de uso cadastrais de baixa complexidade tipicamente envolvem inclusão, alteração, consulta e exclusão de entidades e seguem o padrão de descrição mostrado na Figura 5.11.

[10] CRUD – do inglês: Create, Read, Update and Delete; em português: Criar, Consultar, Atualizar e Excluir, ou seja, casos de uso que proveem as funções básicas de manipulação de dados de uma entidade de interesse do sistema.

Fluxos de Eventos Normais

Criar [Novo Objeto]
O [ator] informa os dados do [novo objeto], a saber: [atributos e associações do objeto]. Caso os dados sejam válidos, as informações são registradas.

Alterar Dados
O [ator] informa o [objeto] do qual deseja alterar dados e os novos dados. Os novos dados são validados e a alteração registrada.

Consultar Dados
O [ator] informa o [objeto] que deseja consultar. Os dados do [objeto] são apresentados.

Excluir [Objeto]
O [ator] informa o [objeto] que deseja excluir. Os dados do [objeto] são apresentados e é solicitada uma confirmação. Se a exclusão for confirmada, o [objeto] é excluído.

Fluxos de Eventos de Exceção

Incluir [Novo Objeto] / Alterar Dados
- Dados do [objeto] inválidos: uma mensagem de erro é exibida, solicitando correção da informação inválida.

Figura 5.11 – Padrão Típico de Descrição de Casos de Uso Cadastrais.

Assim, para simplificar a descrição de casos de uso cadastrais, recomenda-se utilizar o modelo tabular mostrado na Tabela 5.2. Quando essa tabela for empregada, estar-se-á assumindo que o caso de uso envolve os fluxos de eventos indicados (I para inclusão, A para alteração, C para consulta e E para exclusão), com a descrição base mostrada na Figura 5.11.

Tabela 5.2 – Modelo de Descrição de Casos de Uso Cadastrais

Caso de Uso	Ações Possíveis	Observações	Requisitos	Classes
<nome do caso de uso>	< I, A, C, E >			

A coluna **Observações** é usada para listar informações importantes relacionadas às ações, tais como os itens informados na inclusão, uma restrição a ser considerada para que a exclusão possa ser feita, uma informação que não pode ser alterada ou uma informação do objeto que não é apresentada na consulta. Deve-se indicar antes da observação a qual ação ela se refere ([I] para inclusão, [A] para alteração, [C] para consulta e [E] para exclusão).

As colunas **Requisitos** e **Classes** indicam, respectivamente, os requisitos que estão sendo (ou que devem ser) tratados pelo caso de uso e as classes do domínio do problema necessárias para a realização do caso de uso. O objetivo dessas colunas é manter a rastreabilidade dos casos de uso para requisitos e classes, respectivamente, de maneira análoga ao recomendado no formato completo.

A Tabela 5.3 ilustra a descrição de casos de usos cadastrais do subsistema Controle de Acervo de uma videolocadora.

Tabela 5.3 – Descrição de Casos de Uso Cadastrais – Controle de Acervo de Videolocadora.

Caso de Uso	Ações Possíveis	Observações	Requisitos	Classes
Cadastrar Filme	I, A, C, E	[I] Informar: título original, título em português, país, ano, diretores, atores, sinopse, duração, gênero, distribuidora, tipo de áudio (p.ex., Dolby Digital 2.0), idioma do áudio e idioma da legenda. [E] Não é permitida a exclusão de filmes que tenham itens associados. [E] Ao excluir um filme, devem-se excluir as reservas associadas.	RF9, RNF1	Filme, Distribuidora
Cadastrar Item	I, A, C, E	[I] Informar: filme, tipo de mídia, data de aquisição e número de série. [E] Não é permitido excluir um item que tenha locações associadas.	RF9, RNF1, RNF3	Item, Filme, TipoMidia
Cadastrar Distribuidora	I, A, C, E	[I] Informar: razão social, CNPJ, endereço, telefone e pessoa de contato. [E] Não é permitido excluir uma distribuidora que tenha filmes associados.	RF10, RNF1	Distribuidora
Cadastrar Tipo de Mídia	I, A, C, E	[I] Informar: nome e valor de locação. [E] Não é permitido excluir um tipo de mídia que tenha itens associados. [E] Ao excluir um tipo de mídia, devem-se excluir as reservas que especificam apenas esse tipo de mídia.	RF9, RNF1	TipoMidia

Para casos de uso de consulta mais abrangente do que a consulta de um único objeto (já tratada como parte dos casos de uso cadastrais), mas ainda de baixa complexidade (tais como consultas que combinam informações de vários objetos envolvendo filtros), sugere-se utilizar o formato tabular mostrado na Tabela 5.4.

Tabela 5.4 – Modelo de Descrição de Casos de Uso de Consulta

Caso de Uso	Observações	Requisitos	Classes
<nome do caso de uso>			

A coluna **Observações** deve ser usada para listar informações importantes relacionadas à consulta, tais como dados que podem ser informados para a pesquisa, totalizações feitas em relatórios etc.

As colunas **Requisitos** e **Classes** têm a mesma função de suas homônimas no modelo da Tabela 5.2, ou seja, indicam, respectivamente, os requisitos que estão sendo tratados (ou que devem ser) pelo caso de uso e as classes do domínio do problema necessárias para a realização do mesmo.

A Tabela 5.5 ilustra a descrição de um caso de usos de consulta do subsistema Controle de Acervo de uma videolocadora.

Tabela 5.5 – Descrição de Casos de Uso de Consulta – Controle de Acervo de Videolocadora .

Caso de Uso	Observações	Requisitos	Classes
Consultar Acervo	As consultas ao acervo poderão ser feitas informando uma (ou uma combinação) das seguintes informações: título (ou parte dele), original ou em português, gênero, tipo de mídia disponível, ator, diretor, nacionalidade e lançamentos.	RF11, RNF1, RNF2	Filme, Item, TipoMidia, Distribuidora

Descrevendo Informações Complementares

As descrições dos fluxos de eventos principal, variantes e de exceção são cruciais em uma descrição de casos de uso. Contudo, há outras informações complementares que são bastante úteis e, portanto, que devem ser levantadas e documentadas como, por exemplo: pré-condições, pós-condições, requisitos relacionados e classes relacionadas.

Conforme discutido anteriormente, um ator representa um papel que entidades físicas ou sociais podem desempenhar na interação com o sistema. Essas entidades físicas são tipicamente pessoas, dispositivos ou outros sistemas que são externos ao sistema em desenvolvimento. Muitas vezes, apenas o nome de um ator, como mostrado em um diagrama de casos de uso, pode ser pouco para um real entendimento do que representa esse ator. Assim, é importante que uma descrição sucinta dos atores seja feita. Uma vez que um mesmo ator pode atuar em vários casos de uso, a descrição dos atores não deve ser feita dentro da descrição dos casos de uso, mas separada, como uma seção específica dentro do Documento de Especificação de Requisitos. Inicialmente, atores podem ser documentados em uma tabela de duas colunas, contendo o nome e a descrição do ator.

Como atores interagindo com o sistema definem as interfaces do sistema com o mundo externo, pode ser útil adicionar informações sobre o perfil do ator nessa interação. Quando o ator é um ator humano, esse perfil indicaria as habilidades e a experiência do ator, informações valiosas para o projeto da interface com o usuário. Adicionalmente, pode-se incluir uma classificação segundo aspectos como nível de habilidade, nível na organização e membros em diferentes grupos. Pressman (2006) propõe uma classificação que considera três grupos principais:

- *Usuário novato*: conhece pouco a interface para utilizá-la eficientemente (conhecimento sintático; p.ex., não sabe como atingir uma funcionalidade desejada) e entende pouco as funções e objetivos do sistema (conhece pouco a semântica da aplicação) ou não sabe bem como usar computadores em geral;
- *Usuário conhecedor e esporádico*: possui um conhecimento razoável da semântica da aplicação, mas tem relativamente pouca lembrança dos mecanismos de interação providos pela interface (informações sintáticas necessárias para utilizar a interface);
- *Usuário conhecedor e frequente*: possui bom conhecimento tanto sintático quanto semântico e buscam atalhos e modos abreviados de interação.

Não são apenas os atores os interessados em um caso de uso. Outras pessoas ou unidades de uma organização podem ter interesse nos resultados do caso de uso. Seja o caso de uma locadora de automóveis. Em um caso de uso de locação, o único papel a interagir com o sistema é o de funcionário do atendimento. Contudo, o cliente, o setor de preparação de automóveis, a contabilidade, dentre outros, são também interessados neste caso de uso. Assim, mesmo que essas pessoas não interajam diretamente com o sistema para a realização do caso de uso, elas devem ser listadas como interessados. Deve-se lembrar que o sistema deve satisfazer os interesses de todos os envolvidos, direta ou indiretamente. Assim, na seção "Interessados e Interesses", deve-se listar os diversos interessados e uma descrição sucinta de seus interesses em relação à execução do caso de uso. Ao analisar esses interesses é possível, dentre outros, capturar regras de negócio e informações e descobrir ações que o sistema tem de realizar para atender a essas expectativas, tais como validações, atualizações e registros. (WAZLAWICK, 2004; COCKBURN, 2005).

Pré-condições estabelecem o que precisa ser verdadeiro antes de se iniciar um caso de uso. Pós-condições, por sua vez, estabelecem o que será verdadeiro após a execução do caso de uso. Pré-condições precisam ser verdadeiras para que o caso de uso possa ser iniciado. Não se deve confundi-las com exceções. Pré-condições não são testadas durante a execução do caso de uso (como ocorre com as condições que geram exceções). Ao contrário, elas são testadas antes de iniciar o caso de uso. Se a pré-condição é falsa, então não é possível executar o caso de uso. Para documentar as pré-condições, recomenda-se listar as condições que têm de ser satisfeitas na seção "Pré-condições". Pré-condições devem ser escritas como uma simples asserção sobre o estado do mundo no momento em que o caso de uso inicia (COCKBURN, 2005).

Muitas vezes, uma pré-condição para ser atendida requer que um outro caso de uso já executado tenha estabelecido essa pré-condição. Contudo, um erro bastante comum é escrever como uma pré-condição algo que frequentemente, mas não necessariamente, é verdadeiro (COCKBURN, 2005). Seja o caso de uma locadora de vídeos em que clientes em atraso não podem locar novos itens até que regularize suas pendências. Neste caso, uma pré-condição do tipo "cliente não está em atraso" como pré-condição de um caso de uso "efetuar locação" é inadequada. Observe que a identificação do cliente é parte do caso de uso efetuar locação e, portanto, não é possível garantir que o cliente não está em atraso antes de iniciar o caso de uso. Esta situação tem de ser tratada como uma exceção e não como uma pré-condição.

As seções de requisitos e classes relacionados são importantes para a gerência de requisitos. A primeira estabelece um rastro entre casos de uso e os requisitos de usuário documentados no Documento de Requisitos, permitindo, em um primeiro momento, analisar se algum requisito não foi tratado. Em um segundo momento, quando uma alteração em um requisito é solicitada, é possível usar essa informação para analisar o impacto da alteração. Para documentar os requisitos relacionados, recomenda-se listar os identificadores de cada um dos requisitos na seção de "Requisitos Relacionados".

A seção de classes relacionadas indica quais são as classes do modelo conceitual estrutural necessárias para a realização do caso de uso. Essa seção permite rastrear casos de uso para classes em vários níveis, uma vez que há uma grande tendência de as mesmas classes do modelo conceitual estrutural estarem presentes nos modelos de projeto e no código fonte. Para documentar as classes relacionadas, recomenda-se listar o nome de cada uma das classes envolvidas na seção de "Classes Relacionadas". Vale ressaltar que essa informação é tipicamente preenchida durante a modelagem conceitual estrutural ou até mesmo depois,

durante a elaboração de modelos de interação. A partir das informações de requisitos e classes relacionados, pode-se, por exemplo, construir matrizes de rastreabilidade.

5.6.4 Relacionamentos entre Casos de Uso

Para permitir uma modelagem mais apurada dos casos de uso em um diagrama, três tipos de relacionamentos entre casos de uso podem ser empregados. Casos de uso podem ser descritos como versões especializadas de outros casos de uso (relacionamento de **generalização/ especialização**); casos de uso podem ser incluídos como parte de outro caso de uso (relacionamento de **inclusão**); ou casos de uso podem estender o comportamento de um outro caso de uso (relacionamento de **extensão**). O objetivo desses relacionamentos é tornar um modelo mais compreensível, evitar redundâncias entre casos de uso e permitir descrever casos de uso em camadas. A seguir esses tipos de relacionamentos são abordados.

a) Inclusão

Uma associação de inclusão de um *caso de uso base* para um *caso de uso de inclusão* significa que o comportamento definido no caso de uso de inclusão é incorporado ao comportamento do caso de uso base. Ou seja, a relação de inclusão incorpora um caso de uso (o caso de uso incluído) dentro da sequência de comportamento de outro caso de uso (o caso de uso base) (BLAHA; RUMBAUGH, 2006; OLIVÉ, 2007).

Esse tipo de associação é útil para extrair comportamento comum a vários casos de uso em uma única descrição, de modo que esse comportamento não tenha de ser descrito repetidamente. O caso de uso de inclusão pode ou não ser passível de utilização isoladamente. Assim, ele pode ser apenas um fragmento de uma funcionalidade, não precisando ser uma transação completa. A parte comum é incluída por todos os casos de uso base que têm esse caso de uso de inclusão em comum e a execução do caso de uso de inclusão é análoga a uma chamada de subrotina (OLIVÉ, 2007).

Na UML, o relacionamento de inclusão entre casos de uso é mostrado como uma dependência (seta pontilhada) estereotipada com a palavra-chave *include*, partindo do caso de uso base para o caso de uso de inclusão, como ilustra a Figura 5.12.

Figura 5.12 – Associação de Inclusão na UML.

Uma associação de inclusão deve ser referenciada também na descrição do caso de uso base. O local em que esse comportamento é incluído deve ser indicado na descrição do caso de uso base, através de uma referência explícita à chamada ao caso de uso incluído. Assim, a descrição do fluxo de eventos (principal ou alternativo) do caso de uso base deve conter um passo que envolva a chamada ao caso de uso incluído, referenciada por "Incluir *nome do caso de uso incluído*". Para destacar referências de um caso de uso para outro, sugere-se que o nome do caso de uso referenciado seja sublinhado e escrito em itálico.

No exemplo do caixa automático, todos os três casos de uso têm em comum uma porção que diz respeito à validação inicial do cartão. Neste caso, um relacionamento de inclusão deve ser empregado, conforme mostra a Figura 5.13.

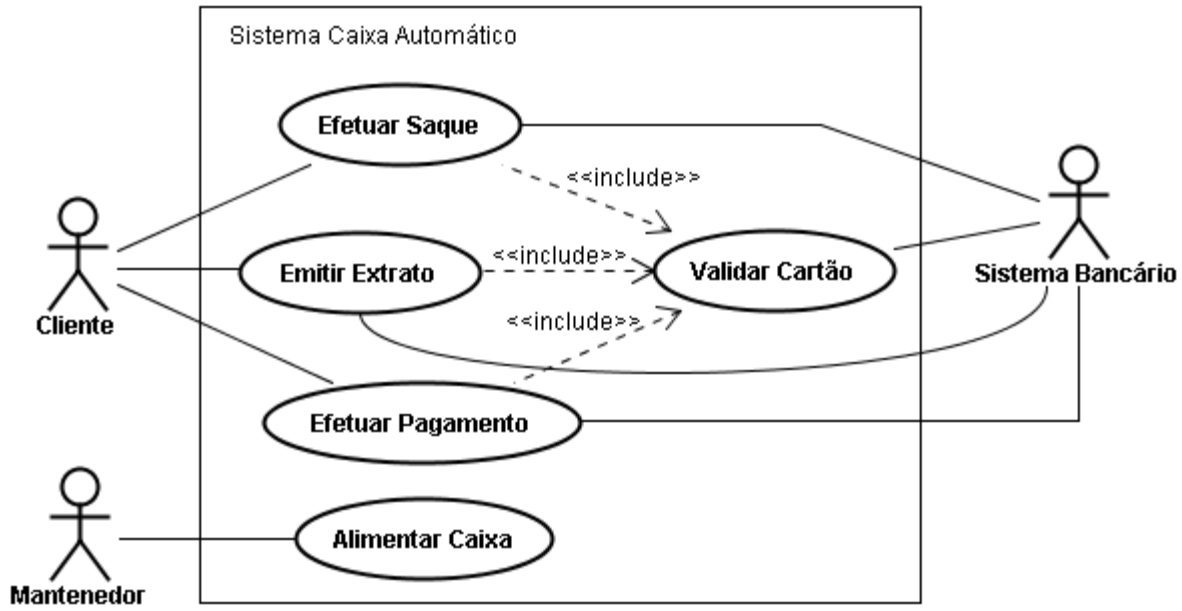

Figura 5.13 - Diagrama de Casos de Uso – Caixa Automático com Inclusão.

O caso de uso *Validar Cartão* extrai o comportamento descrito na Figura 5.14. Ao isolar este comportamento no caso de uso de *Validar Cliente*, o caso de uso *Efetuar Saque* passaria a apresentar a descrição mostrada na Figura 5.15.

Deve-se observar que não necessariamente o comportamento do caso de uso incluído precisa ser executado todas as vezes que o caso de uso base é realizado. Assim, é possível que a inclusão esteja associada a alguma condição. O caso de uso incluído é inserido em um local específico dentro da sequência do caso de uso base, da mesma forma que uma subrotina é chamada de um local específico dentro de outra subrotina (BLAHA; RUMBAUGH, 2006).

Nome: Validar Cartão

Fluxo de Eventos Normal

1. O cliente insere o cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. O caixa automático solicita que o cliente informe a senha.
4. O cliente informa a senha.
5. O caixa automático envia os dados do cartão e da senha para o sistema bancário para validação.
6. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Senha incorreta:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

1 a 5: Cancelamento: O cliente solicita o cancelamento da transação e a transação é abortada.

Figura 5.14 – Descrição do Caso de Uso *Validar Cartão*

Nome: Efetuar Saque

Fluxo de Eventos Normal

1. Incluir *Validar Cartão*.
2. O cliente seleciona a opção saque.
3. O caixa automático solicita que seja informada a quantia.
4. O cliente informa a quantia a ser sacada.
5. O caixa automático envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.
6. As notas são preparadas e liberadas.

Fluxos de Eventos de Exceção

5a - Saque não autorizado: Uma mensagem de erro é exibida e a operação é abortada.

6a - Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.

1 a 3: Cancelamento: O cliente pode cancelar a transação, enquanto o saque não for autorizado pelo sistema bancário. A transação é abortada.

Figura 5.15 – Descrição do Caso de Uso *Efetuar Saque* com inclusão.

Por fim, é importante frisar que não há um consenso sobre a possibilidade (ou não) de um caso de uso incluído poder ser utilizado isoladamente. Diversos autores, dentre eles Olivé (2007) e Blaha e Rumbaugh (2006), admitem essa possibilidade; outros não. Em (BOOCH; RUMBAUGH; JACOBSON, 2006), diz-se explicitamente que um "caso de uso incluído nunca

permanece isolado, mas é apenas instanciado como parte de alguma base maior que o inclui". Neste texto, admitimos a possibilidade de um caso de uso incluído poder ser utilizado isoladamente, pois isso permite representar situações em que um caso de uso chama outro caso de uso (como uma chamada de subrotina), mas este último pode também ser realizado isoladamente. A Figura 5.16 ilustra uma situação bastante comum, em que, ao se realizar um processo de negócio (no caso a reserva de um carro em um sistema de locação de automóveis), caso uma informação necessária para esse processo (no caso o cliente) não esteja disponível, ela pode ser inserida no sistema. Contudo, o cadastro da informação também pode ser feito dissociado do processo de negócio que o inclui (no caso, o cliente pode se cadastrar fora do contexto da reserva de um carro). Ao não se admitir a possibilidade de um caso de uso incluído poder ser utilizado isoladamente, não é possível modelar situações desta natureza, as quais são bastante frequentes.

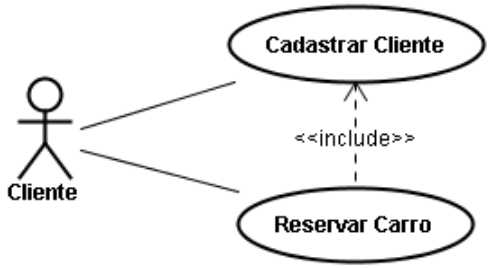

Figura 5.16 – Exemplo de Associação de Inclusão.

b) Extensão

Uma associação de extensão entre um *caso de uso de extensão* e um *caso de uso base* significa que o comportamento definido no caso de uso de extensão pode ser inserido dentro do comportamento definido no caso de uso base, em um local especificado indiretamente pelo caso de uso de extensão. A extensão ocorre em um ou mais pontos de extensão específicos definidos no caso de uso base. A extensão pode ser condicional. Neste caso, a extensão ocorre apenas se a condição é verdadeira quando o ponto de extensão especificado é atingido. O caso de uso base é definido de forma independente do caso de uso de extensão e é significativo independentemente do caso de uso de extensão (OLIVÉ, 2007; BOOCH; RUMBAUGH; JACOBSON, 2006).

Um caso de uso pode ter vários pontos de extensão e esses pontos são referenciados por seus nomes (BOOCH; RUMBAUGH; JACOBSON, 2006). O caso de uso base apenas indica seus pontos de extensão. O caso de uso de extensão especifica em qual ponto de extensão ele será inserido. Por isso, diz-se que o caso de uso de extensão especifica indiretamente o local onde seu comportamento será inserido.

A associação de extensão é como uma relação de inclusão olhada da direção oposta, em que a extensão se incorpora ao caso de uso base, em vez de o caso de uso base incorporar explicitamente a extensão. Ela conecta um caso de uso de extensão a um caso de uso base. O caso de uso de extensão é geralmente um fragmento, ou seja, ele não aparece sozinho como uma sequência de comportamentos. Além disso, na maioria das vezes, a relação de extensão possui uma condição associada e, neste caso, o comportamento de extensão ocorre apenas se a condição for verdadeira. O caso de uso base, por sua vez, precisa ser, obrigatoriamente, um caso de uso válido na ausência de quaisquer extensões (BLAHA; RUMBAUGH, 2006).

Na UML, a associação de extensão entre casos de uso é mostrada como uma dependência (seta pontilhada) estereotipada com a palavra chave *extend*, partindo do caso de

uso de extensão para o caso de uso base, como ilustra a Figura 5.20. Pontos de extensão podem ser indicados no compartimento da elipse do caso de uso, denominado "*extension points*" (pontos de extensão). Opcionalmente, a condição a ser satisfeita e a referência ao ponto de extensão podem ser mostradas por meio de uma nota[11] anexada à associação de extensão (OLIVÉ, 2007). Assim, no exemplo da Figura 5.17, o *Caso de Uso de Extensão 1* é executado quando o *ponto de extensão 1* do *Caso de Uso Base* for atingido, se a *condição* for verdadeira.

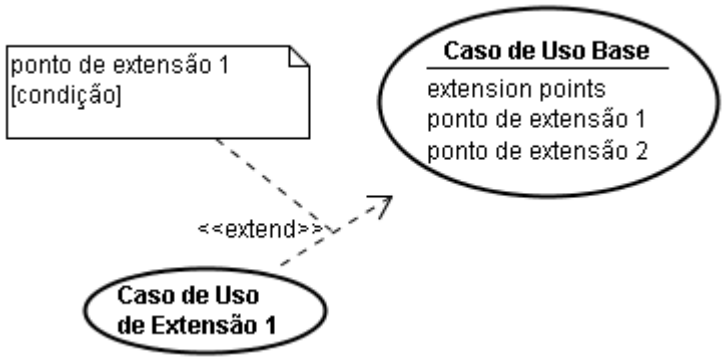

Figura 5.17 – Associação de Extensão na UML.

Uma importante diferença entre as associações de inclusão e extensão é que, na primeira o caso de uso base está ciente do caso de uso de inclusão, enquanto na segunda o caso de uso base não está ciente dos possíveis casos de uso de extensão (OLIVÉ, 2007).

Assim como no caso da inclusão, uma associação de extensão deve ser referenciada na descrição do caso de uso base. Neste caso, contudo, o caso de uso base apenas aponta o ponto de extensão, sem fazer uma referência explícita ao caso de uso de extensão. O local de cada um dos pontos de extensão deve ser indicado na descrição do caso de uso base, através de uma referência ao nome do ponto de extensão seguido de ": ponto de extensão". Assim, a descrição do fluxo de eventos (principal ou alternativo) do caso de uso base deve conter indicações explícitas para cada ponto de extensão.

No exemplo do caixa automático, suponha que se deseja coletar dados estatísticos sobre os valores das notas entregues nos saques, de modo a permitir alimentar o caixa eletrônico com as notas mais adequadas para saque. Poder-se-ia, então, estender o caso de uso *Efetuar Saque*, de modo que, quando necessário, outro caso de uso, denominado *Coletar Estatísticas de Notas*, contasse e acumulasse o tipo das notas entregues em um saque, conforme mostra a Figura 5.18. A Figura 5.19 mostra a descrição do caso de uso *Efetuar Saque* indicando o ponto de extensão *entrega do dinheiro*.

[11] Nota é o único item de anotação da UML. Notas são usadas para explicar partes de um modelo da UML. São comentários incluídos para descrever, esclarecer ou fazer alguma observação sobre qualquer elemento do modelo. Assim, uma nota é apenas um símbolo para representar restrições e comentários anexados a um elemento ou a uma coleção de elementos. Graficamente, uma nota é representada por um retângulo com um dos cantos com uma dobra de página, acompanhado por texto e anexada ao(s) elemento(s) anotados por meio de linha(s) pontilhada(s) (BOOCH; RUMBAUGH; JACOBSON, 2006). No exemplo da Figura 3.23, a nota está anexada ao relacionamento de extensão, adicionando-lhe informações sobre o ponto de extensão e a condição associados à extensão.

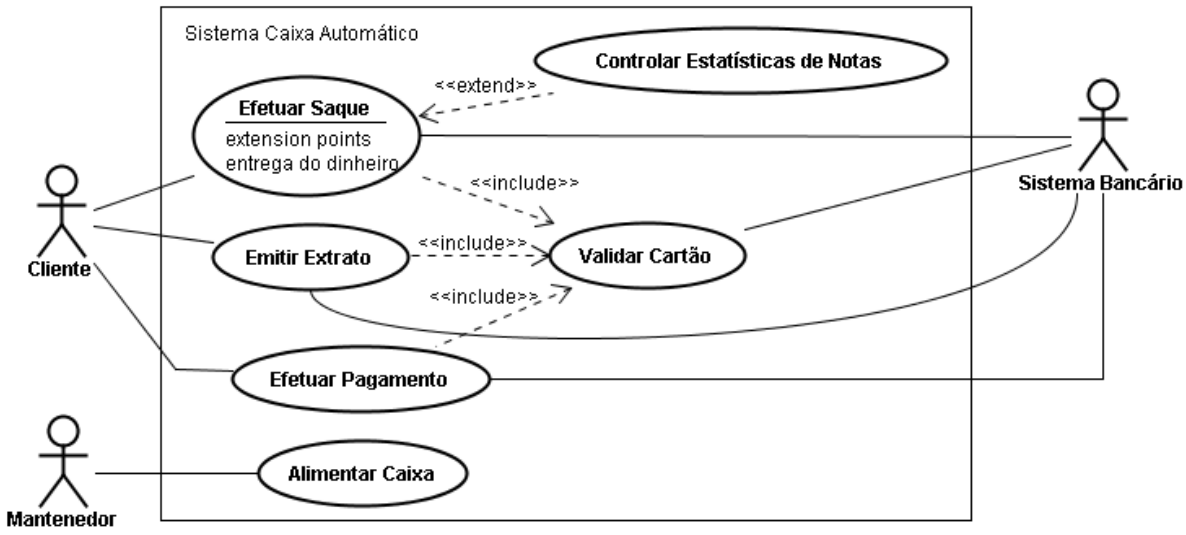

Figura 5.18 - Diagrama de Casos de Uso – Caixa Automático com Extensão.

Nome: Efetuar Saque

Fluxo de Eventos Normal

1. Incluir *Validar Cartão*.
2. O cliente seleciona a opção saque.
3. O caixa automático solicita que seja informada a quantia.
4. O cliente informa a quantia a ser sacada.
5. O caixa automático envia uma requisição para o sistema bancário para que seja efetuado um saque na quantia especificada.
6. As notas são preparadas.

entrega do dinheiro: ponto de extensão.

7. As notas são liberadas

Fluxos de Eventos de Exceção

5a - Saque não autorizado: Uma mensagem de erro é exibida e a operação é abortada.

6a - Não há dinheiro suficiente disponível no caixa eletrônico: Uma mensagem de erro é exibida e a operação é abortada.

1 a 3: Cancelamento: O cliente pode cancelar a transação, enquanto o saque não for autorizado pelo sistema bancário. A transação é abortada.

Figura 5.19 – Descrição do Caso de Uso *Efetuar Saque* com inclusão.

c) Generalização / Especialização

Um relacionamento de generalização / especialização entre um *caso de uso pai* e um *caso de uso filho* significa que o caso de uso filho herda o comportamento e o significado do caso de uso pai, acrescentando ou sobrescrevendo seu comportamento (OLIVÉ, 2007; BOOCH; RUMBAUGH; JACOBSON, 2006). Na UML, relacionamentos de generalização / especialização são representados como uma linha cheia direcionada com uma seta aberta (símbolo de herança), como ilustra a Figura 5.20.

Figura 5.20 – Associação de Generalização / Especialização entre Casos de Uso na UML.

Voltando ao exemplo do sistema de caixa automático, suponha que haja duas formas adotadas para se validar o cartão: a primeira através de senha, como descrito anteriormente, e a segunda por meio de análise da retina do cliente. Neste caso, poderiam ser criadas duas especializações do caso de uso *Validar Cliente*, como mostra a Figura 5.21.

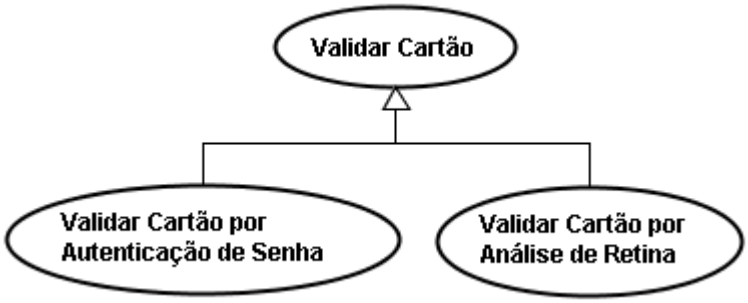

Figura 5.21 – Exemplo de Generalização / Especialização entre Casos de Uso

A descrição do caso de uso pai teria de ser generalizada para acomodar diferentes tipos de validação. Esses tipos de validação seriam especializados nas descrições dos casos de uso filhos. A Figura 5.22 mostra as descrições desses três casos de uso.

A generalização / especialização é aplicável quando um caso de uso possui diversas variações. O comportamento comum pode ser modelado como um caso de uso abstrato e especializado para as diferentes variações (BLAHA; RUMBAUGH, 2006). Contudo, avalie se não fica mais simples e direto descrever essas variações como fluxos alternativos variantes na descrição de casos de uso. Quando forem poucas e pequenas as variações, muito provavelmente será mais fácil capturá-las na descrição, ao invés de criar hierarquias de casos de uso. A Figura 5.23 mostra uma solução análoga à da Figura 5.22, sem usar, no entanto, especializações do caso de uso.

Nome: Validar Cartão

Fluxo de Eventos Normal

1. O cliente insere o cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. O caixa automático solicita informação para identificação do cliente.
4. O cliente informa sua identificação.
5. O caixa automático envia os dados do cartão e da identificação para o sistema bancário para validação.
6. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Dados de Identificação Incorretos:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

1 a 5: Cancelamento: O cliente solicita o cancelamento da transação e a transação é abortada.

Nome: Validar Cartão por Análise de Retina

Fluxo de Eventos Normal

3. O caixa automático solicita que o cliente se posicione corretamente para a captura da imagem da retina.
4. O caixa automático retira uma foto da retina do cliente.
5. O caixa automático envia os dados do cartão e a foto da retina para o sistema bancário para validação.

Nome: Validar Cartão por Autenticação de Senha

Fluxo de Eventos Normal

3. O caixa automático solicita a senha.
4. O cliente informa a senha.
5. O caixa automático envia os dados do cartão e a senha para o sistema bancário para validação.

Figura 5.22 – Descrição do Caso de Uso *Validar Cartão* e suas Especializações.

Nome: Validar Cartão

Fluxo de Eventos Normal

1. O cliente insere o cartão no caixa automático.
2. O caixa automático analisa o cartão e verifica se ele é aceitável.
3. Validar cartão.
4. O caixa automático solicita que o cliente informe o tipo de transação a ser efetuada.

Fluxos de Eventos Variantes

3a – Validar cartão por autenticação de senha:

3a.1 – O caixa automático solicita a senha.

3a.2 – O cliente informa a senha.

3a.3 – O caixa automático envia os dados do cartão e a senha para o sistema bancário para validação.

3b – Validar cartão por análise de retina:

3b.1 – O caixa automático solicita que o cliente se posicione corretamente para a captura da imagem da retina.

