Marcos André Knewitz

Introdução ao desenvolvimento de software com UML

COLEÇÃO EAD
Editora Unisinos

INTRODUÇÃO AO DESENVOLVIMENTO DE SOFTWARE COM UML

Marcos André Knewitz

Editora Unisinos
2011

APRESENTAÇÃO

Para desenvolver um produto de software de qualidade é necessário empreender um projeto em acordo com os preceitos da Engenharia de Software. A aplicação de engenharia em um projeto implica a adoção de um processo de desenvolvimento de software. Processos compreendem a execução ordenada de atividades de forma que um objetivo seja atingido. Este livro tem como foco as principais atividades de um processo de software, excetuando-se as atividades relacionadas à implementação.

O Capítulo 1 faz uma introdução ao desenvolvimento de software, apresentando os conceitos de software, produto de software e Engenharia de Software. O Capítulo 2 aprofunda-se no processo de software, um dos componentes essenciais da Engenharia de Software. O Capítulo 3 aborda a UML – *Unified Modeling Language* − linguagem padrão para modelagem de sistemas utilizada em muitas atividades do processo de software.

Os capítulos seguintes detalham as atividades de cinco importantes disciplinas do processo: modelagem de negócio, requisitos, análise de sistemas, projeto de software e testes. No Capítulo 4 são abordados os artefatos e técnicas para modelagem de negócio, dando ênfase ao diagrama de atividades da UML. O Capítulo 5 aborda conceitos sobre requisitos e engenharia de requisitos, apresentando um método de identificação e especificação de requisitos baseado na modelagem de casos de uso, bem como o respectivo diagrama, o diagrama de casos de uso. No Capítulo 6 é apresentada a análise orientada a objetos, dando-se ênfase na modelagem conceitual e demonstrando-se a utilização do diagrama de classes da UML na construção de um modelo de domínio. O Capítulo 7 é voltado à modelagem funcional. Neste, é demonstrada a utilização do diagrama de sequência para a identificação e especificação de operações de sistema. É exposto, também, o diagrama de máquina de estados como ferramenta para investigar o ciclo de vida dos objetos do sistema. O Capítulo 8 apresenta os principais conceitos relacionados a projeto de software e, assumindo-se uma arquitetura de software em camadas, é demonstrado como projetar cada uma delas usando-se diversos diagramas da UML. É mostrado também como mapear um software orientado a objetos para persistir em um banco de dados relacional. Por fim, no Capítulo 9, a disciplina de testes é abordada. Tipos de teste, estágios de teste e processos de testes são tratados.

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO AO DESENVOLVIMENTO DE SOFTWARE

Este capítulo faz uma introdução ao desenvolvimento de software. Software e produto de software são conceituados e caracterizados. As dificuldades para se desenvolver produtos de software são explanadas e a Engenharia de Software é apresentada como alternativa para viabilizar o desenvolvimento de software com qualidade.

A utilização do termo desenvolvimento indica que um software não nasce pronto. Ele precisa ser concebido, projetado e construído, passo a passo, seguindo-se um processo específico. Processo, no sentido geral, refere-se a um conjunto de atividades organizadas que levam à realização ou à produção de algo. Para a produção de software utilizam-se processos específicos denominados processo de desenvolvimento de software ou, simplesmente, processo de software.

As atividades de um processo podem ser organizadas em disciplinas. Atividades e disciplinas possuem objetivos próprios e, para atingi-los, é importante a utilização de um método. A aplicação de um método experimentado e testado serve como guia na execução de uma atividade ou disciplina e contribui para que os objetivos das mesmas sejam atingidos.

Em função da variedade das aplicações de software e da diversidade de tecnologias e paradigmas de desenvolvimento, muitos métodos estão disponíveis. Por esta razão, este livro não foca em nenhuma metodologia específica, mas sim na apresentação de um grupo comum de disciplinas, atividades e métodos aplicáveis em grande parte dos projetos de desenvolvimento de software. É seguido um processo genérico contemplando as principais atividades técnicas do desenvolvimento de software. Diversos métodos específicos para disciplinas e atividades do processo são apresentados, sendo que os métodos relacionados à análise tomam como base o paradigma da orientação a objetos.

1.1 Software e Produto de Software

Um sistema computacional, ou sistema baseado em computador, possui diversos

componentes: hardware, dados, meios de comunicação e software, que interagem entre si para cumprir determinados objetivos. Software é a parte programável de um sistema computacional.

Produto de software refere-se a software que é produzido para ser entregue a clientes e usuários. O Institute of Electrical and Electronics Engineers (IEEE, 1997) define produto de software como o conjunto completo, ou qualquer item individual do conjunto de programas de computador, procedimentos, dados e documentação associada a serem entregues a um cliente ou usuário final.

1.1.1 Características do Software

O produto de software apresenta características que o diferencia de outros tipos de produtos. Essas características, intrínsecas à natureza do software, trazem dificuldades para o desenvolvimento de software, conduzindo frequentemente a produtos de baixa qualidade e a projetos que ultrapassam o prazo e o custo planejados. A questão torna-se mais crítica quando se considera o volume crescente de projetos de software e a sua importância para as organizações que os demandam.