3b.2 – O caixa automático retira uma foto da retina do cliente.

3b.3 – O caixa automático envia os dados do cartão e a foto da retina para o sistema bancário para validação.

Fluxos de Eventos de Exceção

2a – O cartão não é aceitável: Se o cartão não é aceitável, seja porque sua tarja magnética não é passível de leitura seja porque é de um tipo incompatível, uma mensagem de erro de leitura é mostrada e se retorna ao passo 1.

5a – Dados de Identificação Incorretos:

5a.1 – 1ª e 2ª tentativas: Uma mensagem de erro é mostrada para o cliente. Retornar ao passo 3.

5a.2 – 3ª tentativa: bloquear o cartão e abortar a transação.

1 a 5: Cancelamento: O cliente solicita o cancelamento da transação e a transação é abortada.

Figura 5.23 – Descrição do Caso de Uso *Validar Cartão* com Variantes.

Diretrizes para o Uso dos Tipos de Relacionamentos entre Casos de Uso

Os relacionamentos entre casos de uso devem ser utilizados com cuidado para evitar a introdução de complexidade desnecessária. As seguintes orientações são úteis para ajudar a decidir quando usar relacionamentos entre casos de uso em um diagrama de casos de uso:

- A inclusão é tipicamente aplicável quando se deseja capturar um fragmento de comportamento comum a vários casos de uso. Na maioria das vezes, o caso de uso de inclusão é uma atividade significativa, mas não como um fim em si mesma (BLAHA; RUMBAUGH, 2006). Ou seja, o caso de uso de inclusão não precisa ser

uma transação completa. Um relacionamento de inclusão é empregado quando há uma porção de comportamento que é similar ao longo de um ou mais casos de uso e não se deseja repetir a sua descrição. Para evitar redundância e assegurar reúso, extrai-se essa descrição e se compartilha a mesma entre diferentes casos de uso. Desta maneira, utiliza-se a inclusão para evitar ter de descrever o mesmo fragmento de comportamento várias vezes, capturando o comportamento comum em um caso de uso próprio (BOOCH; RUMBAUGH; JACOBSON, 2006).

- Não se deve utilizar o relacionamento de generalização / especialização para compartilhar fragmentos de comportamento. Para este propósito, deve-se usar a relação de inclusão (BLAHA; RUMBAUGH, 2006).
- A relação de extensão é bastante útil em situações em que se pode definir um caso de uso significativo com recursos adicionais. O comportamento básico é capturado no caso de uso base e os recursos adicionais nos casos de uso de extensão. Use a relação de extensão quando o sistema puder ser usado em diferentes configurações, algumas com os recursos adicionais e outras sem eles (BLAHA; RUMBAUGH, 2006).

5.6.5 Trabalhando com Casos de Uso

Para se utilizar a modelagem de casos de uso para o refinamento de requisitos de cliente em requisitos de sistema é necessário proceder um exame detalhado do processo de negócio a ser apoiado pelo sistema. Assim, atividades de levantamento de requisitos, como entrevistas, observação, workshop de requisitos e cenários, dentre outras, certamente acontecerão em paralelo com a modelagem de casos de uso.

Uma boa maneira de trabalhar com casos de uso consiste em, a partir dos requisitos funcionais de usuário descritos no Documento de Definição de Requisitos, procurar derivar casos de uso. Este é apenas um ponto de partida, uma vez que vários casos de uso podem ser derivados a partir de um mesmo requisito funcional de usuário.

Uma maneira complementar de identificar casos de uso é começar pela identificação de atores. Cada ator deve ter um propósito único e coerente, o qual deve ser descrito e documentado. Para cada ator identificado, pode-se, então, levantar quais são as funcionalidades por ele requeridas, listando-as na forma de casos de uso. As funcionalidades devem ser extraídas do Documento de Definição de Requisitos. Cada caso de uso deve representar uma transação completa que seja algo de valor para os atores envolvidos. Contudo, antes de identificar atores e casos de uso, é necessário determinar claramente os limites do sistema. Sem deixar claro quais são os limites do sistema, é muito difícil identificar atores ou casos de uso (BLAHA; RUMBAUGH, 2006).

Uma vez identificados atores e casos de uso, pode-se elaborar uma versão preliminar do diagrama de casos de uso. Vale lembrar que, até mesmo para sistemas de pequeno porte, é útil trabalhar com subsistemas, procurando agrupar casos de uso em pacotes. Assim, é importante construir também diagramas de pacotes à medida que os casos de uso vão sendo agrupados.

Uma vez identificados e agrupados os casos de uso, é interessante fazer uma descrição sucinta de seu propósito. Não se deve partir diretamente para os detalhes, descrevendo fluxos de eventos e outras informações. Fazendo apenas uma descrição sucinta, é possível levar mais rapidamente os casos de uso à discussão com os clientes e usuários, permitindo identificar melhor quais são efetivamente os casos de uso a serem contemplados pelo sistema. Além disso,

pode-se dividir o trabalho, designando diferentes analistas para trabalhar com casos de uso (ou pacotes) específicos.

Somente então se deve passar para a descrição detalhada dos casos de uso. Inicialmente, o foco deve ser no fluxo de eventos principal, ou seja, aquele em que tudo dá certo na interação. Depois de descrever o fluxo de eventos normal, deve-se analisar de forma crítica cada passo desses fluxos de eventos, procurando verificar o que pode dar errado (WAZLAWICK, 2004), bem como se devem investigar maneiras alternativas, ainda normais, de realizar o caso de uso, permitindo a identificação de fluxos variantes. A partir da identificação de possíveis exceções e variações, deve-se trabalhar na descrição de fluxos alternativos de exceção (descrevendo procedimentos para contornar os problemas) e variantes (descrevendo maneiras alternativas de realizar com sucesso uma certa porção do caso de uso).

Assim, uma maneira adequada para trabalhar com casos de uso consiste em identificá-los, modelá-los e descrevê-los com diferentes níveis de precisão. O seguinte processo resume a abordagem descrita anteriormente:

- Listar atores e casos de uso relacionados: neste momento, é montada apenas uma lista dos atores associados aos casos de uso de seu interesse. Apenas o nome do caso de uso é indicado.
- Para cada caso de uso identificado, fazer uma descrição sucinta do mesmo. Essa descrição deve conter, em essência, o objetivo do caso de uso.
- Elaborar um ou mais diagramas de casos de uso.
- Revisar a exatidão e a completude do conjunto de casos de uso com os interessados e priorizar os casos de uso.
- Definir o formato de descrição de caso de uso a ser usado (e o correspondente modelo de descrição de caso de uso a ser adotado) para cada caso de uso.
- Definir os fluxos de eventos principais a serem comportados pelo caso de uso: de maneira análoga ao passo 1, apenas uma lista dos fluxos de eventos principais é elaborada, sem descrevê-los ainda.
- Descrever cada um dos fluxos principais de eventos do caso de uso, segundo o modelo de descrição de caso de uso estabelecido no passo anterior. De acordo com o modelo predefinido, levantar informações adicionais como pré-condições e requisitos relacionados.
- Identificar fluxos alternativos: neste momento, é levantada apenas uma lista de exceções e variações que podem ocorrer no fluxo principal de eventos do caso de uso, sem, no entanto, definir como o sistema deve tratá-las.
- Descrever os passos dos fluxos alternativos: descrever como o sistema deve responder a cada exceção ou como ele deve funcionar em cada variação.

Vale ressaltar que a descrição de casos de uso na fase de análise de requisitos deve ser feita sem considerar a tecnologia de interface. Neste momento não interessa saber a forma das interfaces do sistema, mas quais informações são trocadas entre o sistema e o ambiente externo (atores). O analista deve procurar abstrair a tecnologia e se concentrar na essência das informações trocadas. Assim, diz-se que a descrição de caso de uso na fase de análise é uma descrição essencial. A tecnologia de interface será objeto da fase de projeto do sistema. Agindo

dessa maneira, abre-se caminho para se pensar em diferentes alternativas de interfaces durante o projeto do sistema (WAZLAWICK, 2004).

5.7 Modelagem Conceitual Estrutural

O modelo conceitual estrutural de um sistema tem por objetivo descrever as informações que esse sistema deve representar e gerenciar. Modelos conceituais devem ser concebidos com foco no domínio do problema e não no domínio da solução e, por conseguinte, um modelo conceitual estrutural é um artefato do domínio do problema e não do domínio da solução.

As informações a serem capturadas em um modelo conceitual estrutural devem existir independentemente da existência de um sistema computacional para tratá-las. Ele deve ser independe da solução computacional a ser adotada para resolver o problema e deve conter apenas os elementos de informação referentes ao domínio do problema em questão. Elementos da solução, tais como interfaces, formas de armazenamento e comunicação, devem ser tratados apenas na fase de projeto (WAZLAWICK, 2004).

Uma vez que requisitos não-funcionais de produto (atributos de qualidade) são inerentes à solução computacional, geralmente eles não são tratados na modelagem conceitual. Ou seja, não se consideram elementos de informação para tratar aspectos como desempenho, segurança de acesso, confiabilidade, formas de armazenamento etc. Esses atributos de qualidade do produto são considerados posteriormente, na fase de projeto.

Os elementos de informação básicos da modelagem conceitual estrutural são os tipos de entidades e os tipos de relacionamentos. A identificação de quais os tipos de entidades e os tipos de relacionamentos que são relevantes para um particular sistema de informação é uma meta crucial da modelagem conceitual (OLIVÉ, 2007).

Na modelagem conceitual segundo o paradigma orientado a objetos, tipos de entidades são modelados como classes. Tipos de relacionamentos são modelados como atributos e associações. Assim, o propósito da modelagem conceitual estrutural orientada a objetos é definir as classes, atributos e associações que são relevantes para tratar o problema a ser resolvido. Para tal, as seguintes tarefas devem ser realizadas:

- Identificação de Classes
- Identificação de Atributos e Associações
- Especificação de Hierarquias de Generalização/Especialização

É importante notar que essas atividades são dependentes umas das outras e que, durante o desenvolvimento, elas são realizadas, tipicamente, de forma paralela e iterativa, sempre visando ao entendimento do domínio do problema, desconsiderando aspectos de implementação.

5.7.1 Identificação de Classes

Classificação é o meio pelo qual os seres humanos estruturam a sua percepção do mundo e seu conhecimento sobre ele. Sem ela, não é possível nem entender o mundo a nossa volta nem agir sobre ele. Classificação assume a existência de tipos e de objetos a serem classificados nesses tipos. Classificar consiste, então, em determinar se um objeto é ou não uma instância de um tipo. A classificação nos permite estruturar conhecimento sobre as coisas em dois níveis:

tipos e instâncias. No nível de tipos, procuramos encontrar as propriedades comuns a todas as instâncias de um tipo. No nível de instância, procuramos identificar o tipo do qual o objeto é uma instância e os valores particulares das propriedades desse objeto (OLIVÉ, 2007).

Tipos de entidade são um dos mais importantes elementos em modelos conceituais. Definir os tipos de entidade relevantes para um particular sistema de informação é uma tarefa crucial na modelagem conceitual. Um tipo de entidade pode ser definido como um tipo cujas instâncias em um dado momento são objetos individuais identificáveis que se consideram existir no domínio naquele momento. Um objeto pode ser instância de vários tipos ao mesmo tempo (OLIVÉ, 2007). Por exemplo, seja o caso dos tipos *Estudante* e *Funcionário* em um sistema de uma universidade. Uma mesma pessoa, por exemplo João, pode ser ao mesmo tempo um estudante e um funcionário dessa universidade.

Na orientação a objetos, tipos de entidade são representados por classes, enquanto as instâncias de um tipo de entidade são objetos. Assim, uma atividade crucial da modelagem conceitual estrutural segundo o paradigma orientado a objetos (OO) é a identificação de classes. Na UML, classes são representadas por um retângulo com três compartimentos: o compartimento superior é relativo ao nome da classe; o compartimento do meio é dedicado à especificação dos atributos da classe; e o compartimento inferior é dedicado à especificação das operações da classe. A Figura 5.24 mostra a notação de classe na UML.

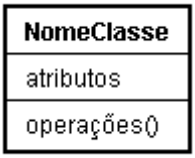

Figura 5.24 – Notação de Classes na UML.

Para nomear classes, sugere-se iniciar com um substantivo no singular, o qual pode ser combinado com complementos ou adjetivos, omitindo-se preposições. O nome da classe deve ser iniciado com letra maiúscula, bem como os nomes dos complementos, sem dar um espaço em relação à palavra anterior. Acentos não devem ser utilizados. Ex.: Cliente, PessoaFisica, ItemPedido.

Tomando por base os requisitos iniciais do usuário e, sobretudo, o modelo de casos de uso, é possível iniciar o trabalho de modelagem da estrutura do sistema. Esse trabalho começa com a descoberta de quais classes devem ser incluídas no modelo. O cerne de um modelo OO é exatamente o seu conjunto de classes.

Durante a análise de requisitos, tipicamente o analista estuda, filtra e modela o domínio do problema. Dizemos que o analista "filtra" o domínio, pois apenas uma parte desse domínio fará parte das responsabilidades do sistema. Assim, um domínio de problemas pode incluir várias informações, mas as responsabilidades de um sistema nesse domínio podem incluir apenas uma pequena parcela deste conjunto.

As classes de um modelo representam a expressão inicial do sistema. As atividades subsequentes da modelagem estrutural buscam obter uma descrição cada vez mais detalhada, em termos de associações e atributos. Contudo, deve-se observar que, à medida que atributos e associações vão sendo identificados, se ganha maior entendimento a respeito do domínio e naturalmente novas classes surgem. Assim, as atividades da modelagem conceitual são iterativas e com alto grau de paralelismo, devendo ser realizadas concomitantemente.

Conforme apontado anteriormente, dois importantes insumos para a atividade de identificação de classes são o Documento de Requisitos e o Modelo de Casos de Uso. Uma maneira bastante prática e eficaz de trabalhar a identificação de classes consiste em olhar esses dois documentos, em especial a descrição do minimundo e as descrições de casos de uso, à procura de classes.

Diversos autores, dentre eles Jacobson (1992) e Wazlawick (2004), sugerem que uma boa estratégia para identificar classes consiste em ler esses documentos procurando por substantivos. Esses autores argumentam que uma classe é, tipicamente, descrita por um nome no domínio e, portanto, aprender sobre a terminologia do domínio do problema é um bom ponto de partida. Ainda que um bom ponto de partida, essa heurística é ainda muito vaga. Se o analista segui-la fielmente, muito provavelmente ele terá uma extensa lista de potenciais classes, sendo que muitas delas podem, na verdade, se referir a atributos de outras classes. Além disso, pode ser que importantes classes não sejam capturadas, notadamente aquelas que se referem ao registro de eventos de negócio, uma vez que esses eventos muitas vezes são descritos na forma de verbos. Seja o seguinte exemplo de uma descrição de um domínio de locação de automóveis: "clientes locam carros". Seriam consideradas potenciais classes: *Cliente* e *Carro*. Contudo, a locação é um evento de negócio importante que precisa ser registrado e, usando a estratégia de identificar classes a partir de substantivos, *Locação* não entraria na lista de potenciais classes.

Assim, neste texto sugere-se que, ao examinar documentos de requisitos e modelos de casos de uso, os seguintes elementos sejam considerados como candidatos a classes:

- Agentes: entidades do domínio do problema que têm a capacidade de agir com intenção de atingir uma meta. Em sistemas de informação, há dois tipos principais de agentes: os agentes físicos (tipicamente pessoas) e os agentes sociais (organizações, unidades organizacionais, sociedades etc.). Em relação às pessoas, deve-se olhar para os papéis desempenhados pelas diferentes pessoas no domínio do problema.
- Objetos: entidades sem a capacidade de agir, mas que fazem parte do domínio de informação do problema. Podem ser também classificados em físicos (p.ex., carros, livros, imóveis) e sociais (p.ex., cursos, disciplinas, leis). Entretanto, há também outros tipos de objetos, tais como objetos de caráter descrito usado para organizar e descrever outros objetos de um domínio (p.ex., modelos de carro), algumas vezes denominados objetos de especificação. Objetos sociais e de descrição (ou especificação) tendem a ser coisas menos tangíveis, mas são tão importantes para a modelagem conceitual quanto os objetos físicos e, portanto, palpáveis.
- Eventos: representam a ocorrência de ações no domínio do problema que precisam ser registradas e lembradas pelo sistema. Eventos acontecem no tempo e, portanto, a representação de eventos normalmente envolve a necessidade de registrar, dentre outros, quando o evento ocorreu. Deve-se observar que muitos eventos ocorrem no domínio do problema, mas grande parte deles não precisa ser lembrada. Para capturar os eventos que precisam ser lembrados e, portanto, registrados, devem-se focalizar os principais eventos de negócio do domínio do problema. Assim, em um sistema de locação de automóveis, são potenciais classes de eventos: *Locação*, *Devolução* e *Reserva*. Por outro lado, a ocorrência de eventos cadastrais, tais como os cadastros de clientes e carros, tende a ser de pouca importância, não sendo necessário lembrar a ocorrência desses eventos.

Seja qual for a estratégia usada para identificar classes, é sempre importante que o analista tenha em mente os objetivos do sistema durante a modelagem conceitual. Não se devem

representar informações irrelevantes para o sistema e, portanto, a relevância para o sistema é o principal critério a ser adotado para decidir se um determinado elemento deve ou não ser incluído no modelo conceitual estrutural do sistema.

O resultado principal da atividade de identificação de classes é a obtenção de uma lista de potenciais classes para o sistema em estudo. Um modelo conceitual estrutural para uma aplicação complexa pode ter dezenas de classes e, portanto, pode ser necessário definir uma representação concisa capaz de orientar um leitor em um modelo desta natureza. O agrupamento de classes em subsistemas serve basicamente a este propósito, podendo ser útil também para a organização de grupos de trabalho em projetos extensos. Através da identificação e agrupamento de classes em subsistemas, é possível controlar a visibilidade do leitor e, assim, tornar o modelo mais compreensível. Assim, da mesma maneira que casos de uso são agrupados em pacotes, classes também devem ser.

Quando uma coleção de classes colabora entre si para realizar um conjunto coeso de responsabilidades (casos de uso), elas podem ser vistas como um subsistema. Assim, um subsistema é uma abstração que provê uma referência para mais detalhes em um modelo de análise, incluindo tanto casos de uso quanto classes. O agrupamento de classes em subsistemas permite apresentar o modelo global em uma perspectiva mais alta. Esse nível ajuda o leitor a rever o modelo, bem como constitui um bom critério para organizar a documentação.

Uma vez identificadas as potenciais classes, deve-se proceder uma avaliação para decidir o que efetivamente considerar ou rejeitar. Conforme discutido anteriormente, a relevância para o sistema deve ser o critério principal. Além desse critério, os seguintes também devem ser considerados nessa avaliação:

- *Estrutura complexa:* o sistema precisa tratar informações sobre os objetos da classe? Tipicamente, uma classe deve ter, pelo menos, dois atributos. Se uma classe apresentar apenas um atributo, avalie se não é melhor tratá-la como um atributo de uma classe existente[12].
- *Atributos e associações comuns*: os atributos e as associações da classe devem ser aplicáveis a todas as suas instâncias, isto é, a todos os objetos da classe.
- *Classes não redundantes:* duas classes são redundantes quando elas têm sempre a mesma população[13]. Seja o exemplo de um modelo conceitual que tenha as classes Pessoa e Funcionário. Se o sistema está interessado apenas nas pessoas empregadas na organização (ou seja, funcionários), então a população dessas duas classes será sempre a mesma. A introdução de classes redundantes afeta e simplicidade do modelo e, portanto, um modelo conceitual não deve incluir classes redundantes.
- *Existência de instâncias:* toda classe deve possuir uma população não vazia. Uma potencial classe que possui uma única instância também não deve ser considerada uma classe. Tipicamente uma classe possui várias instâncias e a população da classe varia ao longo do tempo.

[12] Uma classe que possui um único atributo, mas várias associações, também satisfaz a esse critério.
[13] A população de uma classe em um dado momento é o conjunto de instâncias que existem no domínio naquele momento (OLIVÉ, 2007).

5.7.2 - Identificação de Atributos e Associações

Conforme apontado anteriormente, uma classe típica de um modelo conceitual estrutural deve apresentar estrutura complexa. A estrutura de uma classe corresponde a seus atributos e associações.

Conceitualmente, não há diferença entre atributos e associações. Atributos são, na verdade, tipos de relacionamentos binários. Em um tipo de relacionamento binário, há dois participantes. Em alguns tipos de relacionamentos, esses participantes são considerados "colegas", porque eles desempenham funções análogas e nenhum deles é subordinado ao outro. Seja o caso do tipo de relacionamento "*aluno cursa um curso*". Um aluno não pode cursar sem haver um curso, bem como um curso não pode ser cursado se não houver um aluno. A ordem dos participantes no modelo não implica uma relação de prioridade ou subordinação entre eles (OLIVÉ, 2007). Na orientação a objetos, esse tipo de relacionamento é modelado como uma associação.

Entretanto, há alguns tipos de relacionamentos nos quais usuários e analistas consideram um participante como sendo uma característica do outro. Seja o exemplo do tipo de relacionamento "*filme possui gênero*". Alguém pode argumentar que o participante *gênero* é uma característica de filme e, portanto, subordinado a este. Esse tipo de relacionamento é modelado como um atributo. Assim, um atributo é um tipo de relacionamento binário em que um participante é considerado uma característica de outro. Por conseguinte, um atributo é igual a uma associação, exceto pelo fato de usuários e analistas adicionarem a interpretação que um dos participantes é subordinado ao outro (OLIVÉ, 2007).

De uma perspectiva mais prática, atributos podem ser vistos como informações alfanuméricas ligadas a um conceito. Associações, por sua vez, consistem em um tipo de informação que liga diferentes conceitos entre si (WAZLAWICK, 2004). Atributos ligam classes do domínio do problema a tipos de dados.

Tipos de dados podem ser primitivos ou específicos de domínio. Os tipos de dados primitivos são aplicáveis aos vários domínios e sistemas, tais como strings, datas, inteiros e reais, e são considerados como sendo predefinidos. Os tipos de dados específicos de um domínio de aplicação, por outro lado, precisam ser definidos. São exemplos de tipos de dados específicos: CPF, ISBN de livros, endereço etc.

Neste texto são considerados os seguintes tipos de dados primitivos:

- *String*: cadeia de caracteres;
- *boolean*: admite apenas os valores verdadeiro e falso;
- *Integer* (ou *int*): números inteiros;
- *Float* (ou *float*): números reais;
- *Currency*: valor em moeda (reais, dólares etc.);
- *Date*: datas, com informação de dia, mês e ano;
- *Time*: horas em um dia, com informação de hora, minuto e segundo;
- *DateTime*: combinação dos dois anteriores;
- *YearMonth*: informação de tempo contendo apenas mês e ano;
- *Year*: informação de tempo contendo apenas ano.

a) Atributos

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

Conforme discutido anteriormente, atributos possuem um tipo de dado, que pode ser primitivo ou específico de domínio. Ao identificar um atributo como sendo relevante, deve-se definir qual o seu tipo de dado. Caso nenhum dos tipos de dados primitivos se aplique, deve-se definir, então, um tipo de dados específico. Por exemplo, em domínios que lidem com livros, é necessário definir o tipo ISBN[14], cujas instâncias são ISBNs válidos. Em domínios que lidem com pessoas físicas e jurídicas, CPF e CNPJ também devem ser definidos como tipos de dados específicos. Usar um tipo de dados primitivo nestes casos, tais como *String* ou *int*, é insuficiente, pois não são quaisquer cadeias de caracteres ou números que se caracterizam como ISBNs, CPFs ou CNPJs válidos.

Tipos de dados específicos podem apresentar propriedades. Por exemplo, CPF é um número de 11 dígitos, que pode ser dividido em duas partes: os 9 primeiros dígitos e os dois últimos, que são dígitos verificadores.

Um tipo de dados especial é a enumeração. Na enumeração, os valores do tipo são enumerados explicitamente na forma de literais, como é o caso do tipo *DiaSemana*, que é tipicamente definido como um tipo de dados compreendendo sete valores: Segunda, Terça, Quarta, Quinta, Sexta, Sábado e Domingo. É importante observar que tipos de dados enumerados só devem ser usados quando se sabe à priori quais são os seus valores e eles são fixos. Assim, são bons candidatos a tipos enumerados informações como sexo (M/F), estado civil etc.

Tipos de dados geralmente não são representados graficamente em um modelo conceitual estrutural, de modo a torná-lo mais simples. Na maioria das situações, basta descrever os tipos de dados específicos de domínio no Dicionário de Dados do Projeto. Contudo, se necessário, eles podem ser representados graficamente usando o símbolo de classe estereotipado com a palavra chave <<*dataType*>>. Tipos enumerados também podem ser representados usando o símbolo de classe, mas com o estereótipo <<*enumeration*>>, sendo que ao invés de apresentar atributos de um tipo de dados, enumeram-se os valores possíveis da enumeração. A Figura 5.25 ilustra a notação de tipos de dados na UML.

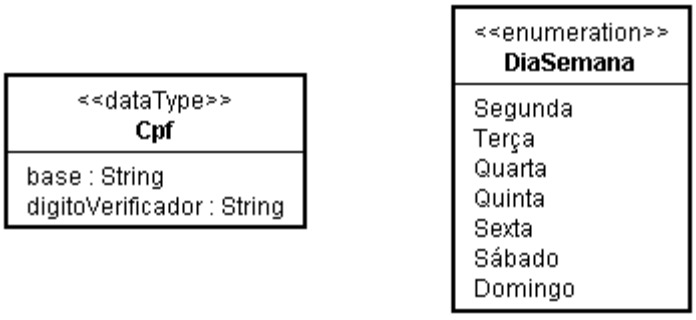

Figura 5.25 – Notação de Tipos de Dados na UML.

[14] O ISBN - *International Standard Book Number* - é um sistema internacional padronizado que identifica numericamente os livros segundo o título, o autor, o país, a editora, individualizando-os inclusive por edição. Utilizado também para identificar software, seu sistema numérico pode ser convertido em código de barras.

Uma dúvida típica e recorrente na modelagem estrutural é se um determinado item de informação deve ser modelado como uma classe ou como um atributo. Para que o item seja considerado uma classe, ele tem de passar nos critérios de inclusão no modelo discutidos na seção anterior. Entretanto, há alguns itens de informação que passam nesses critérios, mas que ainda assim podem ser melhor modelados como atributos, tendo como tipo um tipo de dado complexo, específico de domínio. Um atributo deve capturar um conceito atômico, i.e., um único valor ou um agrupamento de valores fortemente relacionados que sirva para descrever um outro objeto. Além disso, para que um item de estrutura complexa seja modelado como um atributo, ele deve ser compreensível pelos interessados simplesmente pelo seu nome.

É bom realçar que, com o tempo, as classes do domínio do problema tendem a permanecer relativamente estáveis, enquanto os atributos provavelmente se alteram. Atributos podem ser bastante voláteis, em função de alterações nas responsabilidades do sistema.

É muito importante lembrar também que, uma vez que atributos e associações são tipos de relacionamentos, não devemos incluir na lista de atributos de uma classe, atributos representando associações (ou atributos representando "chaves estrangeiras" como a classe fosse uma tabela de um banco de dados relacional). Associações já têm sua presença indicada pela notação de associação, ou seja pelas linhas que conectam as classes que se relacionam.

Um aspecto bastante importante na especificação de atributos é a escolha de nomes. Deve-se procurar utilizar o vocabulário típico do domínio do problema, usando nomes legíveis e abrangentes. Para nomear atributos, sugerem-se nomes iniciando com substantivo, o qual pode ser combinado com complementos ou adjetivos, omitindo-se preposições. O nome do atributo deve ser iniciado com letra minúscula, enquanto os nomes dos complementos devem iniciar com letras maiúsculas, sem dar um espaço em relação à palavra anterior. Acentos não devem ser utilizados. Atributos monovalorados devem iniciar com substantivo no singular (p.ex., nome, razaoSocial), enquanto atributos multivalorados devem iniciar com o substantivo no plural (p.ex., telefones).

A sintaxe de atributos na UML, em sua forma plena, é a seguinte (BOOCH; RUMBAUGH; JACOBSON, 2006):

visibilidade nome: tipo [multiplicidade] = valorInicial {propriedades}

A visibilidade de um atributo indica em que situações esse atributo é visível por outras classes. Na UML há quatro níveis de visibilidade, os quais são marcados pelos seguintes símbolos:

+ público : o atributo pode ser acessado por qualquer classe;

protegido: o atributo só é passível de acesso pela própria classe ou por uma de suas especializações;

- privado: o atributo só pode ser acessado pela própria classe;

~ pacote: o atributo só pode ser acessado por classes declaradas dentro do mesmo pacote da classe a que pertence o atributo.

A informação de visibilidade é inerente à fase de projeto e não deve ser expressa em um modelo conceitual. Assim, em um modelo conceitual, atributos devem ser especificados sem nenhum símbolo antecedendo o nome.

O *tipo* indica o tipo de dado do atributo, o qual deve ser um tipo de dado primitivo ou um tipo de dado específico de domínio. Tipos de dados específicos de domínio devem ser

definidos no Dicionário de Dados do Projeto. Para tornar os modelos conceituais mais simples, de modo a facilitar a comunicação com clientes e usuários, tipos de dados de atributos podem ser omitidos do diagrama de classes.

A multiplicidade é a especificação do intervalo permitido de itens que o atributo pode abrigar. O padrão é que cada atributo tenha um e somente um valor para o atributo. Quando um atributo for opcional ou quando puder ter mais do que uma ocorrência, a multiplicidade deve ser informada, indicando o valor mínimo e o valor máximo, da seguinte forma:

valor_mínimo .. valor_máximo

A seguir, são dados alguns exemplos:

- nome: String → instâncias da classe têm obrigatoriamente um e somente um nome.
- carteira: String [0..1] → instâncias da classe têm uma ou nenhuma carteira.
- telefones: Telefone [0..*] → instâncias da classe têm um ou vários telefones.
- pessoasContato: String [2] → instâncias da classe têm exatamente duas pessoas de contato.

Atributos podem ter um valor padrão inicial, ou seja, um valor que, quando não informado outro valor, será atribuído ao atributo. O campo *valorInicial* descreve exatamente este valor. O exemplo abaixo ilustra o uso de valor inicial.

origem: Ponto = (0,0) → a origem, quando não informado outro valor, será o ponto (0,0)

Finalmente, podem ser indicadas propriedades dos atributos. Uma propriedade que pode ser interessante mostrar em um modelo conceitual é a propriedade *readonly*, a qual indica que o valor do atributo não pode ser alterado após a inicialização do objeto. No exemplo abaixo, está-se indicando que o valor do atributo *numeroSocio* de um sócio de um clube não pode ser alterado.

numSocio: int {readonly}

Além das informações tratadas na declaração de um atributo seguindo a sintaxe da UML, outras informações de domínio, quando pertinentes, podem ser adicionadas no Dicionário de Dados do Projeto, tais como unidade de medida, intervalo de valores possíveis, limite, precisão etc.

b) Associações

Uma associação é um tipo de relacionamento que ocorre entre instâncias de duas ou mais classes. Assim como classes, associações são tipos. Ou seja, uma associação modela um tipo de relacionamento que pode ocorrer entre instâncias das classes envolvidas. Uma instância de uma associação (dita uma ligação) conecta instâncias específicas das classes envolvidas na associação. Seja o exemplo de um domínio em que clientes efetuam pedidos. Esse tipo de relacionamento pode ser modelado como uma associação *Cliente efetua Pedido*. Seja Pedro uma instância de *Cliente* e Pedido100 uma instância de *Pedido*. Se foi Pedro quem efetuou o Pedido100, então a ligação (Pedro, Pedido100) é uma instância da associação *Cliente efetua Pedido*.

Associações podem ser nomeadas. Neste texto sugere-se o uso de verbos conjugados, indicando o sentido de leitura. Ex.: Cliente (classe) *efetua* > (associação) Locação (classe). Cada classe envolvida na associação desempenha um papel, ao qual pode ser dado um nome. Cada

classe envolvida na associação possui também uma multiplicidade[15] nessa associação, que indica quantos objetos podem participar de uma instância dessa associação. A notação da UML tipicamente usada para representar associações em um modelo conceitual é ilustrada na Figura 5.26.

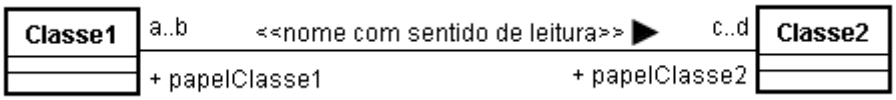

Figura 5.26 – Notação de Associações na UML.

Na ilustração da figura, um objeto da *Classe1* se relaciona com no mínimo *c* e no máximo *d* objetos da *Classe2*. Já um objeto da *Classe2* se relaciona com no mínimo *a* e no máximo *b* objetos da *Classe1*. Objetos da *Classe1* desempenham o papel de "*papelClasse1*" nesta associação, enquanto objetos da *Classe2* desempenham o papel de "*papelClasse2*" nessa mesma associação.

É importante, neste ponto, frisar a diferença entre sentido de leitura (ou direção do nome) de uma associação com a navegação da associação. O sentido de leitura diz apenas em que direção ler o nome da associação, mas nada diz sobre a navegabilidade da associação. A navegabilidade (linha de associação com seta direcionada) é usada para limitar a navegação de uma associação a uma única direção e é um recurso a ser usado apenas na fase de projeto. Em um modelo conceitual, todas as associações são não direcionais, ou seja, navegáveis nos dois sentidos.

Ainda que nomes de associações e papéis sejam opcionais, recomenda-se usá-los para tornar o modelo mais claro. Além disso, há algumas situações em que fica inviável ler um modelo se não houver a especificação do nome da associação ou de algum de seus papéis.

Seja o exemplo da Figura 5.27. Em uma empresa, um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Um departamento, por sua vez, pode ter vários empregados nele lotados, mas apenas um chefe. Sem nomear essas associações, o modelo fica confuso. Rotulando os papéis e as associações, o modelo torna-se muito mais claro. Na Figura 5.27, um departamento exerce o papel de *departamento de lotação* do empregado e, neste caso, um empregado tem um e somente um departamento de lotação. No outro relacionamento, um empregado exerce o papel de *chefe* e, portanto, um departamento possui um e somente um chefe.

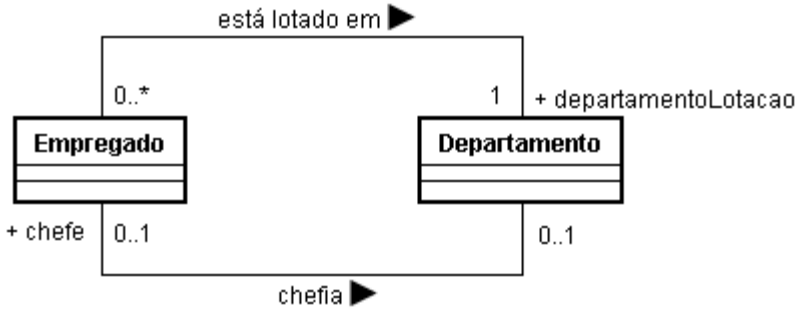

Figura 5.27 – Exemplo: Nomeando Associações.

[15] Multiplicidades em uma associação são análogas às multiplicidades em atributos e especificam as quantidades mínima e máxima de objetos que podem participar da associação. Quando nada for dito, o padrão é 1..1 como no caso de atributos. Contudo, para deixar os modelos claros, recomenda-se sempre especificar explicitamente as multiplicidades das associações.

Ao contrário das classes e dos atributos que podem ser encontrados facilmente a partir da leitura dos textos da descrição do minimundo e das descrições de casos de uso, muitas vezes, as informações sobre associações não aparecem tão explicitamente. Casos de uso descrevem ações de interação entre atores e sistema e, por isso, acabam mencionando principalmente operações. Operações transformam a informação, passando um objeto de um estado para outro, por meio da alteração dos seus valores de atributos e associações. Uma associação, por sua vez, é uma relação estática que pode existir entre duas classes. Assim, as descrições de casos de uso estão repletas de operações, mas não de associações (WAZLAWICK, 2004).

Contudo, conforme discutido na seção anterior, há alguns eventos que precisam ter sua ocorrência registrada e, portanto, são tipicamente mapeados como classes. Esses eventos estão descritos nos casos de uso e podem ter sido capturados como associações. Seja o exemplo de uma concessionária de automóveis. Neste domínio, clientes compram carros, como ilustra a parte (a) da Figura 5.28. Contudo, a compra é um evento importante para o negócio e precisa ser registrado. Neste caso, como ilustra a parte (b) da Figura 5.28, a compra deve ser tratada como uma classe e não como uma associação.

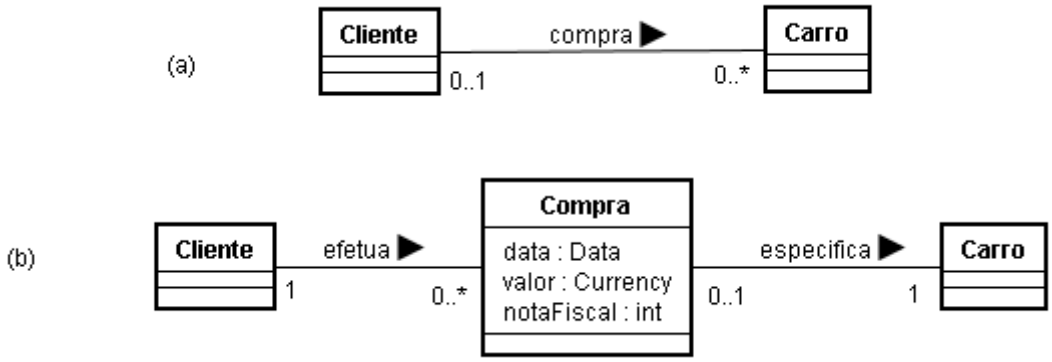

Figura 5.28 – Exemplo: Associação x Classe de Evento Lembrado.

Deve-se notar pelo exemplo acima que o evento é representado por uma classe, enquanto as associações continuam representando relacionamentos estáticos entre as classes e não operações ou transformações (WAZLAWICK, 2004). Assim, deve-se tomar cuidado com a representação de eventos como associações, questionando sempre se aquela associação é relevante para o sistema em questão.

Seja o exemplo da Figura 5.29. Nesse exemplo, o caso de uso aponta que funcionários são responsáveis por cadastrar livros em uma biblioteca. Seria necessário, pois, criar uma associação *Funcionário cadastra Livro* no modelo estrutural? A resposta, na maioria dos casos, é não. Apenas se explicitamente expresso pelo cliente em um requisito que é necessário saber exatamente qual funcionário fez o cadastro de um dado livro (o que é muito improvável de acontecer), é que tal relação deveria ser considerada. Mesmo se houver a necessidade de auditoria de uso do sistema (requisito não funcional relativo à segurança), não há a necessidade de modelar esta associação, pois requisitos não funcionais não devem ser considerados no modelo conceitual, uma vez que soluções bastante distintas à do uso dessa associação poderiam ser adotadas.

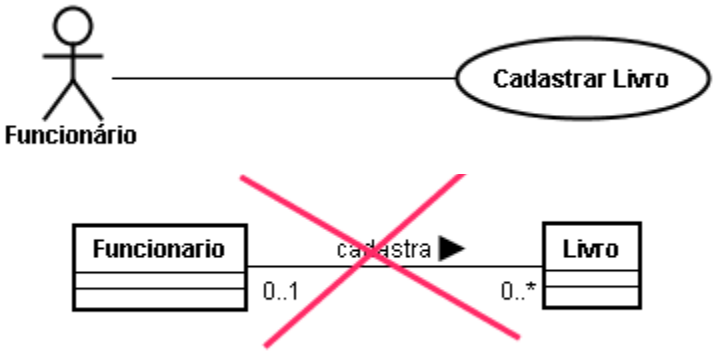

Figura 5.29 – Exemplo: Associação x Caso de Uso.

Na modelagem conceitual é fundamental saber a quantidade de objetos que uma associação admite em cada um de seus papéis, o que é capturado pelas multiplicidades da associação. Esta informação é bastante dependente da natureza do problema e do real significado da associação (o que se quer representar efetivamente), especialmente no que se refere à associação representar apenas o presente ou o histórico (WAZLAWICK, 2004).

Retomemos o exemplo da Figura 5.27, no qual se diz que um empregado está lotado em um departamento e, opcionalmente, pode chefiá-lo. Para definir precisamente as multiplicidades, é necessário investigar os seguintes aspectos: Um empregado pode mudar de lotação? Se sim, é necessário registrar apenas a lotação atual ou é necessário registrar o histórico de lotações dos empregados (ou seja, registrar o evento de lotação de um empregado em um departamento)? Um departamento pode, ao longo do tempo, mudar de chefe? Se sim, é necessário saber quem o histórico de chefias do departamento (ou seja, registrar o evento de nomeação do chefe do departamento)?

Como colocado no modelo da Figura 5.27, está-se representando apenas a situação presente. Se houver mudança de chefe de um departamento ou do departamento de lotação de um empregado, perder-se-á a informação histórica. Na maioria das vezes, essa não é uma solução aceitável. Na maioria dos domínios, as pessoas querem saber a informação histórica. Assim, nota-se que é parte das responsabilidades do sistema registrar a ocorrência dos eventos de nomeação do chefe e de lotação de empregados. Assim, um modelo mais fidedigno a essa realidade é o modelo da Figura 5.30, o qual introduz as classes do tipo "evento lembrado" *NomeacaoChefia* e *Lotacao*.

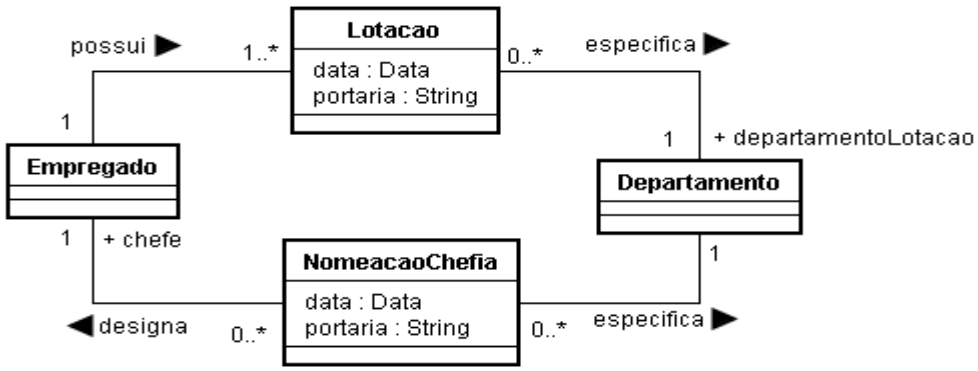

Figura 5.30– Registrando Históricos.

Ainda que este modelo seja mais fidedigno à realidade, ele ainda apresenta problemas. Por exemplo, o modelo diz que um empregado pode ter uma ou mais locações. Mas o empregado pode ter mais de uma lotação vigente? O mesmo vale para o caso da nomeação de

chefia. Um empregado pode ser chefe de mais de um departamento ao mesmo tempo? Um departamento pode ter mais do que um chefe nomeado ao mesmo tempo? Infelizmente, o modelo é incapaz de responder a essas perguntas. Para eliminar essas ambiguidades, é necessário capturar regras de negócio do tipo restrições de integridade. No exemplo acima, as seguintes regras se aplicam:

- Um empregado só pode estar lotado em um único departamento em um dado momento.
- Um empregado só pode estar designado como chefe de um único departamento em um dado momento.
- Um departamento só pode ter um empregado designado como chefe em um dado momento.

Observe que, como um departamento pode ter vários empregados nele lotados ao mesmo tempo, não é necessário escrever uma restrição de integridade, pois este é o caso mais geral (sem restrição). Assim, restrições de integridade devem ser escritas apenas para as associações que são passíveis de restrições.

Restrições de integridade são regras de negócio e poderiam ser lançadas no Documento de Requisitos. Contudo, como elas são importantes para a compreensão e eliminação de ambiguidades do modelo conceitual, é útil descrevê-las no próprio modelo conceitual.

Além das restrições de integridade relativas às multiplicidades *n*, diversas outras restrições podem ser importantes para tornar o modelo mais fiel à realidade. Ainda no exemplo da Figura 3.35, poder-se-ia querer dizer que um empregado só pode ser nomeado como chefe de um departamento, se ele estiver lotado nesse departamento. Restrições deste tipo são comuns em porções fechadas de um diagrama de classes. Assim, toda vez que se detectar uma porção fechada em um diagrama de classes, vale a pena analisá-la para avaliar se há ali uma restrição de integridade ou não. Havendo, deve-se escrevê-la.

Restrições de integridade também podem falar sobre atributos das classes. Por exemplo, a data da portaria de nomeação de um empregado *e* como chefe de um departamento *d* deve ser igual ou posterior à data da portaria de lotação do empregado *e* no departamento *d*.

Geralmente, restrições de integridade são escritas em linguagem natural, uma vez que não são passíveis de modelagem gráfica. Contudo, conforme já discutido anteriormente, o uso de linguagem natural pode levar a ambiguidades. Visando suprir essa lacuna na UML, o OMG[16] incorporou ao padrão uma linguagem para especificação formal de restrições, a OCL (*Object Constraint Language*). Contudo, restrições escritas em OCL dificilmente serão entendidas por clientes e usuários, o que dificulta a validação das mesmas. Assim, neste texto, sugere-se escrever as restrições de integridade em linguagem natural mesmo.

Vale ressaltar que a UML provê alguns mecanismos para representar restrições de integridade em um modelo gráfico. As próprias multiplicidades são uma forma de capturar restrições de integridade (ditas restrições de integridade de cardinalidade). Além das multiplicidades, a UML provê o recurso de restrições, as quais são representadas entre chaves (**{restrição}**). Restrições podem ser usadas, dentre outros, para restringir a ocorrência de associações. Seja o seguinte exemplo: em uma concessionária de automóveis compras podem

[16] *Object Management Group* (http://www.omg.org/) é uma organização internacional que gerencia padrões abertos relativos ao desenvolvimento orientado a objetos, dentre eles a UML.

ser financiadas ou por financeiras ou por bancos. Para capturar essa restrição, pode-se usar a restrição *xor* da UML, como ilustra a Figura 5.31.

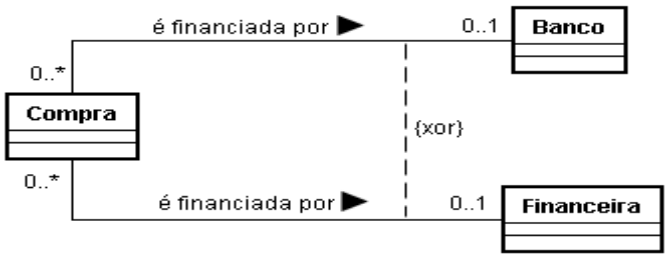

Figura 5.31 – Restrição XOR entre Associações.

Nesta figura, uma compra ou está relacionada a um banco ou a uma financeira. Não é possível que uma compra esteja associada aos dois ao mesmo. Como as multiplicidades mínimas do lado de banco e financeira são zero, uma compra pode não ser financiada.

Ainda em relação às multiplicidades, vale frisar que associações muitos-para-muitos são perfeitamente legais em um modelo orientado a objetos, como ilustra o exemplo da Figura 5.32. Nesse exemplo, está-se dizendo que disciplinas podem possuir vários pré-requisitos e podem ser pré-requisitos para várias outras disciplinas.

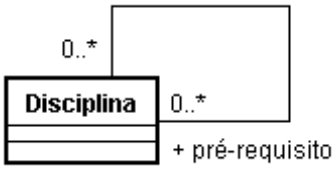

Figura 5.32 – Associação Muitos-para-Muitos.

Deve-se observar, no entanto, que muitas vezes, uma associação muitos-para-muitos oculta a necessidade de uma classe do tipo evento a ser lembrado. Seja o seguinte exemplo: em uma organização, empregados são alocados a projetos. Um empregado pode ser alocado a vários projeto, enquanto um projeto pode ter vários empregados a ele alocados. Tomando por base este fato, seria natural se chegar ao modelo da Figura 5.33(a). Contudo, se quisermos registrar as datas de início e fim do período em que o empregado esteve alocado ao projeto, esse modelo é insuficiente e deve ser alterado para comportar uma classe do tipo evento lembrado *Alocacao*, como mostra a Figura 5.33(b).

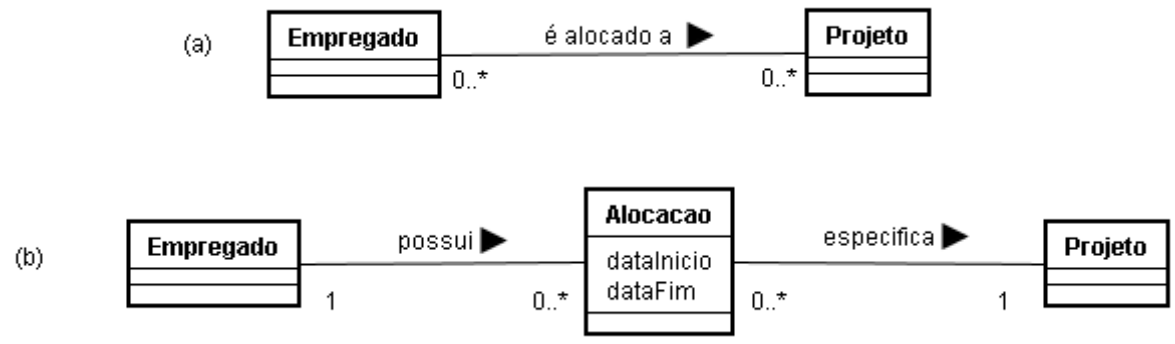

Figura 5.33– Associação Muitos-para-Muitos e Classes de Evento Lembrado.

De fato, o problema por detrás do modelo da Figura 5.33(a) é o mesmo anteriormente discutido na Figura 5.30: a necessidade ou não de se representar informação histórica. Contudo,

de maneira mais abrangente, pode-se pensar que se uma associação apresenta atributos, é melhor tratá-la como uma nova classe. Seja o seguinte exemplo: em uma loja, um cliente efetua um pedido, discriminando vários produtos, cada um deles em uma certa quantidade. O modelo da Figura 5.34(a) procura representar essa situação, mas uma questão permanece em aberto: onde representar a informação da quantidade pedida de cada produto? Essa informação não pode ficar em *Produto*, pois diferentes pedidos pedem quantidades diferentes de um mesmo produto. Também não pode ficar em *Pedido*, pois um mesmo pedido tipicamente especifica diferentes quantidades de diferentes produtos. De fato, quantidade não é nem um atributo da classe *Pedido* nem um atributo da classe *Produto*, mas sim um atributo da associação *especifica*. Assim, uma solução possível é introduzir uma classe *ItemPedido*, reificando[17] essa associação, como ilustra a Figura 5.34(b).

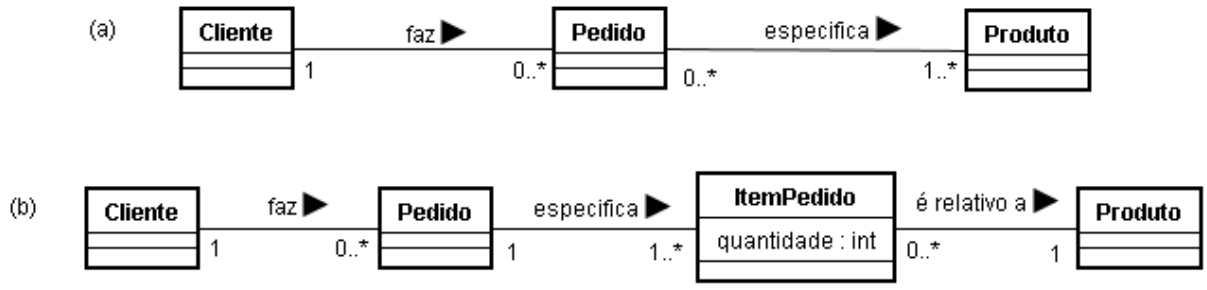

Figura 5.34 – Reificando uma Associação.

A UML oferece uma primitiva de modelagem, chamada *classe de associação*, que pode ser usada para tratar a reificação de associações (OLIVÉ, 2007). Uma classe de associação pode ser vista como uma associação que tem propriedades de classe (BOOCH; RUMBAUGH; JACOBSON, 2006). A Figura 5.35 mostra o exemplo anterior, sendo modelado como uma classe de associação, segundo a notação da UML.

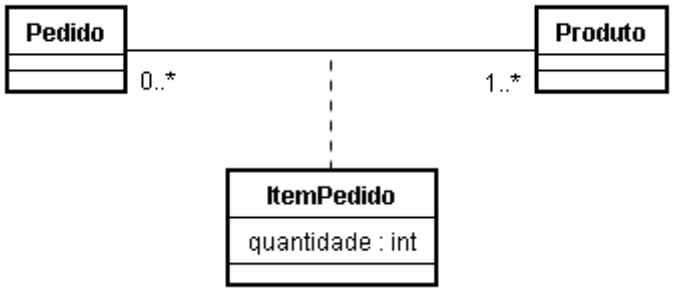

Figura 5.35 – Notação da UML para Classes Associativas.

Classes associativas são ainda representações de associações. Assim como uma instância de uma associação, uma instância de uma classe associativa é um par ordenado conectando duas instâncias das classes envolvidas na associação. Assim, se Pedido100 é uma instância de *Pedido*, Lápis é uma instância de *Produto* e o Pedido100 especifica 5 Lápis, então uma instância de *ItemPedido* é a tupla ((Pedido100, Lápis), 5).

Classes associativas podem ser usadas também para representar eventos cuja ocorrência precisa ser lembrada, como nos exemplos das figuras 5.30 e 5.33. Entretanto, é importante

[17] Reificar uma associação consiste em ver essa associação como uma classe. A palavra "reificação" vem da palavra do latim *res*, que significa coisa. Reificação corresponde ao que em linguagem natural se chama nominalização, que basicamente consiste em transformar um verbo em um substantivo (OLIVÉ, 2007).

observar que o uso de classes associativas nesses casos pode levar a problemas de modelagem. Seja o seguinte contexto: em um hospital, pacientes são tratados em unidades médicas. Um paciente pode ser tratado em diversas unidades médicas diferentes, as quais podem abrigar diversos pacientes sendo tratados. A Figura 5.36(a) mostra um modelo que busca representar essa situação usando uma classe de associação. Como uma classe associativa, as instâncias de Tratamento são pares ordenados (Paciente, Unidade Médica). Assim, cada vez que um paciente é tratado em uma unidade médica diferente tem-se um tratamento. Essa pode, contudo, não ser precisamente a concepção do problema original. Poder-se-ia imaginar que um tratamento é um tratamento de um paciente em várias unidades médicas. A classe de associação não captura isso. Assim, um modelo mais fiel ao domínio é aquele que representa Tratamento como uma classe do tipo evento a ser lembrado e que está relacionada com Paciente e Unidade Médica da forma mostrada na Figura 5.36(b).

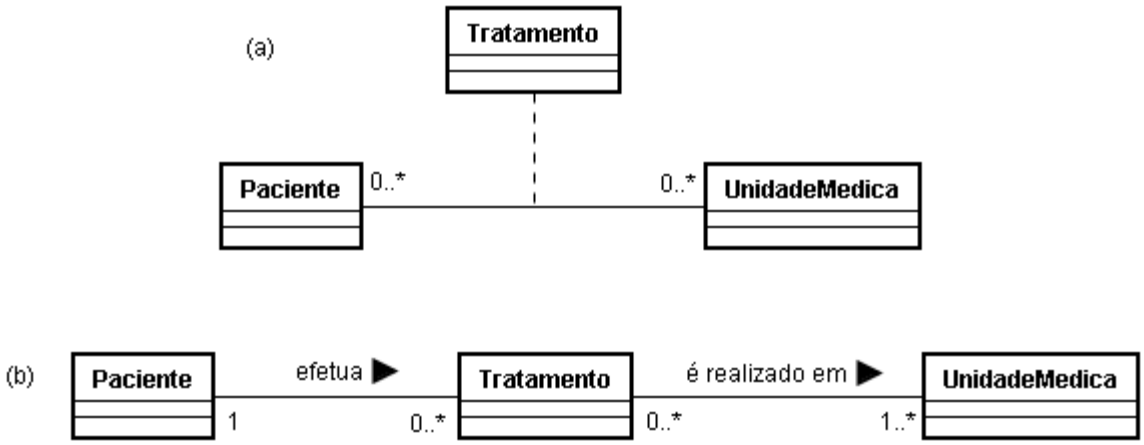

Figura 5.36 – Classes Associativas x Classes do Tipo Evento a Ser Lembrado.

Até o momento, todas as associações mostradas foram associações binárias, i.e., associações envolvendo duas classes. Mesmo o exemplo da Figura 5.32 (Disciplina tem como pré-requisito Disciplina) é ainda uma associação binária, na qual a mesma classe desempenha dois papéis diferentes (disciplina que possui pré-requisito e disciplina que é pré-requisito). Entretanto, associações *n*-árias são também possíveis, ainda que bem menos corriqueiramente encontradas. Uma associação ternária, por exemplo, envolve três classes, como ilustra o exemplo da Figura 5.37. Nesse exemplo, está-se dizendo que fornecedores podem fornecer produtos para certos clientes.

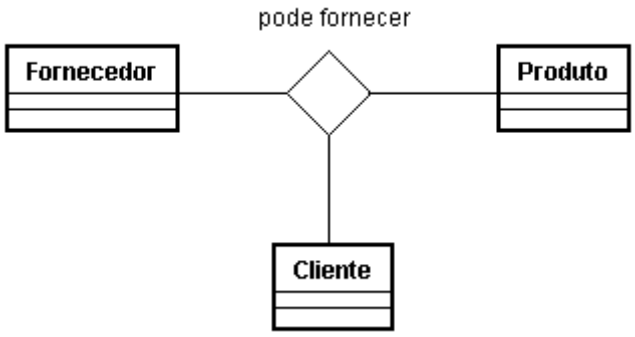

Figura 5.37 – Associação Ternária.

Na UML, associações *n*-árias são mostradas como losangos conectados às classes envolvidas na associação por meio de linhas sólidas, como mostra a Figura 3.42. O nome da associação é colocado dentro ou em cima do losango, sem direção de leitura. Normalmente, multiplicidades não são mostradas, dada a dificuldade de interpretá-las.

Finalmente, algumas associações podem ser consideradas mais fortes do que as outras, no sentido de que elas, na verdade, definem um objeto como sendo composto por outros (WAZLAWICK, 2004). Essas associações todo-parte podem ser de dois tipos principais: agregação e composição.

A *composição* é o tipo mais forte de associação todo-parte. Ela indica que um objeto-parte só pode ser parte de um único todo. Já a *agregação* não implica nessa exclusividade. Um carro, por exemplo, tem como partes um motor e quatro ou cinco rodas. Motor e rodas, ao serem partes de um carro, não podem ser partes de outros carros simultaneamente. Assim, esta é uma relação de composição, como ilustra a Figura 5.38(a). O exemplo da Figura 5.38 (b) ilustra o caso de comissões compostas por professores. Nesse caso, um professor pode participar de mais de uma comissão simultaneamente e, portanto, trata-se uma relação de agregação. Na UML, um losango branco na extremidade da associação relativa ao todo indica uma agregação. Já um losango preto indica uma composição.

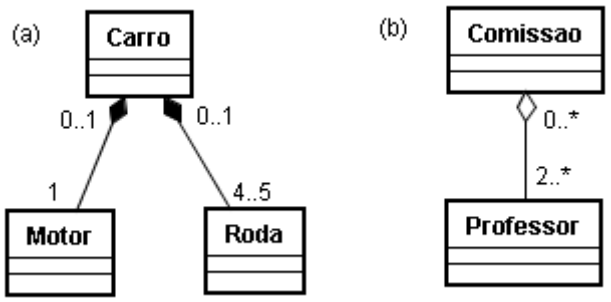

Figura 5.38 – Agregação e Composição.

Relações todo-parte podem ser empregadas em situações como:

- Quando há clareza de que um objeto complexo é composto de outros objetos (componente de). Ex.: Motor é um componente de um carro.
- Para designar membros de coleções (membro de). Ex.: Pesquisadores são membros de Grupos de Pesquisa.

Muitas vezes pode ser difícil perceber a diferença entre uma agregação / composição e uma associação comum. Quando houver essa dúvida, é melhor representar a situação usando uma associação comum, tendo em vista que ela impõe menos restrições.

5.7.3 Especificação de Hierarquias de Generalização / Especialização

Um dos principais mecanismos de estruturação de conceitos é a generalização / especialização. Com este mecanismo é possível capturar similaridades entre classes, dispondo-as em hierarquias de classes. No contexto da orientação a objetos, esse tipo de relacionamento é também conhecido como herança.

É importante notar que a herança tem uma natureza bastante diferente das associações. Associações representam possíveis ligações entre instâncias das classes envolvidas. Já a relação de herança é uma relação entre classes e não entre instâncias. Ao se considerar uma classe *B* como sendo uma subclasse de outra classe *A* está-se assumindo que todas as instâncias de *B* são também instâncias de *A*. Assim, ao se dizer que a classe *EstudanteGraduacao* herda da classe *Estudante*, está-se indicando que todos os estudantes de graduação são estudantes. Em resumo, deve-se interpretar a relação de herança como uma relação de subtipo entre classes. A Figura 5.39 mostra a notação da UML para representar herança.

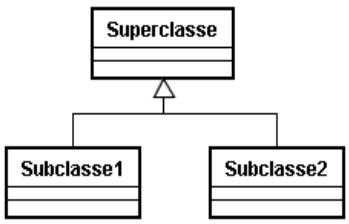

Figura 5.39 – Notação de Herança da UML.

A relação de herança é aplicável quando for necessário fatorar os elementos de informações (atributos e associações) de uma classe. Quando um conjunto de classes possuir semelhanças e diferenças, então elas podem ser organizadas em uma hierarquia de classes, de forma a agrupar em uma superclasse os elementos de informação comuns, deixando as especificidades nas subclasses.

De maneira geral, não faz sentido criar hierarquias de classes, quando as especializações (subclasses) não tiverem nenhum elemento de informação diferente. Quando isso ocorrer, é normalmente suficiente criar um atributo *tipo*, para indicar os possíveis subtipos da generalização. Seja o caso de um domínio em que se faz distinção entre clientes normais e clientes especiais, dos quais se quer saber exatamente as mesmas informações. Neste caso, criar uma hierarquia de classes, como ilustra a Figura 5.40(a), é desnecessário. Uma solução como a apresentada na Figura 5.40(b), em que o atributo *tipo* pode ser de um tipo enumerado com os seguintes valores {Normal, Especial}, modela satisfatoriamente o problema e é mais simples e, portanto, mais indicada.

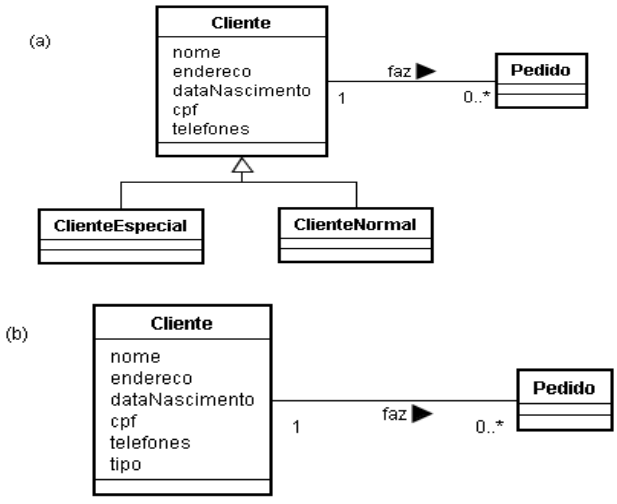

Figura 5.40 – Uso ou não de Herança.

Também não faz sentido criar uma hierarquia de classes em que a superclasse não tem nenhum atributo ou associação. Informações de estados pelos quais um objeto passa também não devem ser confundidas com subclasses. Por exemplo, um carro de uma locadora de automóveis pode estar locado, disponível ou em manutenção. Estes são estados e não subtipos de carro.

De fato, é interessante considerar alguns critérios para incluir uma subclasse (ou superclasse) em um modelo conceitual. O principal deles é o fato da especialização (ou generalização) estar dentro do domínio de responsabilidade do sistema. Apenas subclasses (superclasses) relevantes para o sistema em questão devem ser consideradas. Além desse critério básico, os seguintes critérios devem ser usados para analisar hierarquias de herança:

- Uma hierarquia de classes deve modelar relações "é-um-tipo-de", ou seja, toda subclasse deve ser um subtipo específico de sua superclasse.
- Uma subclasse deve possuir todas as propriedades (atributos e associações) definidas por suas superclasses e adicionar mais alguma coisa (algum outro atributo ou associação).
- Todas as instâncias de uma subclasse têm de ser também instâncias da superclasse.

Atenção especial deve ser dada à nomeação de classes em uma hierarquia de classes. Cada especialização deve ser nomeada de forma a ser autoexplicativa. Um nome apropriado para a especialização pode ser formado pelo nome de sua superclasse, acompanhado por um qualificador que descreve a natureza da especialização. Por exemplo, *EstudanteGraduacao* para designar um subtipo de *Estudante*.

Hierarquias de classes não devem ser usadas de forma não criteriosa, simplesmente para compartilhar algumas propriedades. Seja o caso de uma loja de animais, em que se deseja saber as seguintes informações sobre clientes e animais: nome, data de nascimento e endereço. Não faz nenhum sentido considerar que *Cliente* é uma subclasse de *Animal* ou vice-versa, apenas para reusar um conjunto de atributos que, coincidentemente, é igual.

No que se refere à modelagem de superclasses, deve-se observar se uma superclasse é *concreta* ou *abstrata*. Se a superclasse puder ter instâncias próprias, que não são instâncias de nenhuma de suas subclasses, então ela é uma classe concreta. Por outro lado, se não for possível instanciar diretamente a superclasse, ou seja, se todas as instâncias da superclasse são antes instâncias das suas subclasses, então a superclasse é abstrata. Classes abstratas são representadas na UML com seu nome escrito em itálico e não devem herdar de classes concretas.

Quando modeladas hierarquias de herança, é necessário posicionar atributos e associações adequadamente. Cada atributo ou associação deve ser colocado na classe mais adequada. Atributos e associações genéricos, que se aplicam a todas as subclasses, devem ser posicionados no topo da estrutura, de modo a serem aplicáveis a todas as especializações. De maneira mais geral, se um atributo ou associação é aplicável a um nível inteiro de especializações, então ele deve ser posicionado na generalização correspondente. Por outro lado, se algumas vezes um atributo ou associação tiver um valor significativo, mas em outras ele não for aplicável, deve-se rever seu posicionamento ou mesmo a estrutura de generalização-especialização adotada.

Inevitavelmente, o processo detalhado de designar atributos e associações a classes conduz a um entendimento mais completo da hierarquia de herança do que era possível em um

estágio anterior. Assim, deve-se esperar que o trabalho de reposicionamento de atributos e associações conduza a uma revisão de uma hierarquia de classes.

Por fim vale a pena mencionar que, durante anos, o mecanismo de herança foi considerado o grande diferencial da orientação a objetos. Contudo, com o passar do tempo, essa ênfase foi perdendo força, pois se percebeu que o uso da herança nem sempre conduz à melhor solução de um problema de modelagem. Hoje a herança é considerada apenas mais uma ferramenta de modelagem, utilizada basicamente para fatorar informações, as quais, de outra forma, ficariam repetidas em diferentes classes (WAZLAWICK, 2004).

5.8 Modelagem Dinâmica

Um sistema de informação realiza ações. O efeito de uma ação pode ser uma alteração em sua base de informação e/ou a comunicação de alguma informação ou comando para um ou mais destinatários. Um evento de requisição de ação (ou simplesmente uma requisição) é uma solicitação para o sistema realizar uma ação. O esquema comportamental de um sistema visa especificar essas ações (OLIVÉ, 2007).

Uma parte importante do modelo comportamental de um sistema é o modelo de casos de uso, o qual fornece uma visão das funcionalidades que o sistema deve prover. O modelo conceitual estrutural define os tipos de entidades (classes) e de relacionamentos (atributos e associações) do domínio do problema que o sistema deve representar para poder prover as funcionalidades descritas no modelo de casos de uso. Durante a realização de um caso de uso, atores geram eventos de requisição de ações para o sistema, solicitando a execução de alguma ação. O sistema realiza ações e ele próprio pode gerar outras requisições de ação. É necessário, pois, modelar essas requisições de ações, as correspondentes ações a serem realizadas pelo sistema e seus efeitos. Este é o propósito da modelagem dinâmica.

Em uma abordagem orientada a objetos, requisições de ação correspondem a mensagens trocadas entre objetos. As ações propriamente ditas e seus efeitos são tratados pelas operações das classes[18]. Assim, a modelagem dinâmica está relacionada com as trocas de mensagens entre objetos e a modelagem das operações das classes.

Os diagramas de classes gerados pela atividade de modelagem conceitual estrutural representam apenas os elementos estáticos de um modelo de análise orientada a objetos. É preciso, ainda, modelar o comportamento dinâmico da aplicação. Para tal, é necessário representar o comportamento do sistema como uma função do tempo e de eventos específicos. Um modelo de dinâmico indica como o sistema irá responder a eventos ou estímulos externos e auxilia o processo de descoberta das operações das classes do sistema.

Para apoiar a modelagem da dinâmica de sistemas, a UML oferece três tipos de diagramas (BOOCH; RUMBAUGH; JACOBSON, 2006): *Diagrama de Gráfico de Estados, Diagrama de Interação* e *Diagrama de Atividades.* Neste material é discutida apenas a modelagem de estados.

[18] De fato, abordagens distintas podem ser usadas, tal como representar tipos de requisições como classes, ditas classes de evento, e os seus efeitos como operações das correspondentes classes de evento, tal como faz Olivé (2007).

Diagrama de Estados

Um diagrama de estados mostra uma máquina de estados que consiste dos estados pelos quais objetos de uma particular classe podem passar ao longo de seu ciclo de vida e as transições possíveis entre esses estados, as quais são resultados de eventos que atingem esses objetos. Diagramas de gráfico de estados (ou **diagramas de transição de estados**) são usados principalmente para modelar o comportamento de uma classe, dando ênfase ao comportamento específico de seus objetos.

Classes com estados (ou modais) são classes cujas instâncias podem mudar de um estado para outro ao longo de sua existência, mudando possivelmente sua estrutura, seus valores de atributos ou comportamento dos métodos (WAZLAWICK, 2004).

Classes modais podem ser modeladas como máquinas de estados finitos. Uma máquina de estados finitos é uma máquina que, em um dado momento, está em um e somente um de um número finito de estados (OLIVÉ, 2007). Os estados de uma máquina de estados de uma classe modal correspondem às situações relevantes em que as instâncias dessa classe podem estar durante sua existência. Um estado é considerado relevante quando ele ajuda a definir restrições ou efeitos dos eventos.

Em qualquer estado, uma máquina de estados pode receber estímulos. Quando a máquina recebe um estímulo, ela pode realizar uma transição de seu estado corrente (dito estado origem) para um outro estado (dito estado destino), sendo que se assume que as transições são instantâneas. A definição do estado destino depende do estado origem e do estímulo recebido. Além disso, os estado origem e destino em uma transição podem ser o mesmo. Neste caso, a transição é dita uma autotransição (OLIVÉ, 2007).

Diagramas de Transições de Estados são usados modelar o comportamento de instâncias de uma classe modal como uma máquina de estados. Todas as instâncias da classe comportam-se da mesma maneira. Em outras palavras, cada diagrama de estados é construído para uma única classe, com o objetivo de mostrar o comportamento ao longo do tempo de vida de seus objetos. Diagramas de estados descrevem os possíveis estados pelos quais objetos da classe podem passar e as alterações dos estados como resultado de eventos (estímulos) que atingem esses objetos. Uma máquina de estado especifica a ordem válida dos estados pelos quais os objetos da classe podem passar ao longo de seu ciclo de vida. A Figura 5.41 mostra a notação básica da UML para diagramas de gráfico de estados.

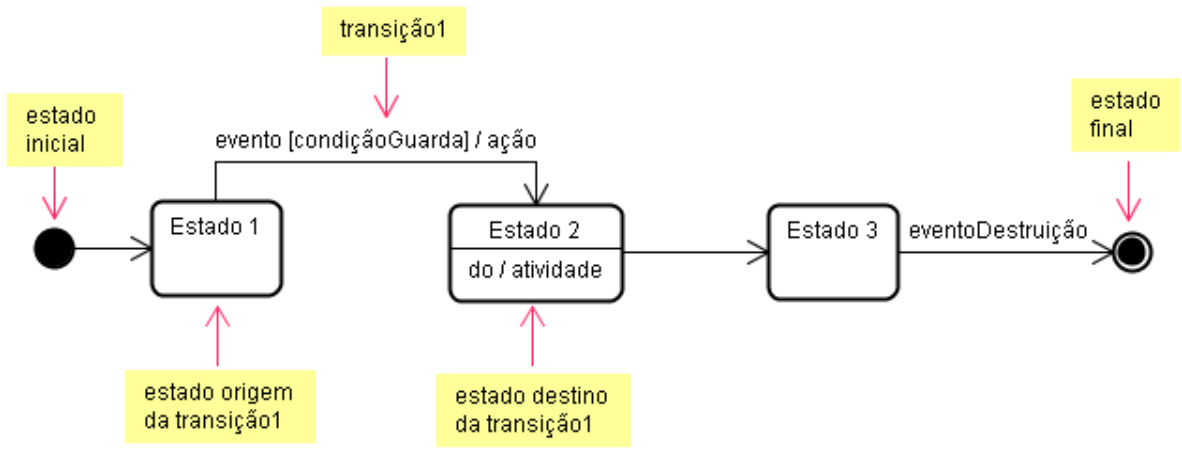

Figura 5.41 - Notação Básica da UML para Diagramas de Gráfico de Estados.

Um estado é uma situação na vida de um objeto durante a qual o objeto satisfaz alguma condição, realiza alguma atividade ou aguarda a ocorrência de um evento (BOOCH; RUMBAUGH; JACOBSON, 2006). Estados são representados por retângulos com os cantos arredondados, sendo que o nome de um estado deve ser único em uma máquina de estados. Uma regra prática para nomear estados consiste em atribuir um nome tal que sejam significativas sentenças do tipo "*o <<objeto>> está <<nome do estado>>*" ou "*o <<objeto>> está no estado <<nome do estado>>*". Por exemplo, em um sistema de locadora de automóveis, um estado possível de objetos da classe *Carro* seria "Disponível". A sentença "o carro está disponível" tem um significado claro (OLIVÉ, 2007).

Quando um objeto fica realizando uma atividade durante todo o tempo em que permanece em um certo estado, deve-se indicar essa atividade no compartimento de ações do respectivo estado. É importante realçar que uma atividade tem duração significativa e, quando concluída, tipicamente a conclusão provoca uma transição para um novo estado. A notação da UML para representar atividades de um estado é: **do / <<nomeAtividade>>**.

Transições são representadas por meio de setas rotuladas. Uma transição envolve um estado origem, um estado destino e normalmente um evento, dito o gatilho da transição. Quando a máquina de estados se encontra no estado origem e recebe o evento gatilho, então o evento dispara a transição e a máquina de estados vai para o estado destino. Se uma máquina recebe um evento que não é um gatilho para nenhuma transição, então ela não é afetada pelo evento (OLIVÉ, 2007).

Uma transição pode ter uma condição de guarda associada. Às vezes, há duas ou mais transições com o mesmo estado origem e o mesmo evento gatilho, mas com condições de guarda diferentes. Neste caso, a transição é disparada somente quando o evento gatilho ocorre e a condição de guarda é verdadeira (OLIVÉ, 2007). Quando uma transição não possuir uma condição de guarda associada, então ela ocorrerá sempre que o evento ocorrer.

Por fim, quando uma transição é disparada, uma ação instantânea pode ser realizada. Assim, o rótulo de uma transição pode ter até três partes, todas elas opcionais:

evento [condiçãoGuarda] / ação

Basicamente, a semântica de um diagrama de estados é a seguinte: quando o *evento* ocorre, se a *condição de guarda* é verdadeira, a *transição* dispara e a *ação* é realizada instantaneamente. O objeto passa, então, do *estado origem* para o *estado destino*. Se o estado destino possuir uma *atividade* a ser realizada, ela é iniciada.

O fato de uma transição não possuir um evento associado normalmente aponta para a existência de um evento implícito. Isso tipicamente ocorre em três situações: (i) o evento implícito é a conclusão da atividade do estado origem e a transição ocorrerá tão logo a atividade associada ao estado origem tiver sido concluída; (ii) o evento implícito é temporal, sendo disparado pela passagem do tempo; (iii) o evento implícito torna a condição de guarda verdadeira na base de informações do sistema, mas o evento em si não é modelado.

Embora ambos os termos ação e atividade denotem processos, eles não devem ser confundidos. Ações são consideradas processos instantâneos; atividades, por sua vez, estão sempre associadas a estados e têm duração no tempo. Vale a pena observar que, no mundo real, não há processos efetivamente instantâneos. Por mais rápida que seja, uma ação ocorrerá sempre em um intervalo de tempo. Esta simplificação de se considerar ações instantâneas no modelo conceitual pode ser associada à ideia de que a ação ocorre tão rapidamente que não é possível interrompê-la. Em contraste, uma atividade é passível de interrupção, sendo possível,

por exemplo, que um evento ocorra, interrompa a atividade e provoque uma mudança no estado do objeto antes da conclusão da atividade.

Às vezes quer se modelar situações em que uma ação instantânea é realizada quando se entra ou sai de um estado, qualquer que seja a transição que o leve ou o retire desse estado. Seja o exemplo de um elevador. Neste contexto, ao parar em um certo andar, o elevador abre a porta. Suponha que a abertura da porta do elevador seja um processo que não possa ser interrompido e, portanto, que se opte por modelá-lo como uma ação. Essa ação deverá ocorrer sempre que o elevador entrar no estado "*Parado*" e deve ser indicada no compartimento de ações desse estado como sendo uma ação de entrada no estado. A notação da UML para representar ações de entrada em um estado é: **entry / <<nomeAção>>**. Para representar ações de saída de um estado a notação é: **exit / <<nomeAção>>**.

Restam ainda na Figura 5.43 dois tipos especiais de estados: os ditos estados inicial e final. Conforme citado anteriormente, um objeto está sempre em um e somente um estado. Isso implica que, ao ser instanciado, o objeto precisa estar em algum estado. O estado inicial é precisamente esse estado. Graficamente, um estado inicial é mostrado como um pequeno círculo preenchido na cor preta. Seu significado é o seguinte: quando o objeto é criado, ele é colocado no estado inicial e sua transição de saída é automaticamente disparada, movendo o objeto para um dos estados da máquina de estados (no caso da Figura 5.43, para o *Estado1*). Toda máquina de estados tem de ter um (e somente um) estado inicial. Note que o estado inicial não se comporta como um estado normal[19], uma vez que objetos não se mantêm nele por um período de tempo. Ao contrário, uma vez que eles entram no estado inicial, sua transição de saída é imediatamente disparada e o estado inicial é abandonado. A transição de saída do estado inicial tem como evento gatilho implícito o evento responsável pela criação do objeto (OLIVÉ, 2007) e, na UML, esse evento não é explicitamente representado. Estados iniciais têm apenas transições de saída. As transições de saída de um estado inicial podem ter condições de guarda e/ou ações associadas. Quando houver condições de guarda, deve-se garantir que sempre pelo menos uma das transições de saída poderá ser disparada.

Quando um objeto deixa de existir, obviamente ele deixa de estar em qualquer um dos estados. Isso pode ser dito no diagrama por meio de uma transição para o estado final. O estado final indica, na verdade, que o objeto deixou de existir. Na UML um estado final é representado como um círculo preto preenchido com outro círculo não preenchido ao seu redor, como mostra a Figura 5.43. As transições para o estado final definem os estados em que é possível excluir o objeto. Classes cujos objetos não podem ser excluídos, portanto, não possuem um estado final (OLIVÉ, 2007). Assim como o estado inicial, o estado final não se comporta como um estado normal, uma vez que o objeto também não permanece nesse estado (já que o objeto não existe mais). Ao contrário do estado inicial, contudo, uma máquina de estados pode ter vários estados finais. Além disso, deve-se representar o evento que elimina o objeto (na Figura 5.43, *eventoDestruição*).

É importante indicar no diagrama de estados os eventos maiores (eventos de domínio e requisições de ações) e não os eventos estruturais que efetivamente alteram o estado do objeto. Assim, neste texto sugere-se indicar como eventos de transições de uma máquina de estados as requisições de realização de casos de uso do sistema (ou de fluxos de eventos específicos, quando um caso de uso tiver mais de um fluxo de eventos normal). Para facilitar a rastreabilidade, sugere-se usar como nome do evento exatamente o mesmo nome do caso de

[19] Por não se comportar como um estado normal, o estado inicial é considerado um pseudoestado no metamodelo da UML.

uso (ou do fluxo de eventos). Seja o exemplo de uma locadora de automóveis, que possua, dentre outros, os casos de uso mostrados na Figura 5.42, os quais possuem os fluxos de eventos mostrados nas notas anexadas aos casos de uso.

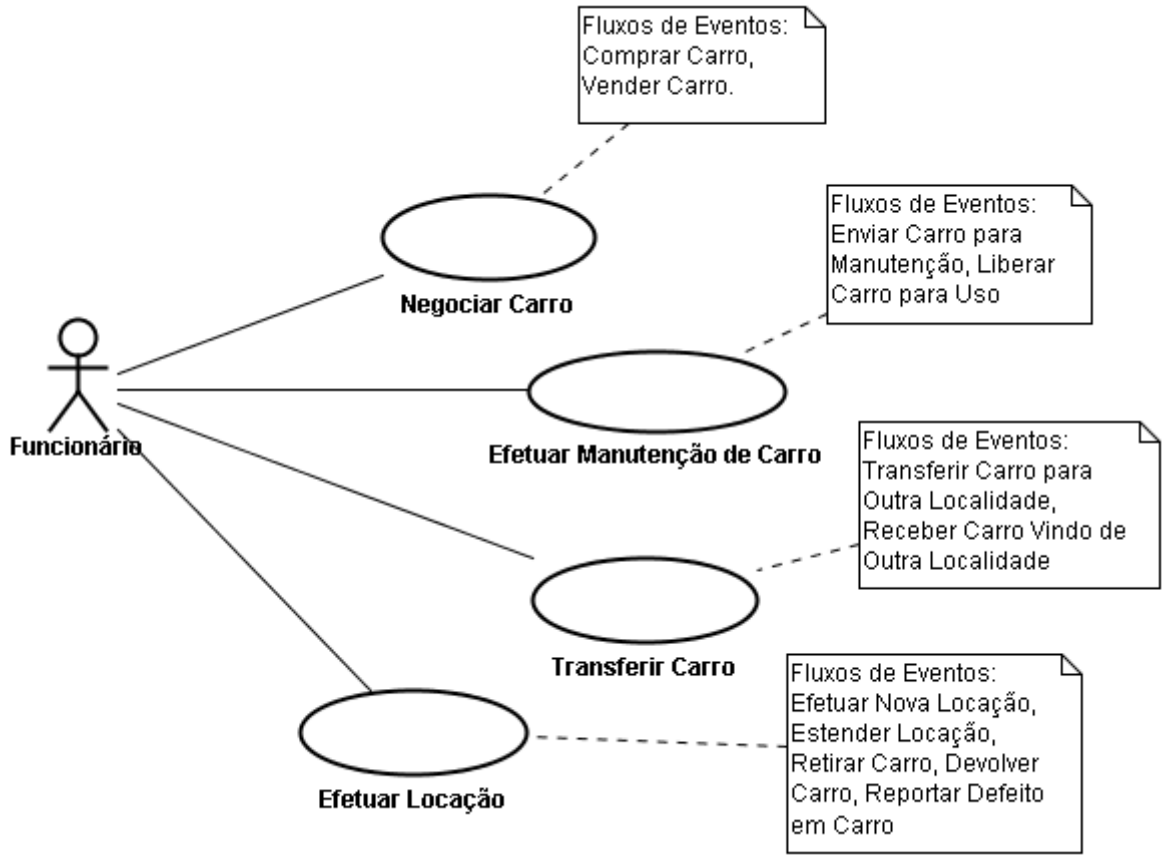

Figura 5.42 – Locadora de Automóveis - Casos de Uso e Fluxos de Eventos Associados.

A classe *Carro* tem o seu comportamento definido pela máquina de estados do diagrama de gráfico de estados da Figura 5.43. Ao ser adquirido (fluxo de eventos *Comprar Carro*, do caso de uso *Negociar Carro*), o carro é colocado *Em Preparação*. Quando liberado para uso (fluxo de eventos *Liberar Carro para Uso*, do caso de uso *Efetuar Manutenção de Carro*), o carro fica *Disponível*. Quando o cliente retira o carro (fluxo de eventos *Retirar Carro*, do caso de uso *Efetuar Locação*), este fica *Em Uso*. Quando é devolvido (fluxo de eventos *Devolver Carro*, do caso de uso *Efetuar Locação*), o carro fica novamente *Em Preparação*. Quando *Disponível*, um carro pode ser transferido de uma localidade para outra (fluxo de eventos *Transferir Carro para Outra Localidade* do caso de uso *Transferir Carro*). Durante o trânsito de uma localidade para outra, o carro está *Em Trânsito*, até ser recebido na localidade destino (fluxo de eventos *Receber Carro Vindo de Outra Localidade*, do caso de uso *Transferir Carro*), quando novamente é colocado *Em Preparação*. Finalmente, carros *Em Preparação* podem ser vendidos (fluxo de eventos *Vender Carro*, do caso de uso *Negociar Carro*), quando deixam de pertencer à locadora e são eliminados de sua base de informações.

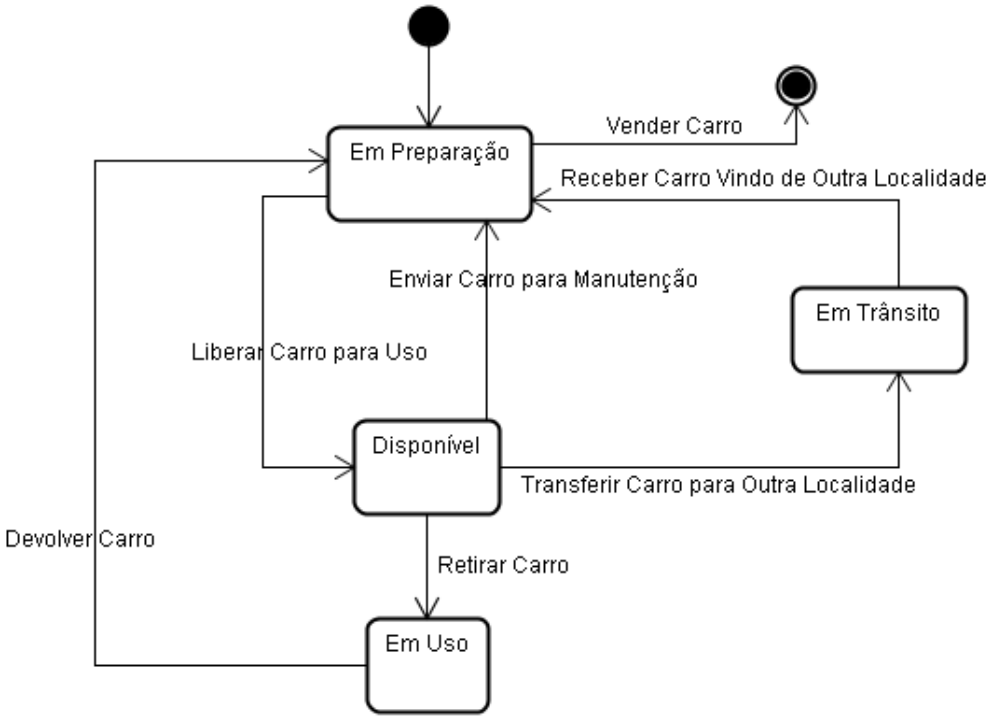

Figura 5.43 – Diagrama de Gráfico de Estados da Classe *Carro* – Disponibilidade (adaptado de (OLIVÉ, 2007)).

Nem todas as classes precisam ser modeladas como máquinas de estados. Apenas classes modais, i.e., que apresentam comportamento variável em função do estado de seus objetos, necessitam ser modeladas como máquinas de estados. Além disso, para os diagramas de estados serem efetivamente úteis, recomenda-se modelar uma máquina de estados somente se a classe em questão tiver três ou mais estados relevantes. Se uma classe possuir apenas dois estados relevantes, ainda cabe desenvolver uma máquina de estados. Contudo, de maneira geral, o diagrama tende a ser muito simples e a acrescentar pouca informação relevante que justifique o esforço de elaboração e manutenção do correspondente diagrama. Neste caso, os estados e transições podem ser levantados, sem no entanto elaborar um diagrama de estados.

Para algumas classes, pode ser útil desenvolver mais do que um diagrama de estados, cada qual modelando o comportamento dos objetos da classe por uma perspectiva diferente. Em um determinado momento, um objeto está em um (e somente um) estado em cada uma de suas máquinas de estado. Cada diagrama define seu próprio conjunto de estados nos quais um objeto pode estar, a partir de diferentes pontos de vista (OLIVÉ, 2007). Seja novamente o exemplo da classe *Carro*. A Figura 5.43 mostra os possíveis estados de um carro segundo um ponto de vista de disponibilidade. Entretanto, independentemente da disponibilidade, do ponto de vista de negociabilidade, um carro pode estar em dois estados (Não à Venda, À Venda), como mostra a Figura 5.44.

Vale ressaltar que os diferentes diagramas de estados de uma mesma classe não devem ter estados comuns. Cada diagrama deve ter seu próprio conjunto de estados e cada estado pertence a somente um diagrama de estados. Já os eventos podem aparecer em diferentes diagramas de estados, inclusive de classes diferentes. Quando um evento aparecer em mais de um diagrama de estados, sua ocorrência vai disparar as correspondentes transições em cada uma das máquinas de estados em que ele aparecer (OLIVÉ, 2007).

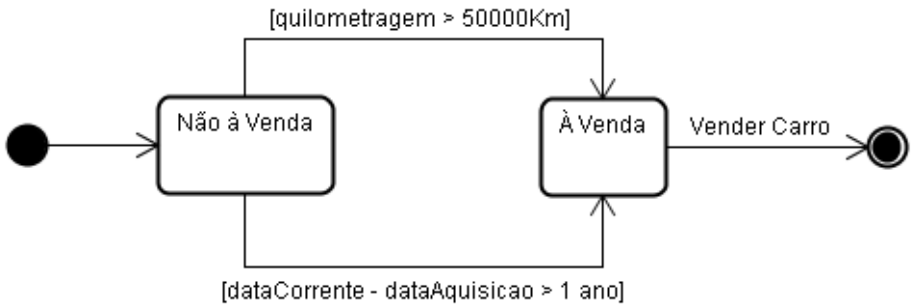

Figura 5.44 – Diagrama de Gráfico de Estados da Classe *Carro* – Negociabilidade (adaptado de (OLIVÉ, 2007)).

A Figura 5.45 mostra duas transições em que os eventos não são declarados explicitamente. No primeiro caso (quilometragem > 50000Km), o evento implícito torna a condição de guarda verdadeira na base de informações do sistema. Esse evento corresponde ao registro no sistema de qual é a quilometragem corrente do carro. Caso esse registro ocorra sempre no ato da devolução do carro pelo cliente (fluxo de eventos *Devolver Carro*, do caso de uso *Efetuar Locação*) e/ou no ato do recebimento do carro vindo de outra localidade (fluxo de eventos *Receber Carro Vindo de Outra Localidade*, do caso de uso *Transferir Carro*), esses eventos poderiam ser explicitamente declarados. Contudo, se o registro pode ocorrer em vários eventos diferentes, é melhor deixar o evento implícito. O segundo caso (dataCorrente – dataAquisicao > 1 ano) trata-se de um evento temporal, disparado pela passagem do tempo.

Todos os estados mostrados até então são estados simples, i.e., estados que não possuem subestados. Entretanto, há também estados compostos, os quais podem ser decompostos em um conjunto de subestados disjuntos e mutuamente exclusivos e um conjunto de transições (OLIVÉ, 2007). Um subestado é um estado aninhado em outro estado. O uso de estados compostos e subestados é bastante útil para simplificar a modelagem de comportamentos complexos. Seja o exemplo da Figura 5.33, que trata da disponibilidade de um carro. Suponha que seja necessário distinguir três subestados do estado *Em Uso*, a saber: *Em Uso Normal*, quando o carro não está quebrado nem em atraso; *Quebrado*, quando o cliente reportar um defeito no carro; e *Em Atraso*, quando o carro não foi devolvido na data de devolução prevista e não está quebrado. A Figura 5.45 mostra a máquina de estados da classe *Carro* considerando, agora, que, quando um carro está em uso, ele pode estar nesses três subestados.

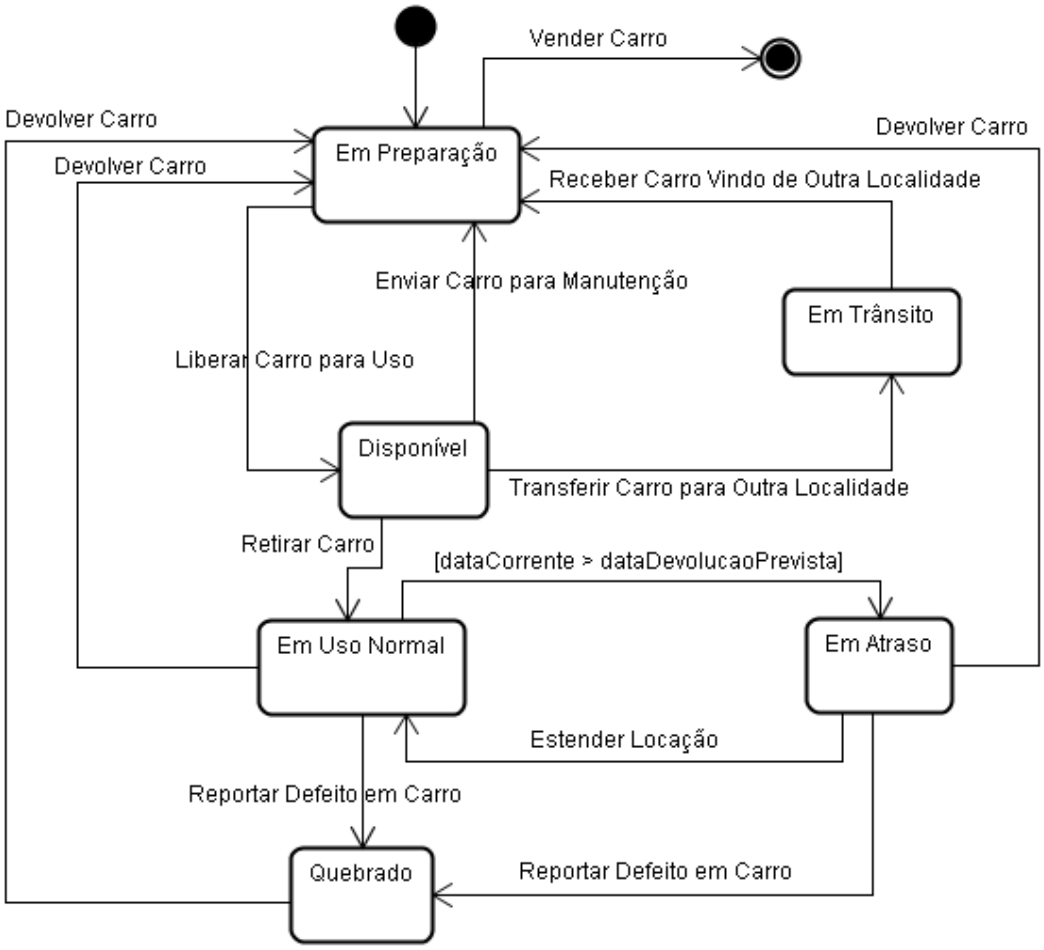

Figura 5.45 – Diagrama de Estados da Classe *Carro* (Disponibilidade) com Subestados de *Em Uso* (adaptado de (OLIVÉ, 2007)).

Nesse diagrama, não está sendo mostrado que os estados *Em Uso Normal*, *Em Atraso* e *Quebrado* são, de fato, subestados do estado *Em Uso* e, portanto, transições comuns (por exemplo, aquelas provocadas pelo evento *Devolver Carro*) são repetidas. Isso torna o modelo mais complexo e fica claro que esta solução representando diretamente os subestados (e omitindo o estado composto) não é escalável para sistemas que possuem muitos subestados, levando a diagramas confusos e desestruturados (OLIVÉ, 2007). A Figura 5.46 mostra uma solução mais indicada, em que tanto o estado composto quanto seus subestados são mostrados no mesmo diagrama. Uma outra opção é ocultar a decomposição do estado composto, mantendo o diagrama como o mostrado na Figura 5.43, e mostrar essa decomposição em um diagrama de estados separado.

Se um objeto está em um estado composto, então ele deve estar também em um de seus subestados. Assim, um estado composto pode possuir um estado inicial para indicar o subestado padrão do estado composto, como representado na Figura 5.46. Entretanto, deve-se considerar que as transições podem começar e terminar em qualquer nível. Ou seja, uma transição pode ir (ou partir) diretamente de um subestado (OLIVÉ, 2007). Assim, uma outra opção para o diagrama da Figura 5.46 seria fazer a transição nomeada pelo evento *Retirar Carro* chegar diretamente ao subestado *Em Uso Normal*, ao invés de chegar ao estado composto *Em Uso*.

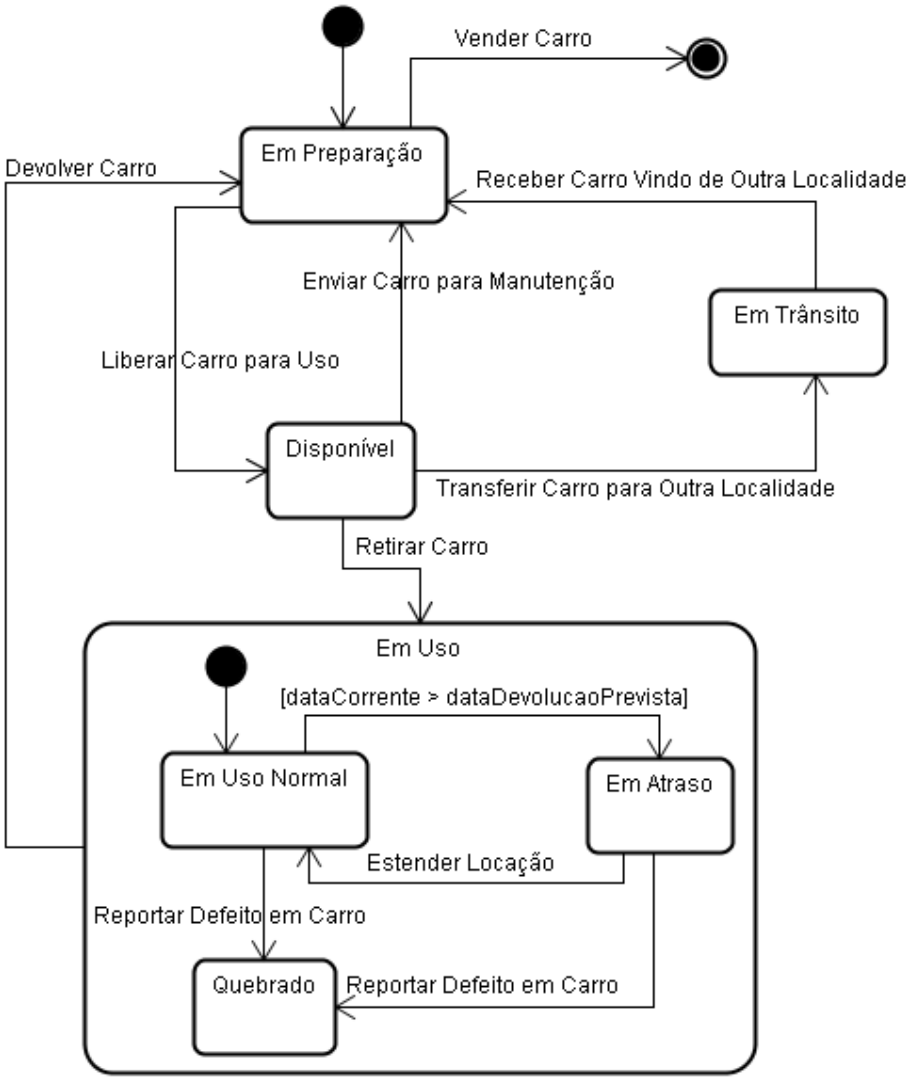

Figura 5.46 – Diagrama de Estados da Classe *Carro* (Disponibilidade) com Estado Composto *Em Uso* (adaptado de (OLIVÉ, 2007)).

O estado de um objeto deve ser mapeado no esquema estrutural. De maneira geral, o estado pode ser modelado por meio de um atributo. Esse atributo deve ser monovalorado e obrigatório ([1..1]). O conjunto de valores possíveis do atributo é o conjunto dos estados possíveis, conforme descrito pela máquina de estados (OLIVÉ, 2007). Assim, é bastante natural que o tipo de dados desse atributo seja definido como um tipo de dados enumerado. Um nome adequado para esse atributo é "estado". Contudo, outros nomes mais significativos para o domínio podem ser atribuídos. Em especial, quando uma classe possuir mais do que uma máquina de estado e, por conseguinte, mais do que um atributo de estado for necessário, o nome do atributo de estado deve indicar a perspectiva capturada pela correspondente máquina de estados.

É interessante observar que algumas transições podem mudar a estrutura da classe. Quando os diferentes estados de um objeto não afetam a sua estrutura, mas apenas, possivelmente, os valores de seus atributos e associações, diz-se que a transição é estável e os diferentes estados podem ser mapeados para um simples atributo (WAZLAWICK, 2004), conforme discutido anteriormente.

Entretanto, há situações em que, conforme um objeto vai passando de um estado para outro, ele vai ganhando novos atributos ou associações, ou seja, há uma mudança na estrutura da classe. Seja o exemplo de uma locação de carro. Como mostra a Figura 5.47, quando uma locação é criada, ela está ativa, em curso normal. Quando o carro não é devolvido até a data de devolução prevista, a locação passa a ativa com prazo expirado. Se a locação é estendida, ela volta a ficar em curso normal. Quando o carro é devolvido, a locação fica pendente. Finalmente, quando o pagamento é efetuado, a locação é concluída.

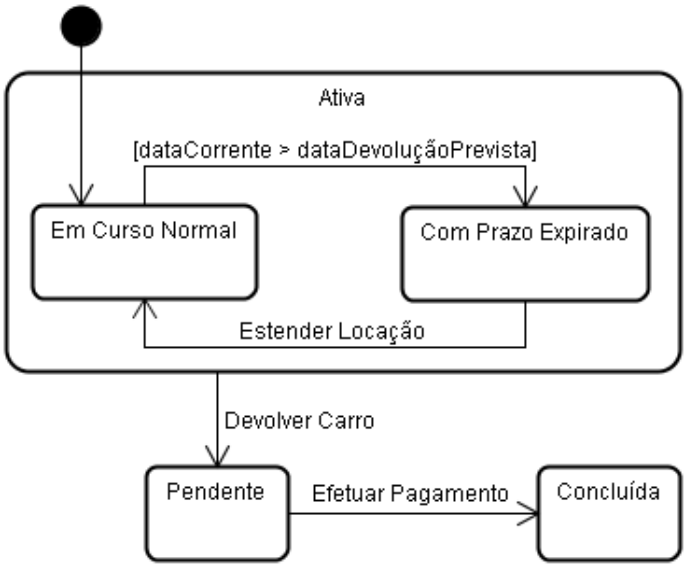

Figura 5.47 – Diagrama de Estados da Classe *Locação*.

Locações ativas (e em seus subestados, obviamente) têm como atributos: data de locação, data de devolução prevista, valor devido e caução. Quando no estado pendente, é necessário registrar a data de devolução efetiva e os problemas observados no carro devolvido. Finalmente, quando o pagamento é efetuado, é preciso registrar a data do pagamento, o valor e a forma de pagamento. Diz-se que as transições dos estados de *Ativa* para *Pendente* e de *Pendente* para *Concluída* são monotônicas[20], porque a cada mudança de estado, novos relacionamentos (atributos ou associações) são acrescentados (mas nenhum é retirado).

Uma solução frequentemente usada para capturar essa situação no modelo conceitual estrutural consiste em criar uma única classe (*Locacao*) e fazer com que certos atributos sejam nulos até que o objeto mude de estado, como ilustra a Figura 5.48. Essa forma de modelagem, contudo, pode não ser uma boa opção, uma vez que gera classes complexas com regras de consistência que têm de ser verificadas muitas vezes para evitar a execução de um método que atribui um valor a um atributo específico de um estado (WAZLAWICK, 2004), tal como *dataPagamento*.

[20] Monotônico diz respeito a algo que ocorre de maneira contínua. Neste caso, a continuidade advém do fato de um objeto continuamente ganhar novos atributos e associações, sem perder os que já possuía.

Figura 5.48 – Classe *Locação* com atributos inerentes a diferentes estados.

É possível modelar essa situação desdobrando o conceito original em três: um representando a locação efetivamente, outro representando a devolução e outro representando o pagamento. Desta forma, captura-se claramente os eventos de locação, devolução e pagamento, colocando as informações de cada evento na classe correspondente, como ilustra a Figura 5.49.

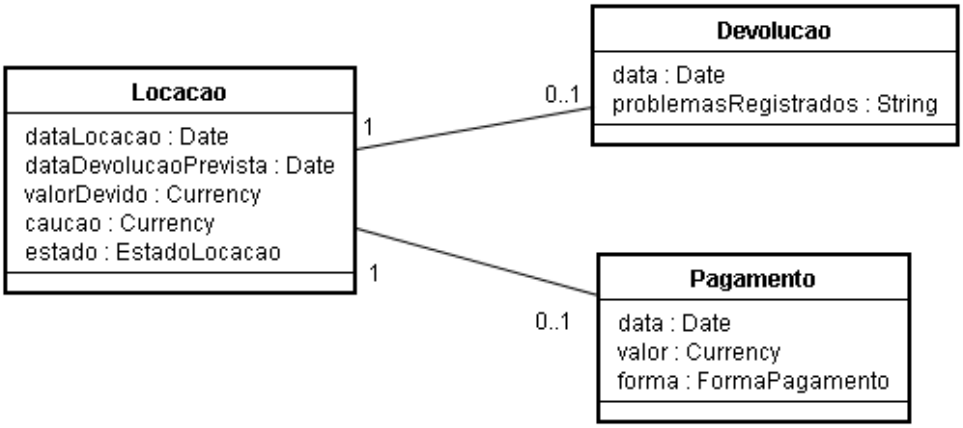

Figura 5.49– Distribuindo as responsabilidades.

Finalmente, vale à pena comentar que estados de uma classe modal podem ser tratados por meio de operações ao invés de atributos. Seja o exemplo anterior de locações de carros (Figura 5.49). O estado de uma locação pode ser computado a partir dos atributos e associações da classe *Locacao*, sem haver a necessidade de um atributo *estado*. Se uma locação não tem uma devolução associada, então ela está ativa. Estando ativa, se a data corrente é menor ou igual à data de devolução prevista, então a locação está em curso normal; caso contrário, ela está com prazo expirado. Se uma locação possui uma devolução, mas não possui um pagamento associado, então ela está pendente. Finalmente, se a locação possui um pagamento associado, então ela está concluída. Em casos como este, pode-se optar por tratar estado como uma operação e não como um atributo. Opcionalmente, pode-se utilizar a operação para calcular o valor de um atributo derivado[21] *estado*. Atributos derivados são representados na UML precedidos por uma barra (no exemplo, */estado*).

[21] Um atributo é derivado quando seu valor pode ser deduzido ou calculado a partir de outras informações (atributos e associações) já existentes no modelo estrutural.

Referências do Capítulo

AURUM, A., WOHLIN, C., *Engineering and Managing Software Requirements*, Springer-Verlag, 2005.

BLAHA, M., RUMBAUGH, J., *Modelagem e Projetos Baseados em Objetos com UML 2*, Elsevier, 2006.

BOOCH, G., *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings Publishing Company, Inc, 1994.

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

COCKBURN, A., *Escrevendo Casos de Uso Eficazes: Um guia prático para desenvolvedores de software*, Porto Alegre: Bookman, 2005.

ISO/IEC 25010, System and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.

ISO/IEC 25023, System and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality, 2016.

ISO/IEC 9126-1, Software Engineering - Product Quality - Part 1: Quality Model, 2001.

ISO/IEC TR 9126-2:2003, Software Engineering – Product Quality – Part 2: External Metrics, 2003a.

ISO/IEC TR 9126-3:2003, Software Engineering – Product Quality – Part 3: Internal Metrics, 2003b.

JACOBSON, I.; *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

LAMSWEERDE, A., *Requirements Engineering – From System Goals to UML Models to Software Specifications*, Wiley, 2009.

OLIVÉ, A., *Conceptual Modeling of Information Systems*, Springer, 2007.

RUMBAUGH, J., et al.; *Modelagem e Projetos Baseados em Objetos*, 1ª Edição, Editora Campus, 1994.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.

Capítulo 6 – Projeto de Software

O objetivo da fase de projeto (ou design) é produzir uma solução para o problema identificado e modelado durante o levantamento e análise de requisitos, incorporando a tecnologia aos requisitos e projetando o que será construído na implementação. Sendo assim, é necessário conhecer a tecnologia disponível e os ambientes de hardware e software onde o sistema será desenvolvido e implantado. Durante o projeto, deve-se decidir como o problema será resolvido, começando em um alto nível de abstração, próximo da análise, e progredindo sucessivamente para níveis mais detalhados até se chegar a um nível de abstração próximo da implementação.

O projeto de software encontra-se no núcleo técnico do processo de desenvolvimento de software e é aplicado independentemente do modelo de ciclo dev ida e paradigma adotados. É iniciado assim que os requisitos do software tiverem sido modelados e especificados pelo menos parcialmente e é a última atividade de modelagem. Por outro lado, corresponde à primeira atividade que leva em conta aspectos tecnológicos (PRESSMAN, 2006).

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 6.1. Assim, o objetivo do projeto é incorporar a tecnologia aos requisitos essenciais do usuário, projetando o que será construído na implementação.

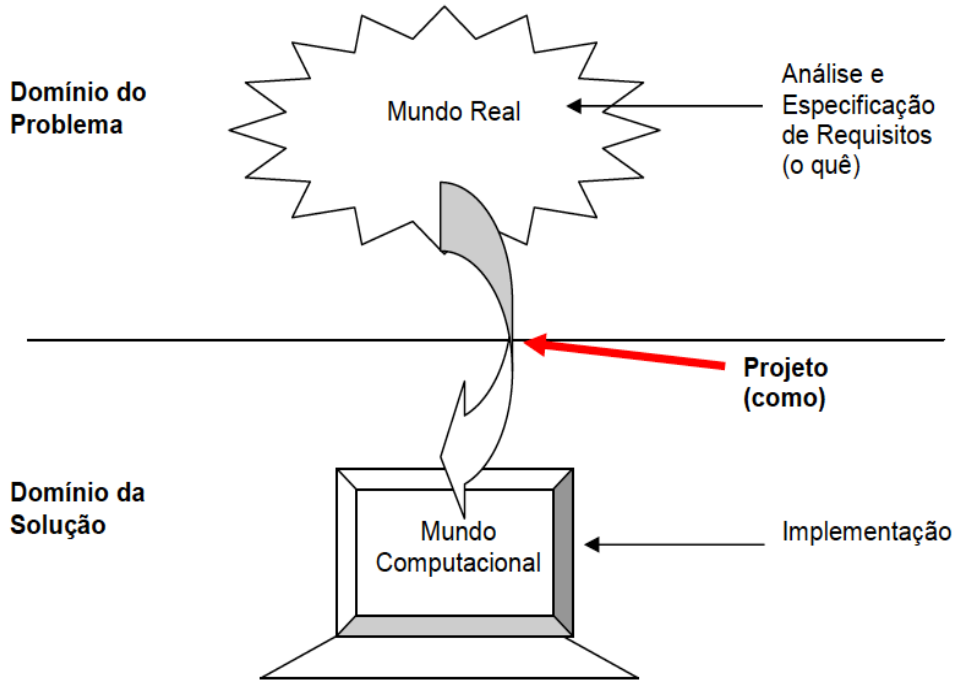

Figura 6.1 – Visão Geral da Atividade de Projeto.

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.

Tendo em vista que a orientação a objetos é um dos paradigmas mais utilizados atualmente no desenvolvimento de sistemas, este texto aborda o projeto de software orientado a objetos. Além disso, o foco deste texto são os sistemas de informação. Considerando essa classe de sistemas, de maneira geral, os seguintes elementos estão presentes na arquitetura de um sistema:

- Lógica de Domínio: é o elemento da arquitetura que trata de toda a lógica do sistema, englobando tanto aspectos estruturais (classes de domínio derivadas dos modelos conceituais estruturais da fase de análise), quanto comportamentais (classes de processo que tratam das funcionalidades descritas pelos casos de uso).
- Interface com o Usuário: é o elemento da arquitetura que trata da interação humano-computador. Envolve tanto as interfaces propriamente ditas (objetos gráficos responsáveis por receber dados e comandos do usuário e apresentar resultados) quanto o controle da interação, abrindo e fechando janelas, habilitando ou desabilitando botões etc. (WAZLAWICK, 2004).
- Persistência: é o elemento da arquitetura responsável pelo armazenamento e recuperação de dados em memória secundária (classes que representam e isolam os depósitos de dados do restante do sistema).

Este capítulo aborda o projeto de software focando em seus principais elementos. Para isso, está assim organizado: a seção 6.1 trata aspectos relevantes para o projeto de software, abordando qualidade do projeto de software, arquitetura de software, padrões (*patterns*) e documentação do projeto de software; a seção 6.2 trata da definição da arquitetura de software; nas seções 6.3, 6.4 e 6.5 são tratadas as camadas da arquitetura de software, sendo, respectivamente, domínio do problema, interface com o usuário e gerência de dados.

6.1 Aspectos Relevantes ao Projeto de Software

Nesta seção são discutidos alguns aspectos relevantes quando se fala em Projeto de Software: qualidade, arquitetura, padrões e documentação.

6.1.1 Qualidade do Projeto de Software

Um bom projeto de software deve apresentar determinadas características de qualidade, tais como facilidade de entendimento, facilidade de implementação, facilidade de realização de testes, facilidade de modificação e tradução correta das especificações de requisitos e de análise (PFLEEGER, 2004). Para se obter bons projetos, é necessário considerar alguns aspectos intimamente relacionados com a qualidade dos projetos, dentre eles (PRESSMAN, 2011):

- **Níveis de Abstração**: a abstração é um dos modos fundamentais pelos quais os seres humanos enfrentam a complexidade. Assim, um bom projeto deve considerar vários níveis de abstração, começando com em um nível mais alto, próximo da fase de análise. À medida que se avança no processo de projeto, o nível de abstração deve ser reduzido. Dito de outra maneira, o projeto deve ser um processo de refinamento, no qual o projeto vai sendo conduzido de níveis mais altos para níveis mais baixos de abstração.

- **Modularidade**: um bom projeto deve estruturar um sistema como módulos ou componentes coesos e fracamente acoplados. A modularidade é o atributo individual que permite a um projeto de sistema ser intelectualmente gerenciável. A estratégia "dividir para conquistar" é reconhecidamente útil no projeto de software, pois é mais fácil resolver um problema complexo quando o mesmo é dividido em partes menores gerenciáveis

- **Ocultação de Informações**: o conceito de modularidade leva o projetista a uma questão fundamental: até que nível a decomposição deve ser aplicada? Em outras palavras, quão modular deve ser o software? O princípio da ocultação de informações sugere que os módulos / componentes sejam caracterizados pelas decisões de projeto que cada um deles esconde dos demais. Módulos devem ser projetados e especificados de modo que as informações neles contidas (dados e algoritmos) sejam inacessíveis a outros módulos, sendo necessário conhecer apenas a sua interface. Ou seja, a ocultação de informação trabalha encapsulando detalhes que provavelmente serão alterados independentemente em diferentes módulos. A interface de um módulo revela apenas aqueles aspectos considerados improváveis de mudar (BASS; CLEMENTS; KAZMAN, 2003).

- **Independência Funcional**: a independência funcional é uma decorrência direta da modularidade e dos conceitos de abstração e ocultação de informações. Ela é obtida pelo desenvolvimento de módulos com finalidade única e pequena interação com outros módulos, isto é, módulos devem cumprir uma função bem estabelecida, minimizando interações com outros módulos. Módulos funcionalmente independentes são mais fáceis de entender, desenvolver, testar e alterar. Efeitos colaterais causados pela modificação de um módulo são limitados e, por conseguinte, a propagação de erros é reduzida. A independência funcional pode ser avaliada usando dois critérios de qualidade: coesão e acoplamento. A coesão se refere ao elo de ligação com o qual um módulo é construído. Uma classe, p.ex., é dita coesa quando tem um conjunto pequeno e focado de responsabilidades e aplica seus atributos e métodos especificamente para implementar essas responsabilidades. Já o 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. Idealmente, classes de projeto em um subsistema deveriam ter conhecimento limitado de classes de outros subsistemas. Coesão e acoplamento são interdependentes e, portanto, uma boa coesão deve conduzir a um pequeno acoplamento. A Figura 6.2 procura ilustrar esse fato.

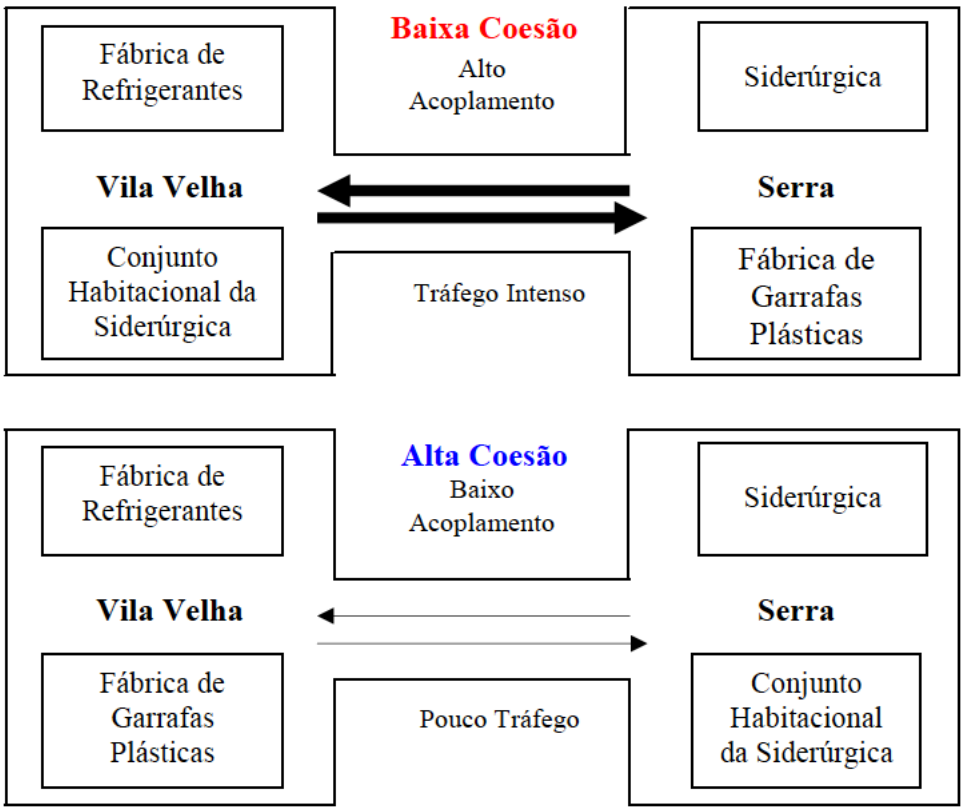

Figura 6.2 – Coesão e Acoplamento

6.1.2 Arquitetura de Software

De acordo com Bass, Clements e Kazman (2003), a arquitetura de software de um sistema computacional é a estrutura (ou estruturas) do sistema que compreende elementos de software, propriedades externamente visíveis desses elementos e os relacionamentos entre eles. A arquitetura define elementos de software, ou módulos, e envolve informação sobre como eles se relacionam uns com os outros. Uma arquitetura pode envolver mais de um tipo de estrutura, com diferentes tipos de elementos e de relacionamentos entre elementos. A arquitetura omite certas informações sobre os elementos que não pertencem às suas interações. As propriedades externamente visíveis indicam as suposições que os demais elementos podem fazer sobre um elemento, tais como serviços providos e características de qualidade esperadas. Assim, uma arquitetura é antes de tudo uma abstração de um sistema que suprime detalhes dos elementos que não afetam como eles são usados, como se relacionam, como interagem e como usam outros elementos. Na maioria das vezes, a arquitetura é usada para descrever aspectos estruturais de um sistema.

Em quase todos os sistemas modernos, elementos interagem com outros por meio de interfaces que dividem detalhes sobre um elemento em partes pública e privada. A arquitetura está preocupada com a parte pública dessa divisão. Detalhes privados, aqueles que têm a ver somente com a implementação interna, não são arquiteturais (BASS; CLEMENTS; KAZMAN, 2003).

Até o projeto arquitetônico, aspectos relacionados ao hardware e à plataforma de implementação ainda não foram tratados, já que a fase de análise pressupõe tecnologia perfeita. Este é o momento para resolver como um modelo ideal vai executar em uma plataforma restrita.

É importante realçar que não existe a solução perfeita. O projeto da arquitetura é uma tarefa de negociação, onde se faz compromissos entre soluções sub-ótimas. O modelo de arquitetura mapeia os requisitos essenciais da fase de análise em uma arquitetura técnica. Uma vez que muitas arquiteturas diferentes são possíveis, o propósito do projeto arquitetônico é escolher a configuração mais adequada. Além disso, fatores que transcendem aspectos puramente técnicos devem ser considerados.

Normalmente, o projeto da arquitetura é discutido à luz dos requisitos do sistema, ou seja, se os requisitos são conhecidos, então se pode projetar a arquitetura do sistema. Contudo, deve-se considerar o projeto arquitetônico como algo mais abrangente, envolvendo aspectos técnicos, sociais e de negócio. Todos esses fatores (e não somente os requisitos do sistema) influenciam a arquitetura de software. Esta, por sua vez, afeta o ambiente da organização (incluindo ambientes técnico, social e de negócio) que vai influenciar arquiteturas futuras, criando um ciclo de realimentação contínua. Por exemplo, se os projetistas encarregados do projeto de um novo sistema obtiveram bons resultados em projetos de sistemas anteriores usando uma particular abordagem de arquitetura, então é natural que eles tentem a mesma abordagem no novo projeto. Por outro lado, se suas experiências anteriores com essa abordagem foram desastrosas, os projetistas vão relutar em tentá-la outra vez, mesmo que ela se apresente como uma solução adequada. Assim, as escolhas são guiadas, também, pela formação e experiência dos projetistas (BASS; CLEMENTS; KAZMAN, 2003).

Outro fator que afeta a escolha da arquitetura é o ambiente técnico (ou plataforma de implementação) corrente. Muitas vezes, há para esse ambiente um conjunto dominante de padrões, práticas e técnicas que é aceito pela comunidade de arquitetos ou pela organização de desenvolvimento. Por fim, a arquitetura é influenciada também pela estrutura e natureza da organização de desenvolvimento (BASS; CLEMENTS; KAZMAN, 2003).

Assim, no projeto da arquitetura de software, projetistas são influenciados por requisitos para o sistema, estrutura e metas da organização de desenvolvimento, ambiente técnico disponível e por suas próprias experiências e formação. Além disso, os relacionamentos entre metas de negócio, requisitos de sistemas, experiência dos projetistas, arquiteturas e sistemas implantados geram diversos laços de realimentação que podem ser gerenciado pela organização.

Muitas pessoas têm interesse na arquitetura de software, tais como clientes, usuários finais, desenvolvedores, gerentes de projeto e mantenedores. Alguns desses interesses são conflitantes e o projetista frequentemente tem de mediar conflitos até chegar à configuração que atenda de forma mais adequada a todos os interesses. Neste contexto, arquiteturas de software são importantes principalmente porque (BASS; CLEMENTS; KAZMAN, 2003):

- Representam uma abstração comum do sistema que pode ser usada para compreensão mútua, negociação, consenso e comunicação entre os interessados. A arquitetura provê uma linguagem comum na qual diferentes preocupações podem ser expressas, negociadas e resolvidas em um nível que seja intelectualmente gerenciável.
- Manifestam as primeiras decisões de projeto. Essas decisões definem restrições sobre a implementação e a estrutura organizacional do projeto. A implementação tem de ser dividida nos elementos prescritos pela arquitetura. Os elementos têm de interagir conforme o prescrito e cada elemento tem de cumprir sua responsabilidade conforme ditado pela arquitetura. Também a estrutura organizacional do projeto, e às vezes até a estrutura da organização como um todo, torna-se amarrada à estrutura proposta pela arquitetura. Neste sentido, a arquitetura pode ajudar a obter estimativas e cronogramas

mais precisos, bem como pode ajudar na prototipagem do sistema. Além disso, a extensão na qual o sistema vai ser capaz de satisfazer os atributos de qualidade requeridos é substancialmente determinada pela arquitetura. Particularmente a manutenibilidade é fortemente afetada pela arquitetura. Uma arquitetura reparte possíveis alterações em três categorias: locais (confinadas em um único elemento), não locais (requerem a alteração de vários elementos, mas mantêm intacta a abordagem arquitetônica subjacente) e arquitetônicas (afetam a estrutura do sistema e podem requerer alterações ao longo de todo o sistema). Obviamente, alterações locais são as mais desejáveis e, portanto, uma arquitetura efetiva deve propiciar que as alterações mais prováveis sejam as mais fáceis de fazer.

- Constituem um modelo relativamente pequeno e intelectualmente compreensível de como o sistema é estruturado e como seus elementos trabalham em conjunto. Além disso, esse modelo é transferível para outros sistemas, em especial para aqueles que exibem requisitos funcionais e não funcionais similares, promovendo reúso em larga escala. Um desenvolvimento baseado na arquitetura frenquentemente enfoca a composição ou montagem de elementos que provavelmente foram desenvolvidos separadamente, até mesmo de forma independente. Essa composição é possível porque a arquitetura define os elementos que devem ser incorporados no sistema. Além disso, a arquitetura restringe possíveis substituições de elementos segundo a forma como eles interagem com o ambiente, recebem e entregam o controle, que dados consomem e produzem, como acessam esses dados e quais protocolos usam para se comunicar e compartilhar recursos.

É importante que o projetista seja capaz de reconhecer estruturas comuns utilizadas em sistemas já desenvolvidos, de modo a poder compreender as relações existentes e desenvolver novos sistemas como variações dos sistemas existentes. O entendimento de arquiteturas de software existentes permite que os projetistas avaliem alternativas de projeto. Neste contexto, uma representação da arquitetura é essencial para permitir descrever propriedades de um sistema complexo, bem como uma análise da arquitetura proposta (MENDES, 2002).

Muitas vezes, arquiteturas são representadas na forma de diagramas contendo caixas (representando elementos) e linhas (representando relacionamentos). Entretanto, tais diagramas não dizem muita coisa sobre o que são os elementos e como eles cooperam para realizar o propósito do sistema e, portanto, não capturam importantes informações de arquitetura (BASS; CLEMENTS; KAZMAN, 2003). Por exemplo, em uma visão de decomposição de módulos, é importante distinguir quando um módulo é decomposto em outros módulos e quando um módulo simplesmente usa outros módulos. Já em uma arquitetura cliente-servidor, é importante apontar quando um módulo é considerado cliente e quando ele é considerado servidor. Assim, idealmente, a representação de uma arquitetura deve incorporar informações acerca dos tipos dos elementos e dos relacionamentos.

6.1.3 Padrões (Patterns)

Todo projeto de desenvolvimento é, de alguma maneira, novo, na medida em que se quer desenvolver um novo sistema, seja porque ainda não existe um sistema para resolver o problema que está sendo tratado, seja porque há aspectos indesejáveis nos sistemas existentes. Isso não quer dizer que o projeto tenha que ser desenvolvido a partir do zero. Muito pelo contrário. A reutilização é um aspecto fundamental no desenvolvimento de software. Muitos sistemas previamente desenvolvidos são similares ao sistema em desenvolvimento e há muito conhecimento que pode ser reaplicado para solucionar questões recorrentes no projeto de

software. Os padrões (*patterns*) visam capturar esse conhecimento, procurando torná-lo mais geral e amplamente aplicável, desvinculando-o das especificidades de um determinado projeto ou sistema.

Um padrão é uma solução testada e aprovada para um problema geral. Diferentes padrões se destinam a diferentes fases do ciclo de vida: análise, arquitetura, projeto e implementação. Um padrão vem com diretrizes sobre quando usá-lo, bem como vantagens e desvantagens de seu uso. Um padrão já foi cuidadosamente considerado por outras pessoas e aplicado diversas vezes na solução de problemas anteriores de mesma natureza. Assim, tende a ser uma solução de qualidade, com maiores chances de estar correto e estável do que uma solução nova, específica, ainda não testada (BLAHA; RUMBAUGH, 2006).

Um padrão normalmente tem o formato de um par nomeado problema/solução, que pode ser utilizado em novos contextos, com orientações sobre com utilizá-lo nessas novas situações (LARMAN, 2007). O objetivo de um padrão de projeto é registrar uma experiência no projeto de software, que possa ser efetivamente utilizado por projetistas. Cada padrão sistematicamente nomeia, explica e avalia uma importante situação de projeto que ocorre repetidamente em sistemas (GAMMA et al., 1995).

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

Em geral, um padrão tem os seguintes elementos (GAMMA et al., 1995) (BUSCHMANN et al., 1996):

- **Nome**: identificação de uma ou duas palavras, utilizada para nomear o padrão.
- **Contexto**: uma situação que dá origem a um problema.
- **Problema**: explica o problema que surge repetidamente no dado contexto.
- **Solução**: descreve uma solução comprovada para o problema, incluindo os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. É importante observar que não descreve um particular projeto concreto ou implementação. Um padrão provê uma descrição abstrata de um problema de projeto e como uma organização geral de elementos resolve esse problema.
- **Consequências**: são os resultados e os comprometimentos feitos ao se aplicar o padrão.

No que concerne aos padrões relacionados à fase de projeto, há três grandes categorias a serem consideradas:

- Padrões Arquitetônicos: definem uma estrutura global do sistema. Um padrão arquitetônico indica um conjunto predefinido de subsistemas, especifica as suas responsabilidades e inclui regras e orientações para estabelecer os relacionamentos entre eles. São aplicados na atividade de projeto da arquitetura de software e podem ser vistos como modelos (*templates*) para arquiteturas de software concretas (BUSCHMANN et al., 1996).
- Padrões de Projeto (*Design Patterns*): atendem a uma situação específica de projeto, mostrando classes e relacionamentos, seus papéis e suas colaborações e também a distribuição de responsabilidades. Um padrão de projeto provê um esquema para refinar subsistemas ou componentes de sistema de software, ou os relacionamentos

entre eles. Ele descreve uma estrutura comumente recorrente de componentes que se comunicam, a qual resolve um problema de projeto geral dentro de um particular contexto (GAMMA et al., 1995).

- Idiomas: representam o nível mais baixo de padrões, endereçando aspectos tanto de projeto quanto de implementação. Um idioma é um padrão de baixo nível, específico de uma linguagem de programação, descrevendo como implementar aspectos particulares de componentes ou os relacionamentos entre eles usando as características de uma dada linguagem (BUSCHMANN et al., 1996).

6.1.4 Documentação de Projeto

Uma vez que o projeto de software encontra-se no núcleo técnico do processo de desenvolvimento, sua documentação tem grande importância para o sucesso do projeto e para a manutenção futura do sistema. Diferentes interessados vão requerer informações diferentes e a documentação de projeto é crucial para a comunicação. Analistas, projetistas e clientes vão precisar negociar para estabelecer prioridades entre requisitos conflitantes; programadores e testadores vão utilizar essa documentação para implementar e testar o sistema; gerentes de projeto vão usar informações da decomposição do sistema para definir e alocar equipes de trabalho; mantenedores vão recorrer a essa documentação na hora de avaliar e realizar uma alteração.

Uma vez que o projeto é um processo de refinamento, a sua documentação também deve prover representações em diferentes níveis de abstração. Além disso, o projeto de um sistema é uma entidade complexa que não pode ser descrita em uma única perspectiva. Ao contrário, múltiplas visões são essenciais e a documentação deve abranger aquelas visões consideradas relevantes. De fato, como muitas visões são possíveis, a documentação é uma atividade que envolve a escolha das visões relevantes, a documentação das visões selecionadas e a documentação de informações que se aplicam a mais do que uma visão (BASS; CLEMENTS; KAZMAN, 2003). A escolha das visões é dependente de vários fatores, dentre eles, do tipo de sistema sendo desenvolvido, dos atributos de qualidade considerados e da audiência da documentação de projeto. Diferentes visões realçam diferentes elementos de um sistema.

De maneira geral, o documento de projeto deve conter (BASS; CLEMENTS; KAZMAN, 2003):

- Informações gerenciais, tais como versão, responsáveis, histórico de alterações;
- Uma descrição geral do sistema;
- Uma lista das visões consideradas e informações acerca do mapeamento entre elas;
- Para cada visão, deve-se ter uma representação básica da visão, que pode ser gráfica, tabular ou textual, sendo a primeira a mais usual, sobretudo na forma de um diagrama UML. Se for usada uma representação gráfica não padronizada, deve-se ter uma legenda explicando a notação ou simbologia usada.

Além das informações anteriormente relacionadas, uma especificação de projeto deve:

- contemplar os requisitos contidos na especificação de requisitos, sendo que, muitas vezes, podem ser levantados novos requisitos, sobretudo requisitos não funcionais, durante a fase de projeto;

- ser um guia legível e compreensível para aqueles que vão codificar, testar e manter o software;
- prover um quadro completo do software, segundo uma perspectiva de implementação.

6.2 Projetando a Arquitetura de Software

Projetar a arquitetura de um software requer o levantamento de informações relativas à plataforma de computação do sistema, as quais se somarão ao conhecimento acerca dos requisitos funcionais e não funcionais, para dar embasar as decisões relativas à arquitetura do sistema que está sendo projetado. De maneira geral, o processo de projetar envolve, dentre outros, os seguintes passos:

1. Levantar informações acerca da plataforma de computação do sistema, incluindo linguagem de programação a ser adotada, mecanismo de persistência e necessidades de distribuição geográfica.
2. Com base nos requisitos, iniciar a decomposição do sistema em subsistemas, considerando preferencialmente uma decomposição pelo domínio do problema. Se na fase de análise já tiver sido estabelecida uma decomposição inicial em subsistemas, esta deverá ser utilizada. Neste momento, deve-se escolher um estilo arquitetônico (ou uma combinação adequada de estilos arquitetônicos) para organizar a estrutura geral do sistema.
3. Estabelecer uma arquitetura base, identificando tipos de módulos e tipos de relacionamentos entre eles, dados essencialmente pela combinação de estilos arquitetônicos escolhidos.
4. Alocar requisitos funcionais (casos de uso) e não funcionais aos componentes da arquitetura.
5. Avaliar a arquitetura, procurando identificar se ela acomoda os requisitos e restrições identificados.
6. Uma vez definida a arquitetura em seu nível mais alto, passar ao projeto de seus elementos. Padrões arquitetônicos são instrumentos muito valiosos para auxiliar o projeto dos componentes da arquitetura.

Em relação ao passo 2, um estilo arquitetônico que combina partições e camadas tende a ser um bom ponto de partida para a estruturação global de sistemas de informação.

- **Partições** podem ser derivadas a partir do domínio do problema, levando-se em conta funcionalidades coesas, visando à criação de subsistemas fracamente acoplados. Idealmente, alguma divisão em subsistemas deve ter sido feita na fase de análise e a mesma deve ser aqui preservada.
- As partições podem ser estruturadas em camadas e novamente um bom ponto de partida é considerar **camadas** típicas de um sistema de informação (Interface com o Usuário, Domínio do Problema e Gerência de Dados), mostradas na Figura 6.3.

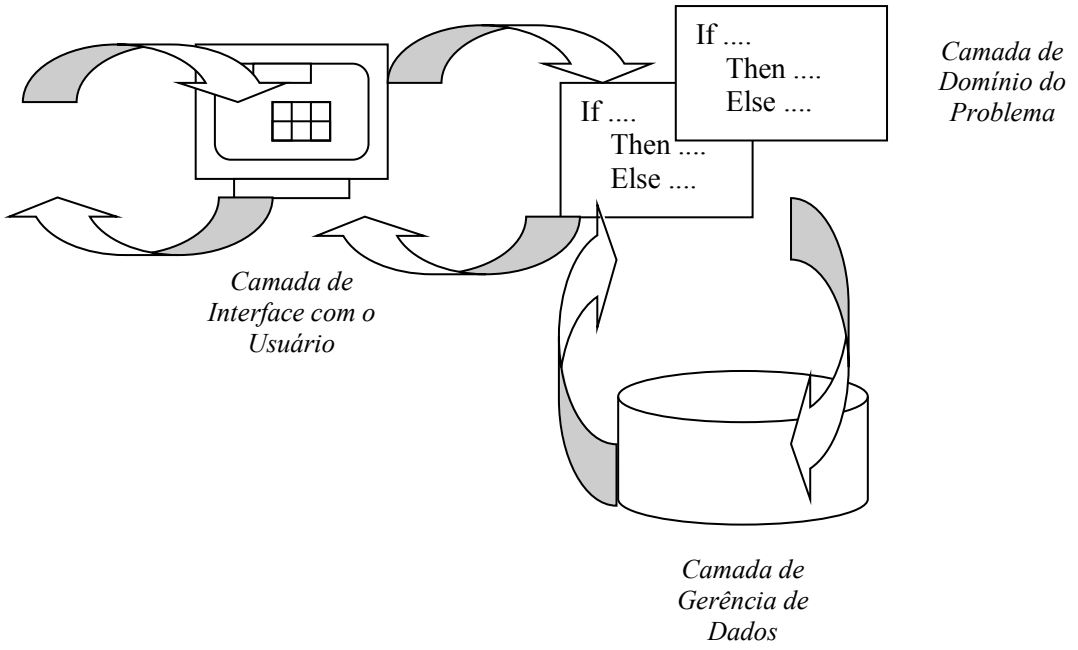

Figura 6.3 – Camadas típicas em sistemas de informação.

Nas próximas seções são apresentadas discussões relacionadas ao projeto de cada uma dessas camadas.

6.3 A Camada de Domínio do Problema

A camada de lógica de negócio engloba o conjunto de classes que vai realizar toda a lógica do sistema de informação. As demais camadas são derivadas ou dependentes da camada de domínio (WAZLAWICK, 2004) e, portanto, é interessante iniciar o projeto dos componentes da arquitetura do sistema pela camada da lógica de negócio.

Os modelos construídos na fase de análise são os principais insumos para o projeto dessa camada, em especial o modelo conceitual e o modelo de casos de uso. Os diagramas de classes da fase de análise serão a base para a construção dos diagramas de classes da fase de projeto. De fato, a versão inicial do modelo estrutural de projeto (diagramas de classes de projeto) da lógica de negócio será uma cópia do modelo conceitual estrutural. Durante o projeto, esse modelo será objeto de refinamento, visando incorporar informações importantes para a implementação, tais como distribuição de responsabilidades entre as classes (definição de métodos das classes) e definição de navegabilidades, visibilidades e tipos de dados. Além disso, alterações na estrutura do diagrama de classes podem ser necessárias para tratar requisitos não funcionais, tais como usabilidade e desempenho.

Para organizar a lógica de negócio, um bom ponto de partida são os padrões arquitetônicos relativos a essa camada. No contexto do desenvolvimento de Sistemas de Informação Orientados a Objetos, merecem destaque os padrões **Modelo de Domínio** e **Camada de Serviço**.

Uma questão bastante importante tratada por esses padrões é a distribuição de responsabilidades ao longo das classes que compõem o sistema. No mundo de objetos, uma funcionalidade é realizada através de uma rede de objetos interconectados, colaborando entre si. Objetos encapsulam dados e comportamento. O posicionamento correto do comportamento

na rede de objetos é um dos principais problemas a serem enfrentados durante o projeto da lógica de negócio. Neste contexto, um desafio a mais se coloca: que classes vão comportar as funcionalidades descritas pelos casos de uso? Duas formas básicas são comumente adotadas:

- Distribuir as responsabilidades para a execução dos casos de uso ao longo dos objetos do domínio do problema: essa abordagem é refletida no padrão Modelo de Domínio e considera que as funcionalidades relativas aos casos de uso do sistema estarão distribuídas nas classes previamente identificadas na fase de análise (Componente Domínio do Problema).
- Considerar que a lógica de negócio é, na verdade, composta por dois tipos de lógica: a lógica de domínio do problema (Componente de Domínio do Problema), que tem a ver puramente com as classes previamente identificadas na fase de análise; e lógica da aplicação (Componente de Gerência de Tarefas), que se refere às funcionalidades descritas pelos casos de uso. Esta segunda opção é a essência do padrão Camada de Serviço.

A seguir são apresentados os padrões Modelo de Domínio e Camada de Serviço. Depois, são apresentados o Componente Domínio do Problema, necessário tanto para o padrão Modelo de Domínio quanto para o padrão Camada de Serviço, e o Componente de Gerência de Tarefas, necessário apenas no padrão Camada de Serviço.

6.3.1 Padrões Arquitetônicos para o Projeto da Lógica de Negócio

Um importante aspecto do projeto da Camada de Lógica de Negócio diz respeito à organização das classes e distribuição de responsabilidades entre elas, o que vai definir, em última instância, os métodos de cada classe dessa camada. Nesse contexto, dos padrões arquitetônicos merecem destaque: Modelo de Domínio e Camada de Serviço.

De forma resumida, no padrão Modelo de Domínio, as responsabilidades são distribuídas nos objetos do domínio do problema e a lógica de aplicação é pulverizada nesses objetos, sendo que cada objeto tem uma parte da lógica que é relevante a ele. Já na abordagem do padrão Camada de Serviço, um conjunto de objetos controladores de casos de uso[22] fica responsável por tratar a lógica de aplicação, controlando o fluxo de eventos dentro do caso de uso. Grande parte da lógica de negócio ainda fica a cargo dos objetos do domínio do problema. Cabe aos controladores de casos de uso apenas centralizar o controle sobre a execução do caso de uso. Assim, a camada de serviço é construída, de fato, sobre a camada de domínio.

A seguir os padrões são apresentados em mais detalhes.

a) Padrão Modelo de Domínio

O padrão Modelo de Domínio preconiza que o próprio modelo de objetos do domínio incorpore dados e comportamento. Um modelo de domínio estabelece uma rede de objetos interconectados, onde cada objeto representa alguma entidade significativa no mundo real. Colocar um modelo de domínio em uma aplicação envolve inserir uma camada de objetos que modela a área de negócio da aplicação. Os objetos dessa camada vão ser abstrações de entidades do negócio que capturam regras que o negócio utiliza. Os dados e os processos são combinados para agrupar os processos próximos dos dados com os quais eles trabalham (FOWLER, 2003).

[22] Classes controladoras de caso de uso são classes que centralizam as interações no contexto de casos de uso específicos e não são consideradas controladores no sentido usado no padrão MVC.

Neste sentido, pode-se dizer que o padrão Modelo de Domínio captura a essência da orientação a objetos. Como benefícios, têm-se os benefícios propalados pela orientação a objetos, tais como evitar duplicação da lógica de negócio e gerenciar a complexidade usando *design patterns* clássicos.

Wazlawick (2004) propõe uma maneira interessante de aplicar esse padrão que consiste em considerar uma classe controladora do sistema no modelo de domínio do problema, relacionando-a a todos os conceitos independentes, correspondentes aos cadastros do sistema, ou seja, os elementos a serem cadastrados para a operação do sistema[23]. O fluxo de controle sempre inicia em uma instância da classe controladora. Essa classe recebe as requisições da interface e, para tratá-las, o controlador invoca métodos das classes do domínio do problema, em uma abordagem chamada de delegação. A delegação consiste em capturar uma operação que trata uma mensagem em um objeto e reenviar essa mensagem para um outro objeto associado ao primeiro que seja capaz de tratar a requisição. Neste caso, a execução é delegada para objetos do domínio do problema, procurando efetuar uma cadeia de delegação sobre as linhas de visibilidade das associações já existentes, até se atingir um objeto capaz de atender à requisição. Novas linhas de visibilidade só devem ser criadas quando isso for estritamente necessário. Deste modo, mantém-se fraco o acoplamento entre as classes, conforme preconiza o padrão de projeto acoplamento fraco (LARMAN, 2004). Assim, partindo-se do controlador do sistema, deve-se identificar qual o objeto a ele relacionado que melhor pode tratar a requisição e delegar essa responsabilidade a ele. Esse objeto, por sua vez, vai colaborar com outros objetos para a realização da funcionalidade.

De maneira geral, um objeto só deve mandar mensagens para outros objetos que estejam a ele associados ou que foram passados como parâmetro no método que está sendo executado. Nessa abordagem, deve-se evitar obter um objeto como retorno de um método (visibilidade local) para mandar mensagens a ele (WAZLAWICK, 2004), conforme apontado pelo padrão "não fale com estranhos" (LARMAN, 2004).

b) Padrão Camada de Serviço

A motivação principal para esse padrão é o fato de algumas funcionalidades não serem facilmente distribuídas nas classes de domínio do problema, principalmente aquelas que operam sobre vários objetos. Uma possibilidade para resolver tal problema é pulverizar esse comportamento ao longo dos vários objetos do domínio do problema, conforme preconiza o padrão Modelo de Domínio. Contudo, esta pode não ser uma boa solução segundo uma perspectiva de manutenibilidade. Uma alteração em tal funcionalidade poderia afetar diversos objetos e, assim, ser difícil de ser incorporada.

Uma abordagem alternativa consiste em criar classes controladoras de casos de uso (gerenciadores ou coordenadores de tarefas)[24], responsáveis pela realização de tarefas sobre um determinado conjunto de objetos. Tipicamente, esses gerenciadores agem como aglutinadores, unindo outros objetos para dar forma a um caso de uso. Consequentemente, gerenciadores de tarefa são normalmente encontrados diretamente a partir dos casos de uso.

---

[23] Vale ressaltar que, embora a classe controladora de sistema seja modelada no diagrama de classes do domínio do problema, ela corresponde, de fato, ao pacote de interface com o usuário, tendo em vista que ela é um controlador no sentido adotado pelo padrão MVC.

[24] Vale lembrar que classes gerenciadoras de casos de uso não são controladores no sentido do padrão MVC.

Os tipos de funcionalidade tipicamente atribuídos a gerenciadores de tarefa incluem comportamento relacionado a transações e sequências de controle específicas de um caso de uso.

O padrão Camada de Serviço (FOWLER, 2003) define uma fronteira da aplicação usando uma camada de serviços que estabelece um conjunto de operações disponíveis e coordena as respostas da aplicação para cada uma das operações. A camada de serviço encapsula a lógica de negócio da aplicação, controlando transações e coordenando respostas na implementação de suas operações. A argumentação em favor desse padrão é que misturar lógica de domínio e lógica de aplicação nas mesmas classes torna essas classes menos reutilizáveis transversalmente em diferentes aplicações, bem como pode dificultar a manutenção da lógica de aplicação, uma vez que a lógica dos casos de uso não é diretamente perceptível em nenhuma classe.

A identificação das operações necessárias na camada de serviço é fortemente apoiada nos casos de uso do sistema. Uma opção é considerar que cada caso de uso vai dar origem a uma classe de serviços, dita classe controladora de caso de uso. Por exemplo, um caso de uso de cadastro, envolvendo funcionalidades de inclusão, alteração, consulta e exclusão, pode ser mapeado em uma classe com operações para tratar essas funcionalidades. Contudo, não há uma prescrição clara, apenas heurísticas. Para uma aplicação relativamente pequena, pode ser suficiente ter uma única classe provendo todas as operações. Para sistemas maiores, compostos de vários subsistemas, pode-se ter uma classe por subsistema (FOWLER, 2003).

A camada de serviço pode ser implementada de duas formas básicas:

- Fachada de Domínio: nessa abordagem, a camada de serviço é implementada como um conjunto fino de fachadas sobre um modelo de domínio (no sentido do padrão Modelo de Domínio). As classes implementando as fachadas não implementam nenhuma lógica de negócio. Esta é implementada pela camada de domínio. As fachadas estabelecem apenas uma fronteira e um conjunto de operações através das quais suas camadas clientes (tipicamente interfaces com o usuário e pontos de integração com outros sistemas) vão interagir com a lógica de negócio.
- Script de Operação: nessa abordagem, a camada de serviço é implementada como um conjunto de classes que implementa diretamente a lógica de aplicação. Contudo, vale frisar que, para implementar a lógica de aplicação, as classes dessa camada delegam diversas responsabilidades para os objetos do domínio do problema.

Não se deve confundir uma abordagem de Camada de Serviços com uma abordagem de Modelo de Domínio Anêmico (*Anemic Domain Model*) (FOWLER, 2003a), na qual os objetos do domínio do problema apresentam comportamento vazio. Nessa abordagem, as classes de análise são divididas em classes de dados (ditos objetos de valor – *Value Objects* – VOs) e classes de lógica (ditos objetos de negócio – *Business Objects* – BOs), que separam o comportamento do estado dos objetos. Os VOs têm apenas o comportamento básico para alterar e manipular seu estado (métodos construtor e destrutor e métodos *get* e *set*). Os BOs ficam com os outros comportamentos, tais como cálculos, validações e regras de negócio. De maneira geral, a abordagem de Modelo de Domínio Anêmico deve ser evitada, sendo, por isso, considerada um anti-padrão. Essa abordagem tem diversos problemas. Primeiro, não há encapsulamento, já que dificilmente um VO vai ser utilizado apenas por um BO. Segundo, a vantagem de se ter um modelo de domínio rico é anulada, já que a proximidade com as abstrações do mundo real é destruída. No mundo real não existe lógica de um lado e dados de outro, mas sim ambos combinados em um mesmo conceito. Outro problema é a manutenção de

um sistema construído desta maneira. Os BOs possuem um acoplamento muito alto com os VOs e a mudança em um afeta drasticamente o outro (FRAGMENTAL, 2007).

Uma maneira de se implementar o padrão Camada de Serviço consiste em ter uma ou mais classes controladoras de casos de uso (veja discussão na seção 4.4), as quais encapsulam a lógica da aplicação. Para realizar um caso de uso, a classe controladora de caso de uso invoca métodos da camada de domínio do problema, tal como ocorre no padrão Modelo de Domínio. A diferença entre os dois padrões reside, neste caso, no fato da classe gerenciadora de tarefa centralizar o controle do caso de uso, evitando delegar responsabilidades a classes que não têm como tratá-las. Para tal, aceita-se que a classe gerenciadora de tarefa obtenha um objeto como retorno de um método (visibilidade local) e mande mensagens a ele. Assim, a classe gerenciadora de tarefa pode ter referência a diversos objetos do domínio, tipicamente aqueles envolvidos na realização do caso de uso correspondente.

6.3.2 Componente de Domínio do Problema (CDP)

No projeto orientado a objetos, os modelos conceituais estruturais (diagramas de classes) produzidos na fase de análise fazem parte do Componente de Domínio do Problema (CDP). Como ponto de partida para a elaboração do diagrama de classes do CDP, deve-se utilizar uma cópia do diagrama de classes de análise. A partir dessa cópia, alterações serão feitas para incorporar as decisões de projeto. Vale ressaltar que o trabalho deve ser efetuado em uma cópia, mantendo o modelo conceitual original intacto para efeito de documentação e manutenção do sistema.

Para se poder conduzir o projeto do CDP de maneira satisfatória, algumas informações acerca da plataforma de implementação são essenciais, dentre elas a linguagem de programação e o mecanismo de persistência de objetos a serem adotados. Além disso, informações relativas aos requisitos não funcionais e suas prioridades são igualmente vitais para se tomar decisões importantes relativas ao projeto do CDP.

As alterações básicas a serem incorporadas em um diagrama de classes do CDP são:

- *Adição de informações relativas a tipos de dados de atributos*: Na fase de análise, é comum não especificar tipos de dados para atributos. Na fase de projeto, contudo, essa é uma informação imprescindível. De modo geral, os atributos são mapeados em variáveis de um tipo de dados provido pela linguagem de implementação, tal como string, inteiro ou booleano. Contudo, muitas vezes, atributos podem dar origem a novas classes (ou tipos de dados enumerados) para atender a requisitos de qualidade, tais como usabilidade, manutenibilidade e reusabilidade.

- *Adição de navegabilidades nas associações*: Na fase de análise, as associações são consideradas navegáveis nos dois sentidos. O mesmo não ocorre na fase de projeto. Pode-se definir que certas associações são navegáveis apenas em um sentido, indicando que apenas um dos objetos terá uma referência para o outro (ou para coleções de objetos, no caso de associações com multiplicidade *). Esta decisão pode ser influenciada pelo mecanismo de persistência de objetos a ser adotado, sobretudo quando esse mecanismo envolve um Sistema Gerenciador de Banco de Dados Relacional.

- *Adição de informações de visibilidade de atributos e associações*: De maneira geral, na fase de análise não se especifica a visibilidade de atributos e associações. Como discutido no item acima, as associações são tipicamente consideradas navegáveis e visíveis nos dois

sentidos. Já os atributos são considerados públicos. Porém essa não é uma boa estratégia para a fase de projeto. Ocultar informações é um importante princípio de projeto. Assim, atributos só devem poder ser acessados pela própria classe ou por suas subclasses. Uma classe não deve ter acesso aos atributos de uma classe a ela associada. Como consequência disso, cada classe deve ter operações para consultar (tipicamente nomeadas como *get*) e atribuir / alterar valor (normalmente nomeada como *set*) de cada um de seus atributos e associações navegáveis. Essas operações, contudo, não precisam ser mostradas no diagrama de classes, visto que elas podem ser deduzidas pela própria existência dos atributos e associações (WAZLAWICK, 2004).

- *Adição de métodos às classes*: Muitas vezes, as classes de um diagrama de classes de análise não têm informação acerca das suas operações. Mesmo quando elas têm, essa informação pode ser insuficiente, tendo em vista que é no projeto que se decide efetivamente como abordar a distribuição de responsabilidades para a realização de funcionalidades. Assim, durante o projeto do CDP atenção especial deve ser dada à definição de métodos nas classes. Para apoiar esta etapa, diagramas de sequência podem ser utilizados para modelar a interação entre objetos na realização de funcionalidades do sistema. A escolha de um padrão arquitetônico para o projeto do CDP também tem influência na distribuição de responsabilidades. Vale ressaltar que já se assume que algumas operações, consideradas básicas, existem e, portanto, não precisam ser representadas no diagrama de classes. Essas operações são as operações de criação e destruição de instâncias, além das operações de consulta e atribuição / alteração de valores de atributos e associações, conforme discutido no item anterior. No diagrama de classes devem aparecer apenas os métodos que não podem ser deduzidos (WAZLAWICK, 2004).

- *Eliminação de classes associativas*: Caso o diagrama de classes de análise contenha classes associativas, recomenda-se substituí-las por classes normais, criando novas associações. Isso é importante, pois as linguagens de programação não têm construtores capazes de implementar diretamente esses elementos de modelo.

Além das alterações básicas a que todos os diagramas de classes do CDP estarão sujeitos, outras fontes de alteração incluem:

- *Reutilizar projetos anteriores e classes já programadas*: é importante que na fase de projeto seja levada em conta a possibilidade de se reutilizar classes já projetadas e programadas (desenvolvimento com reúso), bem como a possibilidade de se desenvolver classes para reutilização futura (desenvolvimento para reúso). Tipicamente, ajustes feitos para incorporar tais classes envolvem alterações na estrutura do modelo, podendo atingir hierarquias de generalização-especialização do modelo, de modo a tratar as classes do domínio do problema como subclasses de classes de biblioteca pré-existentes. Também ao incorporar um padrão de projeto (*design pattern*), muito provavelmente a estrutura do diagrama de classes de projeto sofrerá alterações.

- *Ajustar hierarquias de generalização-especialização*: muitas vezes, as hierarquias de herança da fase de análise não são adequadas para a fase de projeto. Dentre os fatores que podem provocar mudanças na hierarquia de herança destacam-se o mecanismo de herança suportado pela linguagem de programação a ser usada na implementação e a definição de operações.

    - *Ajustar hierarquias de generalização-especialização para adequação ao mecanismo de herança suportado pela linguagem de programação a ser usada na implementação*: se, por exemplo, o modelo de análise envolve herança múltipla e a linguagem de implementação não oferece tal recurso, alterações no modelo são necessárias. Quando se estiver avaliando hierarquias de classes para eliminar

relações de herança múltipla, deve-se considerar se uma abordagem de delegação não é mais adequada do que o estabelecimento de uma relação de herança.

- *Ajustar hierarquias de generalização-especialização para aproveitar oportunidades decorrentes da definição de operações*: as definições de operações nas classes podem também conduzir a alterações na hierarquia de generalização-especialização. De fato, pode ser que durante a fase de análise não sejam exploradas todas as oportunidades de herança. É útil reexaminar o diagrama de projeto procurando observar se determinadas classes têm comportamento parcialmente comum, abrindo-se espaço para a criação de uma superclasse encapsulando as propriedades (atributos e operações) compartilhadas, abstraindo o comportamento comum. Conforme discutido anteriormente, a reutilização pode ser um fator motivador para a criação de novas superclasses. Contudo, deve-se tomar cuidado com a refatoração da hierarquia de classes. Criar uma nova classe para abstrair comportamento comum somente se justifica quando há, de fato, uma relação de subtipo entre as classes existentes e a nova classe criada; ou seja, pode-se dizer que a subclasse é semanticamente um subtipo da superclasse. Não se deve alterar a hierarquia de classes simplesmente para herdar uma parte do comportamento, quando as classes envolvidas não guardam entre si uma relação efetivamente de subtipo, em uma abordagem de herança de implementação (BLAHA; RUMBAUGH, 2006).

- *Ajustar o modelo para melhorar o desempenho*: Visando melhorar o desempenho do sistema, o projetista pode alterar o diagrama de classes do CDP para melhor acomodar os ajustes necessários. Atributos e associações redundantes podem ser adicionados para evitar recomputação, bem como podem ser criadas novas classes para registrar estados intermediários de um processo.

- *Ajustar o modelo para facilitar o projeto de interfaces com o usuário amigáveis*: com o objetivo de incorporar o atributo de qualidade usabilidade, pode ser importante considerar novas classes (ou tipos enumerados de dados) que facilitem a apresentação de listas para seleção do usuário.

- *Ajustar o modelo para incorporar aspectos relacionados à segurança*: táticas como autenticação e autorização requerem novas funcionalidades que, por sua, vez, requerem novas classes do CDP. Em casos como esse, pode ser útil separar as classes relativas a essas funcionalidades em um novo pacote, visando ao reúso.

Além dos ajustes discutidos anteriormente, vários deles relacionados a atributos de qualidade (a saber, reusabilidade, desempenho, usabilidade e segurança), o CDP pode ser alterado, ainda, para comportar outros requisitos não funcionais, tais como testabilidade, confiabilidade etc.

Como dito anteriormente, O CDP é um componente obrigatório, tanto quando se adota o padrão Modelo de Domínio quanto quando se adota o padrão Camada de Serviço. No padrão Modelo de Domínio, o CDP é a própria camada de lógica de negócio, tendo em vista que não há classes gerenciadoras de tarefas (controladoras de casos de uso). No caso do padrão Camada de Serviço, além do CDP, a camada de lógica de negócio tem outro componente, o Componente de Gerência de Tarefas, discutido a seguir.

6.3.3 Componente de Gerência de Tarefas

O Componente de Gerência de Tarefas (CGT) compreende a definição das classes gerenciadoras de tarefas e seu projeto está intimamente relacionado ao modelo de casos de uso.

Em um esboço preliminar, pode-se atribuir um gerenciador de tarefa para cada caso de uso, sendo que os seus fluxos de eventos principais dão origem a operações da classe que representa o caso de uso (classe controladora de caso de uso). Se a abordagem de Script de Operação do padrão Camada de Serviço for adotada, a manutenibilidade pode ser facilitada, uma vez que, detectado um problema em um caso de uso, é fácil identificar a classe que trata do mesmo. Um possível problema, contudo, é o desempenho: para sistemas grandes, com muitos casos de uso, haverá muitas classes de gerência de tarefa e, para realizar uma tarefa complexa, pode ser necessária muita comunicação entre essas classes.

Uma solução diametralmente oposta consiste em definir uma única classe de aplicação para todo o sistema. Neste caso, os cenários de todos os casos de uso dão origem a operações dessa classe. Fica evidente que, exceto para sistemas muito pequenos, essa classe tende a ter muitas operações. Caso a abordagem de Script de Operação do padrão Camada de Serviço seja adotada, essa classe será extremamente complexa e, portanto, essa opção tende a não ser prática. Quando a abordagem de Fachada de Domínio do padrão Camada de Serviço é utilizada, ainda que a classe gerenciadora de tarefas tenha muitas operações, isso pode não ser um problema efetivamente, uma vez que essa classe apenas delega a responsabilidade pela execução das operações para objetos do domínio do problema.

No caso da abordagem de Script de Operação do padrão Camada de Serviço, normalmente, uma solução intermediária entre as duas anteriormente apresentadas conduz a melhores resultados. Nessa abordagem, casos de uso complexos são designados a classes de gerência de tarefas específicas. Casos de uso mais simples e de alguma forma relacionados são tratados por uma mesma classe de gerência de tarefas.

O conjunto de tarefas a serem apoiadas pelo sistema oferece um recurso bastante útil para a definição das janelas, menus e outros componentes de interface com o usuário necessários para cada uma dessas tarefas. Assim os projetos dos componentes de gerência de tarefa e de apresentação (seção adiante) estão bastante relacionados e devem ser realizados conjuntamente, uma vez que, muitas vezes, são as tarefas que determinam a necessidade de elementos de interface com o usuário para sua execução.

6.4 A Camada de Interface com o Usuário (CIU)

Sistemas, em especial os sistemas de informação, são desenvolvidos para serem utilizados por pessoas. Assim, um aspecto fundamental no projeto de sistemas é a interface com o usuário (IU). O projeto da IU estabelece uma forma de comunicação entre as pessoas e o sistema computacional. A IU define como um usuário comandará o sistema e como o sistema apresentará as informações a ele.

A camada de IU envolve dois tipos de funcionalidades:

- Visão: refere-se aos objetos gráficos usados na interação com o usuário;
- Controle de Interação: diz respeito ao controle da lógica da interface, envolvendo a ativação dos objetos gráficos (p.ex., abrir ou fechar uma janela, habilitar ou desabilitar um item de menu etc.) e o disparo de ações.

Um dos princípios fundamentais para um bom projeto de software é a separação da apresentação (camada de IU) da lógica de negócio. Essa separação é importante por diversas razões, dentre elas (FOWLER, 2003):

- O projeto de IU e o projeto da lógica de negócio tratam de diferentes preocupações. No primeiro, o foco está nos mecanismos de interação e em como dispor uma boa IU. O segundo concentra-se em conceitos e políticas de negócio.
- Usuários podem querer ver as mesmas informações de diferentes maneiras (p.ex., usando diferentes interfaces, tais como interfaces ricas de sistemas *desktop*, navegadores *Web*, interfaces de linha de comando etc.). Neste contexto, separar a IU da lógica de negócio permite o desenvolvimento de múltiplas apresentações.
- Objetos não visuais são geralmente mais fáceis de testar do que objetos visuais. Ao separar objetos da lógica de negócio de objetos de IU, é possível testar os primeiros sem envolver os últimos.

Dada a importância dessa separação, é importante usar algum padrão arquitetônico que trabalhe essa separação, tal como o padrão **Modelo-Visão-Controlador** (**MVC**) (FOWLER, 2003).

6.4.1 O Padrão Modelo-Visão-Controlador (MVC)

O padrão Modelo-Visão-Controlador (MVC) considera três papéis relacionados à interação humano-computador. O *modelo* refere-se aos objetos que representam alguma informação sobre o negócio e corresponde, de fato, a objetos da camada de Lógica de Negócio. A *visão* refere-se à entrada e a exibição de informações na IU. Qualquer requisição é tratada pelo terceiro papel: o controlador. Este pega a entrada do usuário, envia uma requisição para a camada de lógica de negócio, receber sua resposta e solicita que a visão se atualize conforme apropriado. Assim, a IU é uma combinação de visão e controlador (FOWLER, 2003). Em outras palavras, elementos da visão representam informações de modelo e as exibem ao usuário, que pode enviar, por meio da visão, requisições ao sistema. Essas requisições são tratadas pelo controlador, que as repassa para classes do modelo. Uma vez alterado o estado dos elementos do modelo, o controlador pode, se apropriado, selecionar outros ou alterar elementos de visão a serem exibidos ao usuário. Assim, o controlador situa-se entre o modelo e a visão, isolando-os um do outro.

Neste ponto é importante distinguir o controlador do padrão MVC do controlador de caso de uso do Componente de Gerência de Tarefas. Este último representa uma classe da lógica de negócio, que encapsula a lógica de um caso de uso. Já um controlador do padrão MVC é um controlador de interação, ou seja, ele controla a lógica de interface, abrindo e fechando janelas, habilitando ou desabilitando botões, enviando requisições etc. Para diferenciá-los, neste texto utilizamos o termo controlador de interação para designar os controladores de interface.

O padrão MVC trabalha dois tipos de separação. Primeiro, separa a apresentação (visão) da lógica de negócio (modelo), conforme advogado pelas boas práticas de projeto. Segundo, mantém também separados o controlador e a visão. Essa segunda separação (entre a visão e o controlador) é menos importante que a primeira (entre a visão e a lógica de negócio). A maioria dos sistemas tem um único controlador por visão e, por isso, a separação entre a visão e o controlador muitas vezes não é feita. Em sistemas de interfaces ricas *desktop* ela é muitas vezes desprezada. Contudo, em interfaces Web, essa separação é comum, já que a parte de visão *front*

end é naturalmente separada do controlador. De fato, a maioria dos padrões de projeto de interfaces Web é baseada nesse princípio (FOWLER, 2003). A Figura 6.4 mostra um diagrama de pacotes ilustrando o padrão MVC.

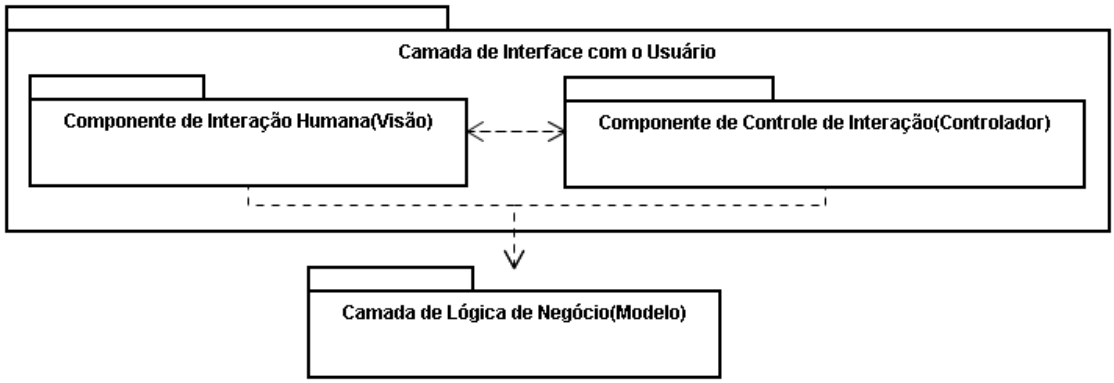

Figura 6.4 – O Padrão MVC.

A separação entre visão e controlador dá origem a dois tipos de classes que podem ser organizados em dois pacotes na camada de interface com o usuário: o Componente de Interação Humana (CIH), que é responsável pelas interfaces com o usuário propriamente ditas (janelas, painéis, botões, menus etc.) e representa a visão no modelo MVC; e o Componente de Controle de Interação (CCI), que é responsável por controlar a interação, recebendo requisições da interface, disparando operações da lógica de negócio e atualizando a visão com base no retorno dessas operações. O CCI é, portanto, o controlador do modelo MVC.

É importante frisar que, mesmo quando se opta por não fazer a separação física em pacotes de visão e controlador, é útil ter classes distintas para desempenhar esses papéis. As classes controladoras de interação devem ser marcadas com o estereótipo <<control>> para diferenciá-las das classes de visão, que devem ser marcadas com o estereótipo <<boundary>>.

No que se refere à interação entre as camadas de IU e Lógica de Negócio (modelo), ela se dá de maneiras distintas em função do padrão arquitetônico adotado nesta última. Quando o padrão Modelo de Domínio é adotado, os controladores de interação enviam as requisições diretamente para os objetos do domínio do problema (CDP), uma vez que neste caso não existem objetos gerenciadores de tarefa (CGT). Quando o padrão Camada de Serviço é adotado, as requisições dos controladores de interação são enviadas para os objetos gerenciadores de tarefas (CGT).

6.4.2 Componente de Interação Humana (CIH)

A porção do sistema que lida com a visão da interface com o usuário deve ser mantida tão independente e separada do resto da arquitetura do software quanto possível. Aspectos de interface com o usuário provavelmente serão alvo de alterações ao longo da vida do sistema e essas alterações devem ter um impacto mínimo nas demais partes do sistema.

O Componente de Interação Humana (CIH) trata do projeto da interação humano-computador, definindo formato de janelas, formulários, relatórios, entre outros. Durante o projeto do CIH, é muito útil construir protótipos, de modo a apoiar a seleção e o desenvolvimento dos mecanismos de interação a serem usados.

Como discutido anteriormente, o ponto de partida para o projeto da CIH é o modelo de casos de uso e as descrições de atores e casos de uso. Com base nos casos de uso, deve-se projetar uma hierarquia de comandos, definindo barras de menus, menus *pull-down*, ícones etc., que levem à execução dos casos de uso quando acionados pelo usuário. A hierarquia de comandos deve respeitar convenções e estilos existentes com os quais o usuário já esteja familiarizado. Note que a hierarquia de comandos é, de fato, um meio de apresentar ao usuário as várias funcionalidades disponíveis no sistema. Assim, a hierarquia de comandos deve permitir o acesso aos casos de uso do sistema.

Uma vez definida a hierarquia de comandos, as interações detalhadas entre o usuário e o sistema devem ser projetadas. Neste momento, é útil observar atentamente táticas de usabilidade, discutidas logo adiante.

Normalmente, não é necessário projetar as classes básicas de interfaces gráficas com o usuário. Existem vários ambientes de desenvolvimento de interfaces, oferecendo classes reutilizáveis (janelas, ícones, botões etc.) e, portanto, basta especializar as classes e instanciar os objetos que possuem as características apropriadas para o problema em questão. Hierarquias de classes de visão podem ser desenvolvidas visando à uniformidade da apresentação e ao reúso.

Táticas de Usabilidade

Diversas táticas relativas à usabilidade podem ser aplicadas durante o projeto de IU. Algumas dessas táticas gerais incluem:

- Facilidade de Ajuda: Ajuda é fundamental para os usuários, sobretudo aqueles novatos ou conhecedores, mas esporádicos. Para projetar adequadamente uma facilidade de ajuda é necessário definir, dentre outros:
    - quando a ajuda estará disponível e para que funções do sistema;
    - como ativar (botão, tecla de função, menu);
    - como representar (janela separada, local fixo da tela);
    - como retornar à interação normal (botão, tecla de função);
    - como estruturar a informação (estrutura plana, hierárquica, hipertexto).
- Mensagens de Erro e Avisos: Mais até do que a ajuda, as mensagens de erro e avisos são fundamentais para uma boa interação humano-computador. Ao definir mensagens de erro e avisos considere as seguintes diretrizes:
    - Descreva o problema com um vocabulário passível de entendimento pelo usuário.
    - Sempre que possível, proveja assistência para recuperar o erro.
    - Quando for o caso, indique as consequências negativas do erro.
    - Para facilitar a percepção da mensagem por parte do usuário, pode ser útil que a mesma seja acompanhada de uma dica visual ou sonora.
    - Não censure o usuário.

- Tipos de Comandos: Diferentes grupos de usuários têm diferentes necessidades de interação. Em muitas situações é útil prover ao usuário mais de uma forma de interação. Nestes casos, é necessário definir e avaliar:
    - se toda opção de menu terá um comando correspondente;
    - a forma do comando, tais como controle de sequência (p.ex., ^Q), teclas de função (p.ex., F1) e comandos digitados;
    - quão difícil é aprender e lembrar o comando;
    - possibilidade de customização de comandos (macros);
    - padrões para todo sistema e conformidade com outros padrões, tal como o definido pelo sistema operacional ou por produtos de software tipicamente utilizados pelos usuários.
- Tempo de Resposta: É importante mostrar o progresso do processamento para os usuários, principalmente para eventos com tempo de resposta longo ou com grande variação de tempos de resposta.

Levando-se em conta princípios gerais de projeto de IU, algumas orientações adicionais devem ser consideradas, dentre elas:

- Seja consistente. Use formatos consistentes para seleção de menus, entrada de comandos, apresentação de dados etc.
- Ofereça retorno significativo ao usuário.
- Peça confirmação para ações destrutivas, tais como ações para apagar ou sobrepor informações, terminar a seção corrente do aplicativo.
- Permita reversão fácil da maioria das ações (função *Desfazer*).
- Reduza a quantidade de informação que precisa ser memorizada entre ações.
- Busque eficiência no diálogo (movimentação, teclas a serem apertadas).
- Trate possíveis erros do usuário. O sistema deve se proteger de erros, casuais ou não, provocados pelo usuário.
- Classifique atividades por função e organize geograficamente a tela de acordo. Menus do tipo *pull-down* são uma boa opção.
- Proveja facilidades de ajuda sensíveis ao contexto.
- Use verbos de ação simples ou frases curtas para nomear funções e comandos.

No que se refere à apresentação de informações, considere as seguintes diretrizes:

- Mostre apenas informações relevantes ao contexto corrente.
- Use formatos de apresentação que permitam assimilação rápida da informação, tais como gráficos e figuras.
- Use rótulos consistentes, abreviaturas padrão e cores previsíveis.
- Produza mensagens de erro significativas.

- Projete adequadamente o layout de informações textuais. Leve em consideração o bom uso de letras maiúsculas e minúsculas, identação, agrupamento de informações etc.
- Separe diferentes tipos de informação. Painéis ou mesmo janelas podem ser usadas para este fim.
- Use formas de representação análogas às do mundo real para facilitar a assimilação da informação. Para tal considere o uso de figuras, cores etc.

No que se refere à entrada de dados, considere as seguintes diretrizes:

- Minimize o número de ações de entrada requeridas e possíveis erros. Para tal considere a seleção de dados a partir de um conjunto pré-definido de valores de entrada, o uso de valores *default* e macros etc.
- Mantenha consistência entre apresentação e entrada de dados; ou seja, mantenha as mesmas características visuais, dentre elas tamanho do texto, cor e localização.
- Permita ao usuário customizar a entrada para seu uso, quando possível, dando-lhe liberdade para definir comandos customizados, dispensar algumas mensagens de aviso e verificações de ações, dentre outros.
- Flexibilize a interação, permitindo afiná-la ao modo de entrada preferido do usuário (comandos, botões, *plug-and-play*, digitação etc.).
- Desative comandos inapropriados para o contexto das ações correntes.
- Proveja ajuda significativa para assistir as ações de entrada de dados.
- Nunca requeira que o usuário entre com uma informação que possa ser adquirida automaticamente pelo sistema ou computada por ele.

6.4.3 Componente de Controle de Interação (CCI)

O Componente de Controle de Interação (CCI) trata das classes responsáveis por controlar a interação (ativação / desativação dos objetos do CIH) e enviar requisições para os objetos da Lógica de Negócio.

Em sistemas rodando em plataforma *desktop*, deve haver pelo menos uma classe controladora, dita classe controladora de sistema, representando o sistema como um todo. Os objetos dessa classe representam as várias sessões (execuções) do sistema. Neste caso, é necessário levar em conta, ainda, quantos executáveis devem ser gerados para o sistema. Se mais do que um for necessário, cada executável terá de dar origem a uma classe controladora. Esta, contudo, é apenas uma abordagem possível. Analogamente ao projeto do CGT (veja seção 4.4), é possível definir um número arbitrário de controladores de interação. Uma opção em linha com o projeto da CGT é definir um controlador de interação para cada caso de uso. Contudo, uma vez que as classes controladoras de interação tendem a ser muito simples, essa abordagem pode ser exagerada. Assim, ainda que haja uma analogia entre o projeto do CCI e o projeto do CGT, as motivações são bastante diferentes e a escolha dos controladores de interação tende a ser diferente da escolha dos gerenciadores de tarefas.

6.5 A Camada de Gerência de Dados (CGD)

A maioria dos sistemas requer alguma forma de armazenamento de dados. Para tal, há várias alternativas, dentre elas a persistência em arquivos e bancos de dados. Em especial os sistemas de informação envolvem grandes quantidades de dados e fazem uso de sistemas gerenciadores de bancos de dados (SGBDs). Há diversos tipos de SGBDs, dentre eles os relacionais e os orientados a objetos, sendo os primeiros os mais utilizados atualmente no desenvolvimento de sistemas de informação.

Quando SGBDs Relacionais são utilizados, é necessário um mapeamento entre as estruturas de dados dos modelos orientado a objetos e relacional, de modo que objetos possam ser armazenados em tabelas. Dentre as principais diferenças entre esses modelos, destacam-se as diferentes formas como objetos e tabelas tratam ligações e na ausência do mecanismo de herança no modelo relacional. Essas diferenças levam à necessidade de reversões das estruturas de dados entre objetos e tabelas, tratadas como mapeamento objeto-relacional.

Além das diferenças estruturais, outros aspectos têm de ser tratados durante o projeto da persistência, dentre eles o modo como a camada de lógica de negócio se comunica com o banco de dados, o problema comportamental que diz respeito a como obter vários objetos do banco e como salvá-los, e o tratamento de conexões com o banco de dados e transações (FOWLER, 2003).

É importante enfatizar que muitos desses problemas são tratados por *frameworks* de persistência de objetos em bancos de dados relacionais (ou *frameworks* de mapeamento objeto-relacional), tal como o Hibernate[25]. Os desenvolvedores desses frameworks têm despendido muitos esforços trabalhando nesses problemas e tais ferramentas são bem mais sofisticadas do que a maioria das soluções específicas que podem ser construídas à mão. Contudo, mesmo quando um framework de mapeamento objeto-relacional (O/R) é utilizado, é importante estar ciente dos padrões usados. Boas ferramentas de mapeamento O/R dão várias opções de mapeamento para um banco de dados e esses padrões vão ajudá-lo a entender quando selecionar as diferentes opções (FOWLER, 2003).

A seguir são apresentados alguns padrões para o projeto do **Componente de Gerência de Dados.**

6.5.1 Padrões Arquitetônicos para a Camada de Gerência de Dados

A Camada de Persistência (ou Camada de Gerência de Dados ou, ainda, **Componente de Gerência de Dados - CGD**) provê a infraestrutura básica para o armazenamento e a recuperação de objetos no sistema. Sua finalidade é isolar os impactos da tecnologia de gerenciamento de dados sobre a arquitetura do software (COAD; YOURDON, 1993).

A despeito da opção de persistência adotada (SGBD relacional, SGBD orientado a objetos, arquivos), há uma importante questão a ser considerada no projeto do CGD: Que classes devem suportar a persistência dos objetos?

Uma abordagem consiste em isolar completamente a lógica de negócio e o banco de dados, criando uma camada responsável pelo mapeamento entre objetos do domínio e tabelas do banco de dados. Os padrões Mapeador de Dados (*Data Mapper*) (FOWLER, 2003) e Objeto

[25] http://www.hibernate.org/

de Acesso a Dados (*Data Access Object* - DAO) (BAUER; KING, 2007) adotam esta filosofia, de modo que apenas uma parte da arquitetura de software fica ciente da tecnologia de persistência adotada. Essa parte, o Componente de Gerência de Dados (CGD), serve como uma camada intermediária separando objetos do domínio de objetos de gerência de dados. Via conexões de mensagem, o CGD lê e escreve dados, estabelecendo uma comunicação entre a base de dados e os objetos do sistema. Qualquer código SQL[26] está confinado nessas classes, de modo que não código desse tipo em outras classes da arquitetura do software.

a) O Padrão Data Mapper

O padrão Mapeador de Dados (FOWLER, 2003) prescreve uma camada de objetos mapeadores que transferem dados entre objetos em memória e o banco de dados, mantendo-os independentes um do outro e dos mapeadores em si. Os objetos em memória não têm qualquer conhecimento acerca do esquema do banco de dados e não precisam de nenhuma interface para código SQL. De fato, eles não precisam saber sequer que há um banco de dados. O banco de dados, por sua vez, desconhece completamente os objetos que o utilizam.

Em sua versão mais simples, para cada classe a ser persistida em uma tabela, há uma correspondente classe mapeadora. Seja o exemplo da Figura 6.5. Nesse exemplo, a classe mapeadora *PersonMapper* intermedeia a classe *Person* da lógica de negócio e o acesso a seus dados no banco de dados, na correspondente tabela (FOWLER, 2003).

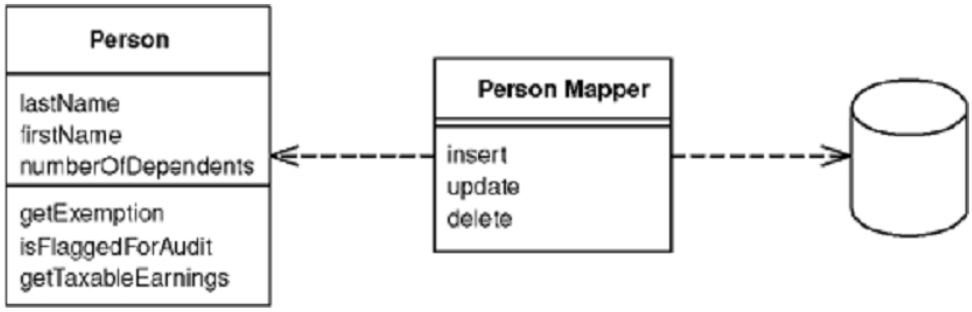

Figura 6.5 – Padrão *Data Mapper* (FOWLER, 2003).

b) O Padrão DAO

O padrão DAO define uma interface de operações de persistência, incluindo métodos para criar, recuperar, alterar, excluir e diversos tipos de consulta, relativa a uma particular entidade persistente, agrupando o código relacionado à persistência daquela entidade (BAUER; KING, 2007). A estrutura básica do padrão, como proposto em (BAUER; KING, 2007), é apresentada na Figura 6.6.

[26] SQL é a abreviatura de *Structured Query Language* (Linguagem Estruturada de Consulta), a linguagem de consulta dos bancos de dados relacionais.

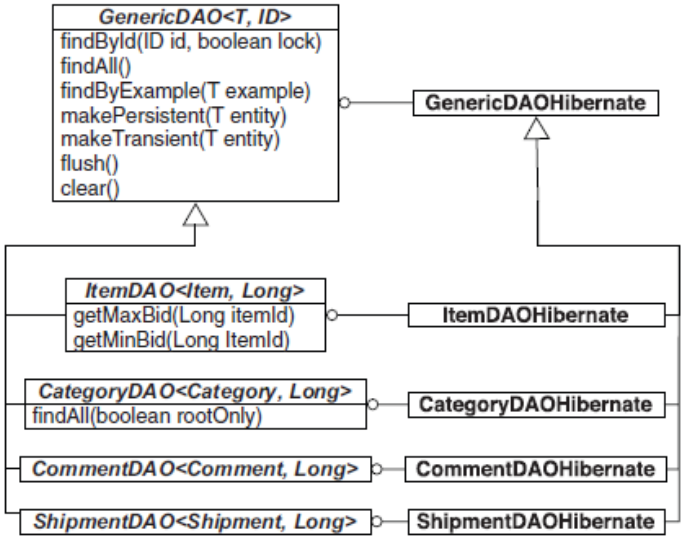

Figura 6.6 – Padrão DAO (BAUER; KING, 2007).

Seguindo esse padrão, a camada de persistência é implementada por duas hierarquias paralelas: interfaces à esquerda e implementações à direita. As operações básicas de armazenamento e recuperação de objetos são agrupadas em uma interface genérica (*GenericDAO*) e uma superclasse genérica (no exemplo da Figura 6.6, *GenericDAOHibernate*). Esta última implementa as operações com uma particular solução de persistência (no caso, Hibernate). A interface genérica é estendida por interfaces para entidades específicas que requerem operações adicionais de acesso a dados. O mesmo ocorre com a hierarquia de classes de implementação. Uma característica marcante desta solução é que é possível ter várias implementações de uma mesma interface DAO (BAUER; KING, 2007).

Referências do Capítulo

BAUER, C., KING, G., *Java Persistence with Hibernate*, Manning, 2007.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M., *Pattern-Oriented Software Architecture: A System of Patterns*, Volume 1, Wiley, 1996.

COAD, P., YOURDON, E., *Projeto Baseado em Objetos*, Editora Campus, 1993.

FOWLER, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

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

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

SOUZA, V.E.S., FrameWeb: um Método baseado em *Frameworks* para o Projeto de Sistemas de Informação *Web*, Dissertação de Mestrado, Programa de Pós-Graduação em Informática, UFES, Universidade Federal do Espírito Santo, 2005.

WAZLAWICK, R.S., *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, Elsevier, 2004.

Capítulo 7 – 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.

7.1 Implementação

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

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

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

7.2 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 7.1 ilustra um cenário típico da atividade de teste.

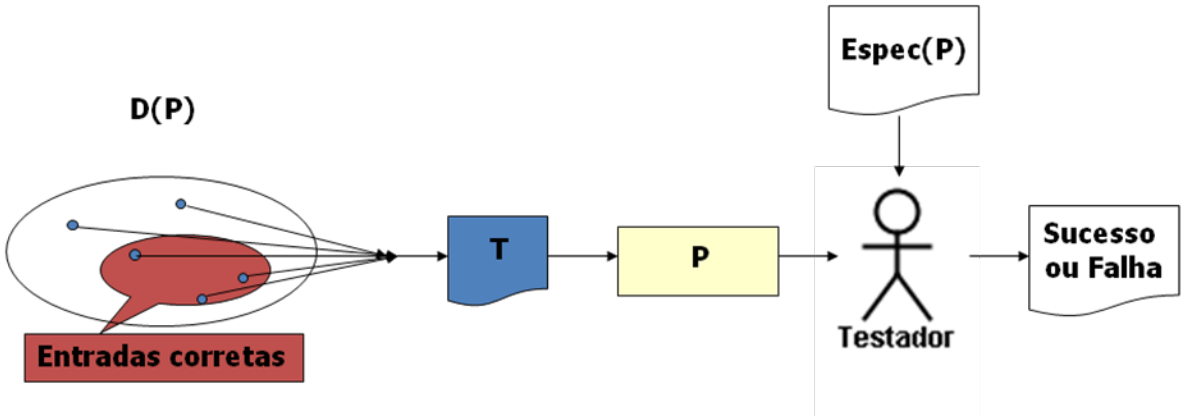

Figura 7.1 – Atividade de Teste.

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.

7.3 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.

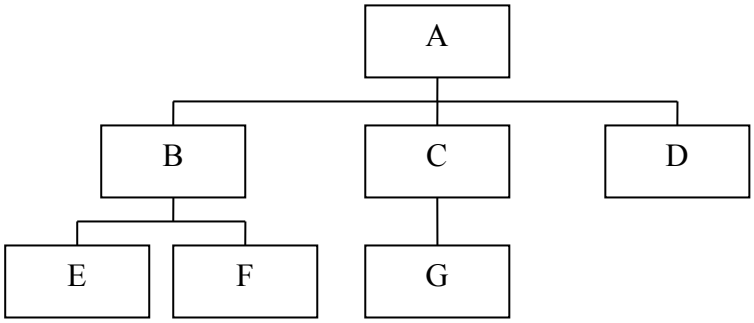

Figura 7.2 – Exemplo de uma hierarquia de módulos.

- **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 7.2, 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).

7.4 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).

7.4.1 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 7.1, 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 7.1 – 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 7.2 apresenta o conjunto de classes de equivalência correspondentes e a Tabela 7.3 mostra casos de teste representativos de cada uma dessas classes.

Tabela 7.2 – 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 7.3 – 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 7.4.

Tabela 7.4 – 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.

7.4.2 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 7.2 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

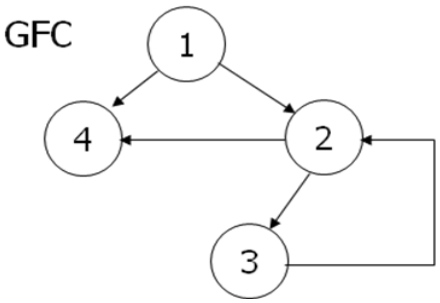

Figura 7.2 – Função que retorna o enésimo termo da Série de Fibonacci.

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: nº_arcos – nº_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.

7.4.3 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.

7.5 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 8 – Entrega e Manutenção

Uma vez que o sistema é aceito e instalado, se está 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.

8.1 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).

8.2 Manutenção

O desenvolvimento de um sistema termina quando o produto é entregue para o cliente e entra em operação. A partir daí, deve-se garantir que o sistema continuará a ser útil e atendendo às necessidades do usuário, o que pode demandar alterações no mesmo. Começa, então, a fase de manutenção 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 9 – 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 9.1 trata da documentação produzida ao longo de processo de desenvolvimento de software; a seção 9.2 retoma o tema Verificação & Validação (V&V), parcialmente abordado no Capítulo 7, o qual trata de Teste de Software, para discutir técnicas complementares de revisão, as quais se aplicam a documentos; a Seção 9.3 discute o processo de Gerência de Configuração de Software; finalmente, a Seção 9.4 trata de Medição aplicada à garantia da qualidade.

9.1 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.

9.2 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.

9.3 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.

9.3.1 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).

9.4 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

ISO/IEC 9126-1, Software Engineering - Product Quality - Part 1: Quality Model, 2001.

ISO/IEC TR 9126-2:2003, Software Engineering – Product Quality – Part 2: External Metrics, 2003a.

ISO/IEC TR 9126-3:2003, Software Engineering – Product Quality – Part 3: Internal Metrics, 2003b.

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 10 – Gerência de Projetos de Software

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.

10.1 Projeto de Software e Gerência de Projetos

Projeto, como definido pelo PMBOK[27], 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).

a) 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 de equipes é importante para uma ampla variedade de situações, tal como uma formação de uma equipe de futebol.

[27] 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).

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.

b) 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.

c) 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.

d) 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 10.1.

Tabela 10.1 – 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*).

10.2 O Processo de Gerência de Projetos de Software

Como ilustra a Figura 10.1, 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 10.1) 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*.

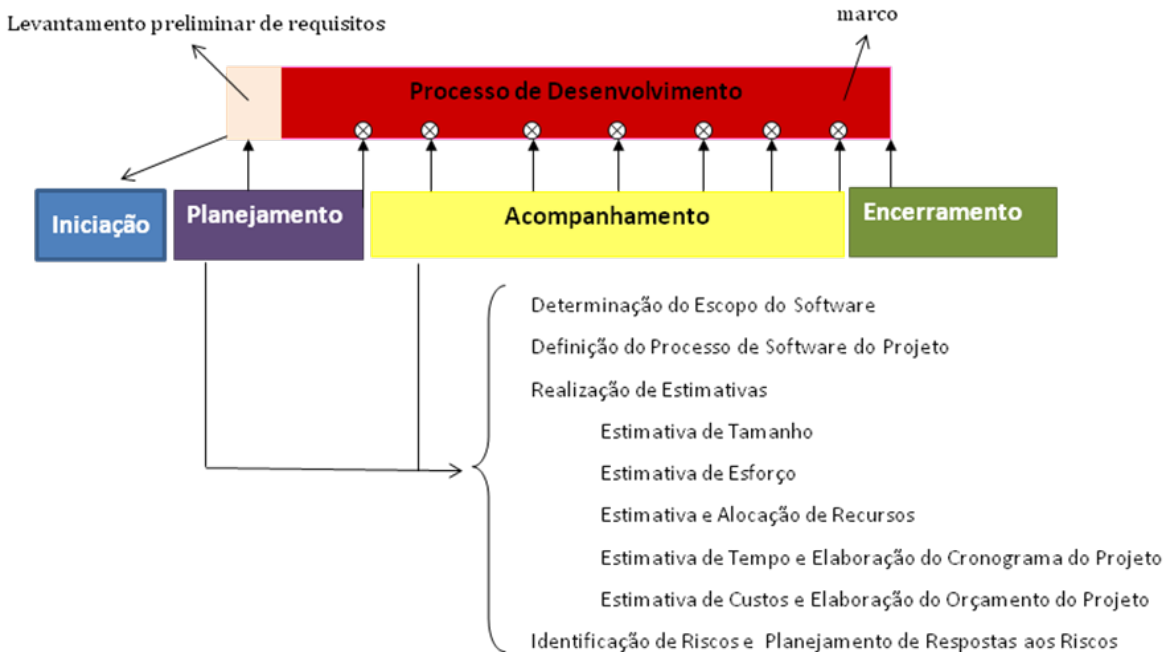

Figura 10.1 – O Processo de Gerência de Projetos de Software.

As atividades relacionadas no quadro na parte inferior na Figura 10.1 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.

10.3 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.

10.4 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 macroatividades 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.

10.5 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 10.1, 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).

10.5.1 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 linhas 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.

10.5.2 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 10.2 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).

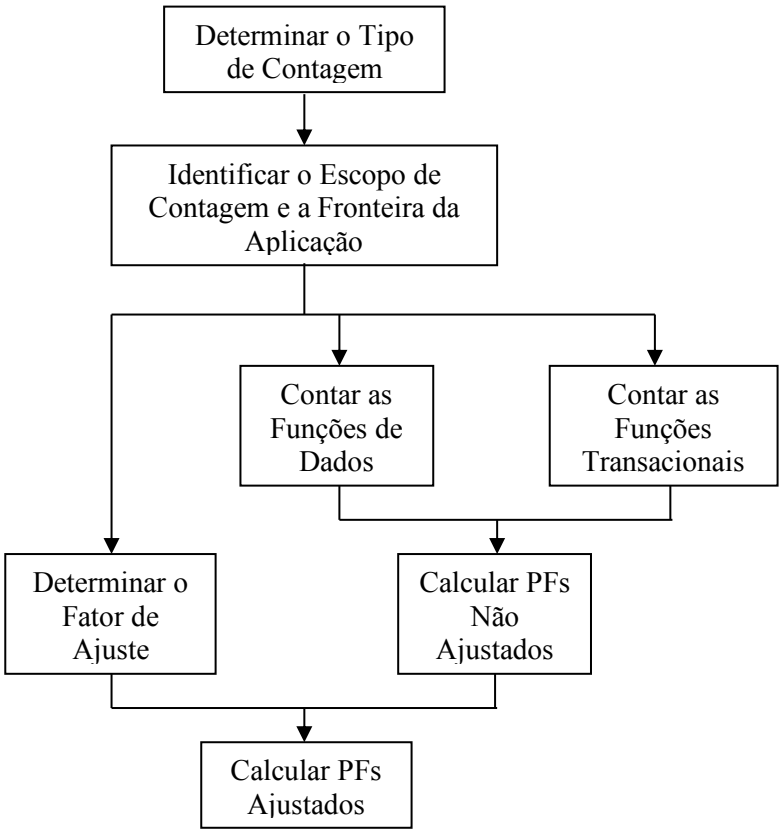

Figura 10.2 – Passos do Método de Análise de Pontos de Função.

- **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 10.3 apresenta uma visão esquemática dos tipos de função que são considerados na contagem da APF.

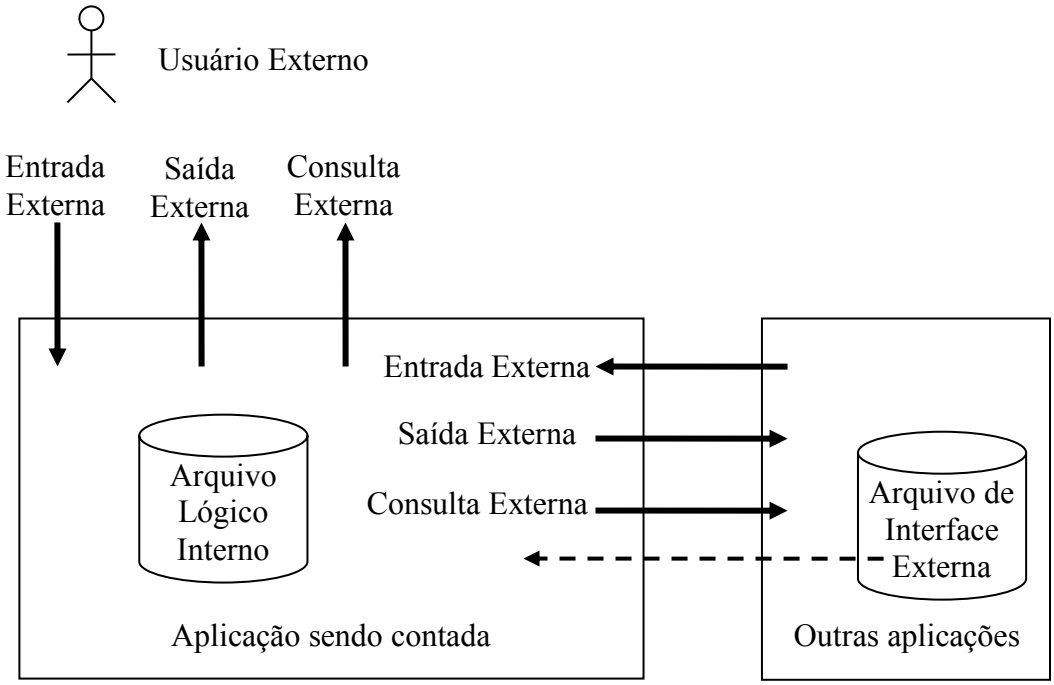

Figura 10.3 – Visão Esquemática das Funções de uma Aplicação segundo a APF.

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.

10.5.3 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 10.1 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).

10.5.4 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.

10.5.5 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 10.2 (PRESSMAN, 2011).

Tabela 10.2 – 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.

10.5.6 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.

10.6 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 10.3 e 10.4 para as escalas acima.

Tabela 10.3 - Categorias de Probabilidade (JALOTE, 1999)

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

Tabela 10.4 - 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.

10.7 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 11 – 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.

11.1 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, 2015), ISO/IEC 12207 (ISO/IEC, 2008), ISO/IEC 15504 (ISO/IEC, 2004) e os modelos CMMI (SEI, 2010) e MPS.BR (SOFTEX, 2016).

11.1.1 Normas ISO

As normas da família ISO 9000 (ISO, 2015) 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 11.1 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.

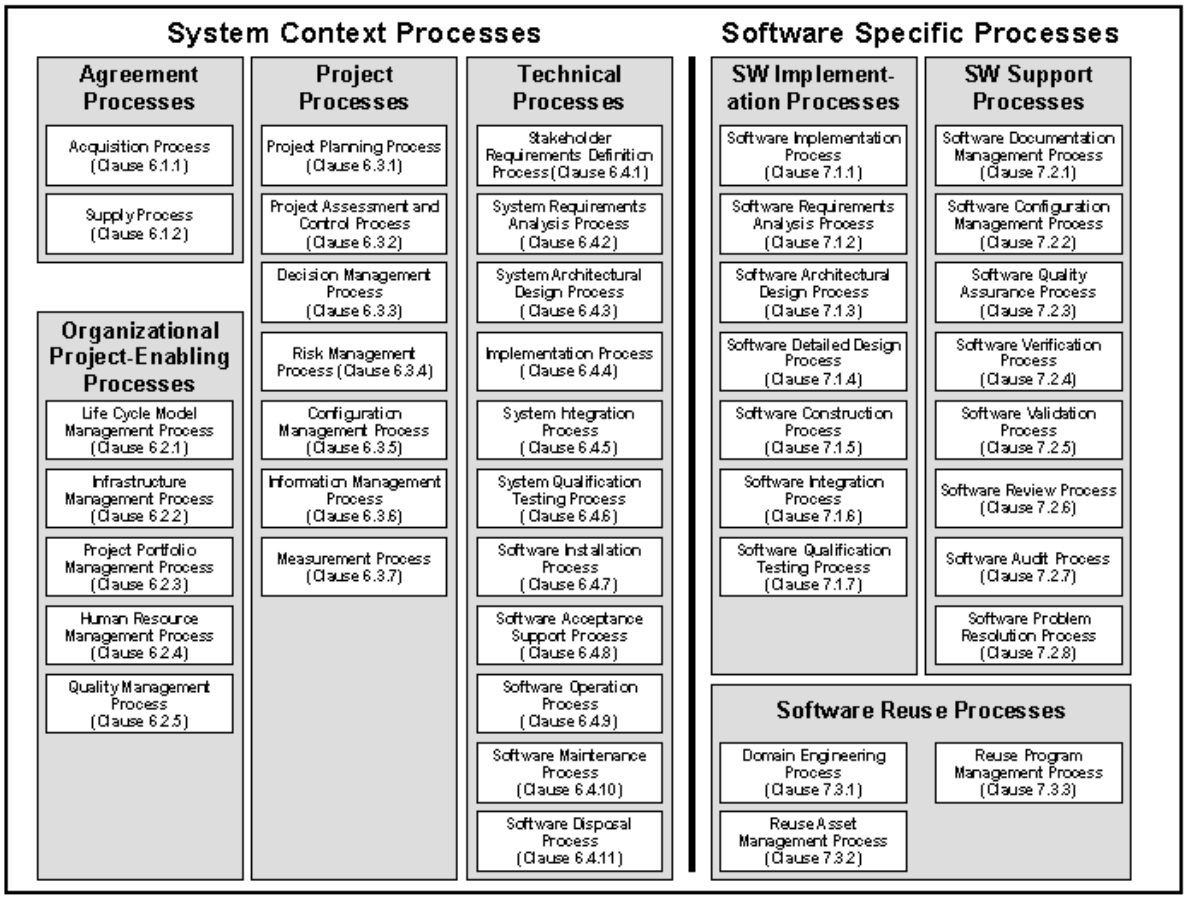

Figura 11.1 – Processos de Ciclo de Vida da ISO/IEC 12207.

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.

11.1.2 O Modelo CMMI

O Modelo de Maturidade e Capacidade Integrado (*Capability Maturity Model Integration - CMMI*) (SEI, 2010a) 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 11.2 mostra os níveis de maturidade do CMMI e, em seguida, na Tabela 11.1 são apresentadas as áreas de processo de cada nível.

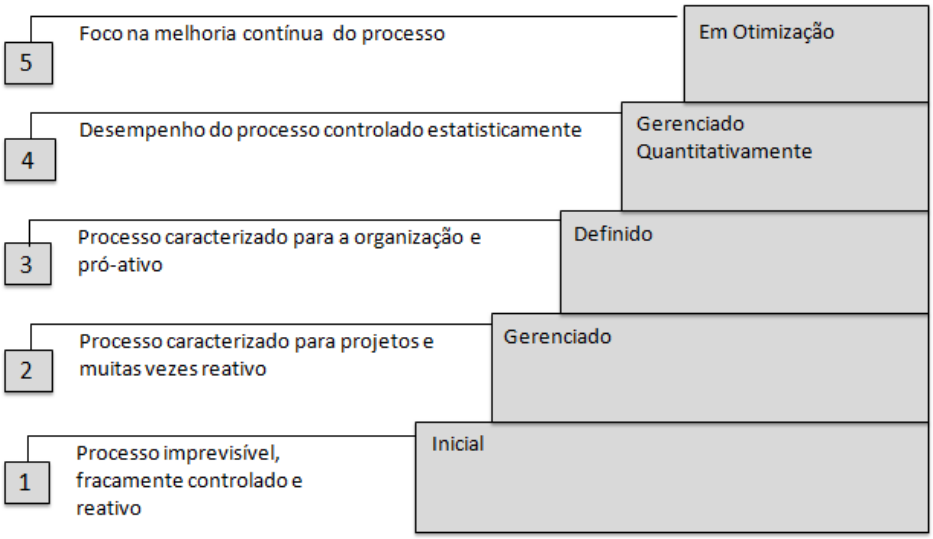

Figura 11.2 - Os níveis de maturidade do CMMI.

Tabela 11.1 – 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

11.1.3 O Modelo de Referência Brasileiro – MPS.BR

O MPS.BR – Melhoria de Processo do Software Brasileiro (SOFTEX, 2016) 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, ISO/IEC 15504 e ISO/IEC 20000 (ISO/IEC, 2011), estando totalmente aderente a essas normas. Além disso, o MPS.BR também cobre o conteúdo do CMMI-DEV (SEI, 2010a) e CMMI-SVC (SEI, 2010b).

O MPS.BR está dividido em três componentes:

- Modelo de Referência MPS para Software (MR-MPS-SW): contém os requisitos que as organizações deverão atender para estar em conformidade com o MPS.BR para processos de software. Define, também, os níveis de maturidade e de capacidade de processos e os processos em si.
- Modelo de Referência MPS para Serviços (MR-MPS-SV): contém os requisitos que as organizações deverão atender para estar em conformidade com o MPS.BR para processos relacionados a serviços. 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 11.2 mostra os níveis de maturidade do MPS.BR e seus processos. A Tabela 11.3 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 11.2 – 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 11.3 – 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

11.2 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 11.3 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.

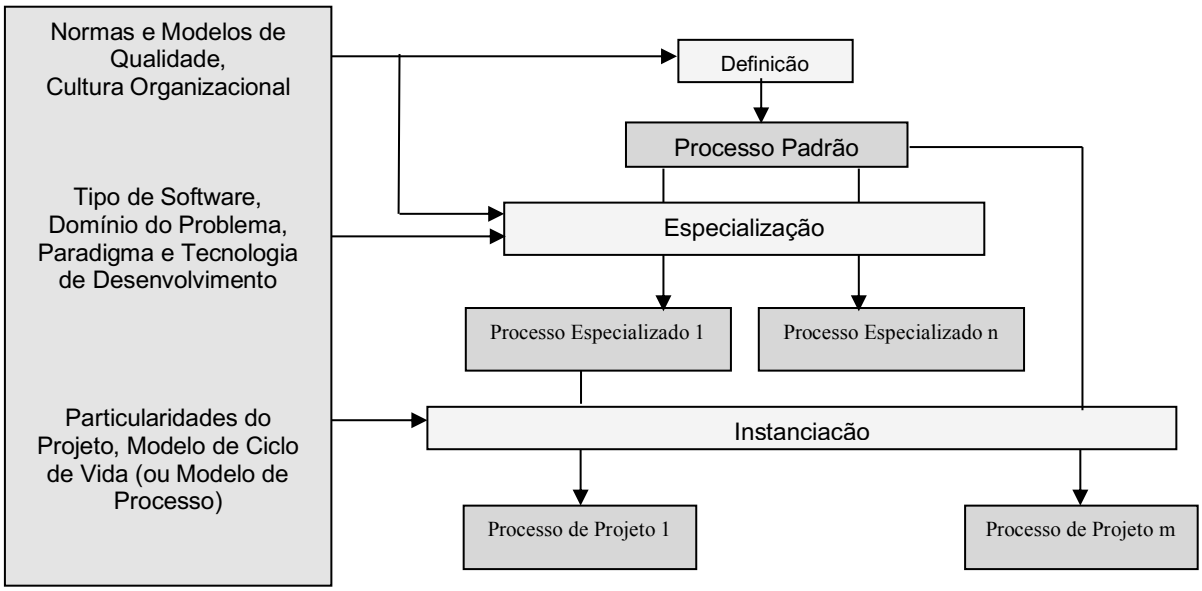

Figura 11.3 – Modelo para Definição de Processos em Níveis.

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.

11.3 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 autogerencia 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.

11.3.1 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.

11.3.2 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*.

11.4 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*, 2015.

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.

ISO/IEC, *ISO/IEC 20000 Information Technology– Service Management*, 2011.

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, 2010a.

SEI, *CMMI for Services, Version 1.3*, Technical Report ESC-TR-2010-034, 2010b.

SOFTEX, *MPS.BR – Melhoria de Processo do Software Brasileiro – Guia Geral*, 2016.

Anexo A - Análise de Pontos de Função

A.1- O Processo de Contagem de Pontos de Função

O processo de contagem dos pontos de função pode ser dividido em sete etapas: *(i)* determinar tipo de contagem; *(ii)* identificar a fronteira da aplicação; *(iii)* contar as funções tipo dados; *(iv)* contar as funções tipo transação; *(v*) calcular pontos de função não ajustados (com base nos resultados obtidos em *(iii)* e *(iv)*); *(vi)* calcular o valor do fator de ajuste; e *(vii)* calcular os pontos de função ajustados (com base nos resultados obtidos em *(v)* e *(vi)*) , como mostra a figura A.1. As etapas são descritas a seguir.

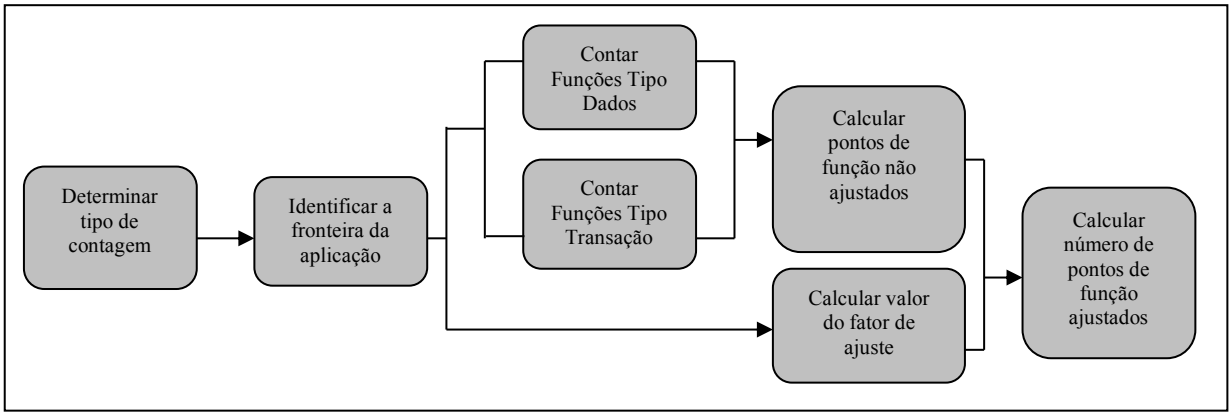

Figura A1.1 – Visão Geral do Processo de Análise de Pontos de Função (GARMUS e HERRON, 2001).

(i) *Determinar o Tipo de Contagem*

Para realizar a contagem dos pontos de função de um projeto, inicialmente, é preciso determinar o tipo de contagem a ser realizada, podendo esta ser:

- *Projeto de Desenvolvimento:* mede a funcionalidade fornecida aos usuários finais do *software* para a primeira instalação da aplicação. Inclui asfuncionalidades da contagem inicial da aplicação e as funcionalidades requeridas para conversão de dados.
- *Projeto de Manutenção:* mede as modificações realizadas para aplicações existentes. Inclui as funcionalidades fornecidas aos usuários através de adição, modificação ou exclusão de funções na aplicação. As funcionalidades de conversão de dados também devem ser consideradas, caso existam. Após a manutenção, a contagem da aplicação deve ser refeita para refletir as alterações realizadas.
- *Aplicação:* mede uma aplicação instalada. É também referenciada como contagem de linha de base ou contagem instalada e avalia as funcionalidades correntes providas aos usuários finais da aplicação.

(ii) I*dentificar a Fronteira da Aplicação*

Após determinado o tipo de contagem, a fronteira da aplicação deve ser identificada. Ela indica a separação entre o projeto que está sendo medido e as aplicações externas ao domínio do usuário. É através dela que torna-se possível definir quais funcionalidades serão incluídas no processo de contagem dos pontos de função.

(iii) *Contar Funções Tipo Dados*

Nesta etapa as funcionalidades da aplicação começam a ser identificadas e contadas.

A funcionalidade da aplicação é avaliada em termos do *quê* é fornecido pela mesma, não do *como* é fornecido. Apenas componentes definidos e solicitados pelo usuário devem ser contados (GARMUS e HERRON, 2001).

As Funções Tipo Dados representam as funcionalidades fornecidas pelo sistema ao usuário, para atender às necessidades referentes aos dados que o sistema irá manipular. Essas funções podem ser:

1) *Arquivo Lógico Interno (ALI) :* grupo logicamente relacionado de dados ou informações de controle, identificável pelo usuário, mantido dentro da fronteira da aplicação que está sendo controlada. Por exemplo: as tabelas ou classes do sistema.
2) *Arquivo de Interface Externa (AIE):* grupo logicamente relacionado de dados ou informações de controle, referenciado pela aplicação, identificável pelo usuário, mantido fora da fronteira da aplicação que está sendo controlada. Por exemplo: as tabelas acessadas em um outro sistema.

A diferença básica entre um ALI e um AIE é que o último **não** é mantido pela aplicação que está sendo contada. Um AIE contado para uma aplicação sempre será contado como um ALI em sua aplicação de origem.

Nas definições de ALI e AIE foram utilizados alguns termos e expressões que merecem esclarecimento. São elas:

- *Informações de Controle:* são dados utilizados pela aplicação para garantir aderência com os requisitos funcionais especificados pelo usuário. Por exemplo: datas e horas são utilizadas pelos usuários para estabelecer a sequência ou o momento de eventos. Assim, datas e horas são informações de controle.
- *Identificável pelo Usuário:* refere-se aos requisitos específicos que um usuário ou grupo de usuários seria capaz de definir para a aplicação.
- *Mantido:* refere-se ao fato de que o dado pode ser modificado através de um processo elementar da aplicação. Um processo elementar é a menor atividade capaz de produzir resultados significativos para o usuário. Por exemplo: incluir, alterar e excluir.

Cada Arquivo Lógico Interno e cada Arquivo de Interface Externa possui dois tipos de elementos que devem ser contados para cada função identificada:

- *Tipos de Elementos de Dados (TED):* campo único, reconhecido pelo usuário, não recursivo. Por exemplo: campos das tabelas.
- *Tipos de Elementos de Registros (TER):* subgrupo de dados, reconhecido pelo usuário. Por exemplo: generalização/especialização de classes.

Ao final dessa etapa devem estar identificados quantos Arquivos Lógicos Internos e Arquivos de Interface Externa o sistema possui e para eles, quantos são os Tipos de Elementos de Dados e os Tipos de Registros encontrados.

(iv) *Contar Funções Tipo Transação*

As Funções Tipo Transação representam as funcionalidades de processamento dos dados fornecidas pelo sistema ao usuário. Essas funções podem ser:

1) *Entrada Externa (EE):* processo elementar da aplicação que processa dados ou informações de controle que vêm de fora da fronteira da aplicação que está sendo controlada. Exemplos: validações, fórmulas e cálculos matemáticos cujos parâmetros vêm de fora da fronteira da aplicação.
2) *Saída Externa (SE):* processo elementar da aplicação que gera dados ou informações de controle que são enviados para fora da fronteira da aplicação que está sendo controlada. Exemplos: relatórios e gráficos.
3) *Consulta Externa (CE):* processo elementar da aplicação que representa uma combinação de entrada (solicitação de informação) e saída (recuperação de informação). Exemplos: consultas implícitas, verificação de senhas e recuperação de dados com base em parâmetros.

Cada Entrada Externa, Saída Externa e Consulta Externa possui dois tipos de elementos que devem ser contados para cada função identificada:

- *Tipos de Elementos de Dados (TED):* campo único, reconhecido pelo usuário, não recursivo. Por exemplo: campos das tabelas.
- *Tipos de Arquivos Referenciados ou Arquivos Referenciados (TAR):* arquivos lógicos utilizados para processar a entrada e/ou saída. É o total de ALI e AIE utilizados pela transação.

Ao final dessa etapa devem estar identificadas quantas Entradas Externas, Saídas Externas e Consultas Externas o sistema possui e, para elas, quantos são os Tipos de Elementos de Dados e os Arquivos Referenciados encontrados.

(v) *Calcular os Pontos de Função Não Ajustados*

Após serem contadas todas as Funções Tipo Dados e as Funções Tipo Transação e seus elementos, é preciso calcular os pontos de função não ajustados, que refletem especificamente as funcionalidades fornecidas ao usuário pelo produto. Para isso, é preciso identificar a complexidade e a contribuição, em pontos por função, de cada uma das funções e elementos contados.

Para determinar a complexidade e contribuição das funções e seus elementos, é necessário utilizar as relações dos valores de complexidade e contribuição fornecidas pela técnica. A seguir são apresentadas tabelas que indicam a complexidade e contribuição das funções e seus elementos em um sistema, de acordo com a contagem estabelecida nas etapas *(iii)* e *(iv)*.

A Tabela A.1 indica a complexidade de um Arquivo Lógico Interno ou Arquivo de Interface Externa de acordo com o número de Tipos de Elementos de Dados e de Tipos de Elementos de Registros identificados para ele.

Tabela A.1 – Complexidade de Arquivos Lógicos Internos e Arquivos de Interface Externa.

Tipos de Elementos de Registros	*Tipos de Elementos de Dados*		
	1 a 19	***20 a 50***	***≥ 51***
1	*BAIXA*	*BAIXA*	*MÉDIA*
2 a 5	*BAIXA*	*MÉDIA*	*ALTA*
≥ 6	*MÉDIA*	*ALTA*	*ALTA*

A Tabela A.2 indica a complexidade de uma Entrada Externa de acordo com o número de Tipos de Elementos de Dados e de Arquivos Referenciados identificados para ela. Também é utilizada para determinar a complexidade das entradas de uma Consulta Externa.

Tabela A.2 – Complexidade de Entradas Externas e Entradas das Consultas Externas.

Tipos de Arquivos Referenciados	*Tipos de Elementos de Dados*		
	1 a 4	***5 a 15***	***≥ 16***
0 a1	*BAIXA*	*BAIXA*	*MÉDIA*
2	*BAIXA*	*MÉDIA*	*ALTA*
≥ 3	*MÉDIA*	*ALTA*	*ALTA*

A Tabela A.3 indica a complexidade de uma Saída Externa de acordo com o número de Tipos de Elementos de Dados e de Arquivos Referenciados identificados para ela. Também é utilizada para determinar a complexidade das saídas de uma Consulta Externa.

Tabela A.3 - Complexidade de Saídas Externas e Saídas das Consultas Externas.

Tipos de Arquivos Referenciados	*Tipos de Elementos de Dados*		
	1 a 5	***6 a 19***	***≥ 20***
0 a1	*BAIXA*	*BAIXA*	*MÉDIA*
2 a 3	*BAIXA*	*MÉDIA*	*ALTA*
≥ 4	*MÉDIA*	*ALTA*	*ALTA*

A Tabela A.4 indica as contribuições (pesos) obtidas através das complexidades calculadas para as funções identificadas.

Tabela A.4 - Contribuições (pesos) das complexidades.

		Contribuições (pesos)				
Complexidades		***ALI***	***AIE***	***EE***	***SE***	***CE***
	BAIXA	7	5	3	4	3
	MÉDIA	10	7	4	5	4
	ALTA	15	10	6	7	6

Para calcular os pontos de função não ajustados, multiplica-se o número de funções identificadas para uma determinada complexidade por sua contribuição. Ao final, soma-se todos os pontos de função encontrados.

Na Tabela A.5 é apresentado um exemplo para o cálculo dos pontos de função não ajustados (PFNA) gerados pelos ALI de um sistema hipotético. O mesmo deve ser feito para a outras funções do sistema (AIE, EE, SE e CE).

Tabela A.5 – Exemplo de cálculo dos pontos de função não ajustados.

Função	***Itens Contados por Complexidade***	***Contribuição***	***Total por Complexidade***	***Total de PFNA da Função***
ALI	1 Baixa 2 Média 1 Alta	x 7 x 10 x 15	7 20 15	42

(vi) *Calcular Valor do Fator de Ajuste*

O número de pontos de função não ajustados de um sistema reflete a funcionalidade que o sistema fornecerá ao usuário, sem considerar as especificidades do sistema. Por exemplo, um mesmo sistema pode ser implementado para operar *stand alone* para um cliente e em arquitetura cliente servidor para outro. As funcionalidades seriam as mesmas, o que resultaria na mesma contagem de pontos de função não ajustados, mas quando considera-se as características do sistema para cada cliente, observa-se que os pontos de função devem ser ajustados para refletir a maior complexidade do sistema na arquitetura cliente servidor.

Para ajustar os pontos de função encontrados na etapa *(v)* devem ser levadas em consideração 14 (quatorze) características do sistema que serão analisadas e fornecerão o valor do fator de ajuste. São elas: Comunicação de Dados, Processamento Distribuído, Performance, Configuração Altamente Utilizada, Taxa de Transações, Entrada de Dados On-Line, Eficiência do Usuário Final, Atualização *On-Line*, Processamento Complexo, Reutilização, Facilidade de Operação, Facilidade de Instalação, Múltiplos Locais e Modificações Facilitadas.

Para cada característica deve ser atribuído um nível de influência de 0 (zero) a 5 (cinco), onde 0 (zero) indica nenhuma influência, 1 (um) influência mínima, 2 (dois) influência moderada, 3 (três) influência média, 4 (quatro) influência significativa e 5 (cinco) grande influência.

Para calcular o valor do fator de ajuste deve-se seguir a relação

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

Onde

- VFA é o valor do fator de ajuste
- GIT é o grau de influência total (soma de todos os valores dos níveis de influência).

(vii) *Calcular Pontos de Função Ajustados*

Após calculado o valor do fator de ajuste, os pontos de função não ajustados serão ajustados, multiplicando-se o valor dos pontos de função não ajustados (PFNA), obtidos em *(v),* pelo valor do fator de ajuste (VFA),obtido em *(vi)*.

Assim,

$$PFA = PFNA \times VFA$$

O número de pontos de função encontrado representa o tamanho da aplicação de acordo com sua funcionalidade.

Para calcular as estimativas de esforço, prazo e custos para a aplicação é necessário conhecer valores como o custo de um ponto de função (por exemplo R$200,00) e o tempo necessário para realizar um ponto de função (por exemplo 2,5 h), ou o esforço para realizar um ponto de função (por exemplo 14 pessoas/mês) e o custo do esforço. Com esses valores é possível calcular as estimativas para o projeto através das relações entre o número total de pontos de função do sistema e os valores de um ponto de função.

Para determinar os valores de um ponto de função, a organização pode realizar medições em projetos anteriores e obter um valor médio para o ponto de função. Caso não existam projetos anteriores podem ser consultadas tabelas disponibilizadas pelo IFPUG *(Institute Function Point Users Group)* e por seus órgãos representantes em cada país.

A.2 - Um Exemplo de Uso da Análise de Pontos de Função[28]

Para exemplificar a utilização da técnica Análise de Pontos de Função, consideremos um pequeno sistema hipotético desenvolvido para uma academia de ginástica, com o objetivo de cadastrar os alunos matriculadose emitir um relatório gerencial que apresente o número de alunos matriculados totalizados por mês.

Considere o diagrama da Figura A.2 como representação do sistema hipotético. O arquivo (tabela) Alunos possui 10 atributos.

[28] O exemplo aqui apresentado foi extraído de WEBER *et. al* (2001*)*

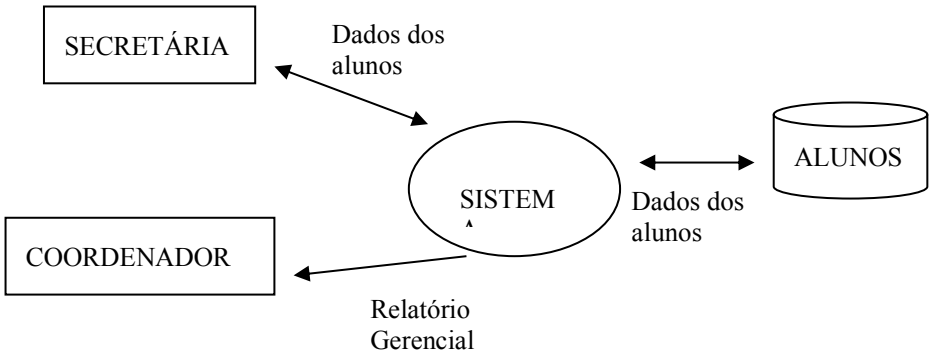

Figura A.2 – Diagrama de Contexto do sistema hipotético.

Passos:

(i) *Determinar o Tipo de Contagem*

O tipo de contagem é para um *Projeto de Desenvolvimento*, uma vez que se trata de um sistema a ser desenvolvido e não de manutenção ou de medição em aplicação instalada.

(ii) I*dentificar a Fronteira da Aplicação*

Não há interação com outros sistemas.

(iii) *Contar Funções Tipo Dados*

O número de Arquivos Lógicos Internos é 1 pois só há manipulação do arquivo Alunos. O número de Elementos de Dados é 10, que são os atributos do arquivo Alunos. Só há um Tipo de Registro em Alunos, pois não há especialização deste arquivo.

Não há Arquivos de Interface Externa, uma vez que não há interação com outros sistemas.

Tabela A.6 – Exemplo de Contagem de Funções Tipo Dados.

Arquivos Lógicos Internos	**Tipos de Elementos de Dados**	**Tipos de Elementos de Registros**
1 (Alunos)	10 (atributos de Alunos)	1
Arquivos de Interface Externa	**Tipos de Elementos de Dados**	**Tipos de Elementos de Registros**
0 (não há interação com outros sistemas)	0	0

(iv) *Contar Funções Tipo Transação*

Existem três Entradas Externas: inclusão, alteração e exclusão de alunos. Para as duas primeiras existem 10 Elementos de Dados, que são os atributos que são fornecidos como entrada para inclusão ou os atributos que podem ser modificados em uma alteração. Para a exclusão, apenas um Elemento de Dados é considerado, que é o código do aluno que será excluído. Em todas as entradas há apenas um arquivo referenciado: Alunos.

Supondo que o sistema possui uma consulta aos dados cadastrais, a função Consultas Externas apresenta contagem 1. Todos os atributos do arquivo Alunos são exibidos, totalizando 10 Elementos de Dados. Apenas o arquivo Alunos é utilizado, então Arquivos Referenciados é igual a 1.

A única Saída Externa é o Relatório Gerencial. Supondo que ele apresente: o código do aluno, nome do aluno, mês da matrícula, totalizador de alunos matriculados por mês e totalizador de alunos matriculados no ano, temos 5 Elementos de Dados. Apenas o arquivo Alunos é utilizado, então Arquivos Referenciados é igual a 1.

Tabela A.7 – Exemplo de Contagem de Funções Tipo Transação.

Entradas Externas	**Tipos de Elementos de Dados**	**Arquivos Referenciados**
1 (Inclusão)	10 (atributos de Alunos)	1 (Alunos)
1 (Alteração)	10 (atributos de Alunos)	1 (Alunos)
1 (Exclusão)	1 (código do aluno)	1 (Alunos)
Consultas Externas	**Tipos de Elementos de Dados**	**Arquivos Referenciados**
1 (Consulta aos dados cadastrais)	10 (atributos de Alunos)	1 (Alunos)
Saídas Externas	**Tipos de Elementos de Dados**	**Arquivos Referenciados**
1 (Relatório Gerencial)	5 (informações apresentadas no relatório)	1 (Alunos)

(v) *Calcular os Pontos de Função Não Ajustados*

Analisando os valores obtidos no passos acima e as tabelas A.1 a A.4 chegamos aos seguintes valores:

Tabela A.8 – Exemplo de cálculo dos pontos de função não ajustados.

Função	***Itens Contados por Complexidade***	***Contribuição***	***Total por Complexidade***	***Total de PFNA da Função***
ALI	1 Baixa 0 Média 0 Alta	x 7 x 10 x 15	7 0 0	7
AIE	0 Baixa 0 Média 0 Alta	x 5 x 7 x 10	0 0 0	0
EE	3 Baixa 0 Média 0 Alta	x 3 x 4 x 6	9 0 0	9
CE	1 Baixa 0 Média 0 Alta	x 3 x 4 x 6	3 0 0	3
SE	1 Baixa 0 Média 0 Alta	x 4 x 5 x 7	4 0 0	4

Total de Pontos de Função Não Ajustados: 23

(v) *Calcular Valor do Fator de Ajuste*

Para calcular o fator de ajuste, as 14 características foram consideradas, obtendo-se os seguintes valores:

Tabela 9 – Exemplo de cálculo do fator de ajuste.

Características Gerais do Sistema	***Nível de Influência***	***Justificativa***
Comunicação de Dados	0	O sistema opera em micro *stand-alone,* portanto, não possui comunicação de dados.
Processamento Distribuído	0	O sistema opera em micro *stand-alone.*
Performance	1	Requisitos de *performance* foram estabelecidos, mas nenhuma ação especial foi necessária.
Configuração altamente utilizada	0	Não há restrições operacionais.
Volume de Transações	0	Nenhum período de pico de transações esperado.
Entrada de dados *on-line*	5	Sistema *on-line.*
Eficiência do usuário final	3	Sistema desenvolvido com interface gráfica.
Atualização *on-line*	3	Sistema *on-line*, sem proteção para perda de dados.
Processamento complexo	0	O sistema não executa processamento matemático ou de segurança.
Reusabilidade	1	O sistema foi desenvolvido levando-se em conta reuso de rotinas.
Facilidade de instalação	4	Utilização de ferramenta automática para implantação do sistema.
Facilidade de Operação	2	Sistema *on-line.*
Múltiplos locais	0	Nenhuma solicitação do usuário para implantar a aplicação em mais de um local..
Modificação facilitada	0	Nenhuma solicitação do usuário para projetar a aplicação visando minimizar ou facilitar mudanças.
Grau de Influência Total = 19		
VFA = (GIT * 0,01) + 0,65 = 0,84		

(vi) *Calcular Pontos de Função Ajustados*

Para ajustar os pontos de função do sistema, basta multiplicar os pontos de função não ajustados pelo valor do fator de ajuste, como apresentado abaixo:

PFA = 23 * 0,84 = 19,32

Sendo assim, o sistema hipotético possui 19 pontos de função.

A.3 - As 14 Características Gerais e seus Graus de Influência (Dias, 2004)

A seguir são apresentadas as 14 características gerais a serem analisadas para a contagem de pontos de função e as descrições necessárias para a identificação do seu grau de influência. Os graus de influência possíveis são apresentados na Tabela A.10.

Tabela A.10 – Graus de influência.

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[29] 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;

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;

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

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 bilingüe (contar como 4 itens)
- Suporte multilíngüe. (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 a 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 freqüência.
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 influencia 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.

Referências do Anexo

DIAS, R., 2004, "Análise por Pontos de Função: Uma Técnica para Dimensionamento de Sistemas de Informação", on-line. Disponível em: www.presidentekennedy.br/resi/edicao03/artigo02.pdf.

WEBER, C. K., ROCHA, A. R. C., NASCIMENTO, C. J., 2001, "Qualidade e Produtividade em Software", 4ª Edição, Ed. Makron Books.