Uma das principais características do software é que seu desenvolvimento pertence a um domínio de alta modificação e instabilidade conhecido como domínio de desenvolvimento de produto novo. Nesses ambientes não é viável definir os requisitos em detalhe no início do processo de desenvolvimento.

Comparando-se a construção de software com construção civil tem-se que na engenharia civil nenhum tijolo é assentado sem que antes um conjunto de plantas, maquetes e imagens da obra sejam produzidas e aprovadas pelo cliente. Uma vez aprovadas, inicia-se a construção, e mudanças representativas não mais serão possíveis. Já na produção de software, os requisitos do produto continuarão a se modificar ao longo do projeto. Tentativas de capturar e congelar esses requisitos para então iniciar a produção do software serão frustradas e potencialmente conduzirão ao fracasso do projeto.

Esta situação é agravada pelo fato de que em uma obra civil um cliente é capaz de compreender as plantas baixas, ao mesmo tempo em que projeções de vistas, cortes e fachadas terão grande valor para que o cliente possa vislumbrar o produto final. Contudo, para a produção de software não se tem elementos com potencial equivalente. Pressman (2001) denota três características importantes do software e suas implicações.

1.1.1.1 Software é um produto lógico e não físico

Em função de ser um produto lógico e não físico, o software é desenvolvido ou projetado por engenharia e não manufaturado no estrito senso. Na produção de automóveis, por exemplo, são feitos o projeto e a construção de um piloto, a partir do qual, inúmeras unidades são manufaturadas. Na produção de software ocorrem o projeto e o desenvolvimento de uma única entidade. Isto implica que o sucesso de um projeto de software é dado pela qualidade de uma única entidade e não pela qualidade de muitas entidades manufaturadas.

1.1.1.2 Software não desgasta

Um automóvel ou um prédio desgastam. Nesses casos, o reparo geralmente é feito pela substituição ou restauração do componente danificado ou desgastado por um novo. Software não desgasta, mas pode apresentar defeitos ou tornar-se funcionalmente inadequado devido à modificação das necessidades para as quais foi projetado. A manutenção de software (reparação de erros e evolução do produto) implica, muitas vezes, mudanças estruturais no produto e a necessidade de um novo projeto.

1.1.1.3 Software geralmente é feito sob medida e não a partir de componentes

Embora a indústria de software esteja movendo-se na direção de montagem de componentes, a maioria dos softwares continua sendo construída de maneira customizada, diferentemente da produção de um automóvel ou de um prédio, nos quais o projeto e a construção podem considerar uma série de componentes disponibilizados em catálogos específicos.

1.1.2 Desafios para o Desenvolvimento de Software

Além das dificuldades trazidas pelas próprias características, o desenvolvimento de software apresenta outros três grandes desafios: o do legado, o da heterogeneidade e o do fornecimento. O primeiro diz respeito à manutenção e atualização do software já produzido, sem impactar nos serviços que eles proveem. O segundo refere-se à produção de software capaz de executar em ambientes cada vez mais heterogêneos. O terceiro e último diz respeito à produção de software sob restrições crescentes de

prazo e custo para sistemas cada vez mais complexos, sem comprometer a qualidade dos mesmos.

As dificuldades em produzir software não são recentes. Na década de 1960, o avanço do poder computacional passou a possibilitar a produção de software cada vez maior e mais complexo. Uma abordagem informal para desenvolvimento de software não era mais possível. Esse momento foi chamado de crise do software. Em uma conferência em 1968 organizada para discutir a crise do software, o termo Engenharia de Software foi utilizado pela primeira vez. De lá para cá, a Engenharia de Software tem sido apontada como uma das principais alternativas para superar os desafios do desenvolvimento de software.

1.2 Engenharia de Software

Engenharia trata da aplicação dos princípios científicos à exploração dos recursos naturais, ao projeto e construção de comodidades e ao fornecimento de utilidades. A Engenharia de Software trata do projeto, construção e fornecimento de uma utilidade chamada de produto de software.

Sommerville (2010) define Engenharia de Software como a aplicação dos princípios científicos, métodos, modelos, padrões e teorias que possibilitam gerenciar, planejar, modelar, projetar, implementar, medir, analisar, manter e aprimorar um software. Ao aplicar a abordagem da Engenharia se estabelece que o software deve ser construído de maneira similar a outros produtos, ou seja, através da aplicação de um processo que conduza a um produto de alta qualidade para atender às necessidades das pessoas que o demandaram e vão utilizá-lo.

A Engenharia de Software não se refere apenas a processos de desenvolvimento de software, mas também a atividades tais como gerenciamento do projeto de software e o desenvolvimento de ferramentas e métodos para suportar a produção de software.

1.2.1 Processo

A base da Engenharia de Software é o processo de software. Processo de software envolve uma sequência de atividades que produzem uma variedade de documentos, culminando em um programa satisfatório e executável. O processo define um framework que forma a base para o controle do projeto de software e estabelece o contexto no qual métodos técnicos e ferramentas são aplicados, artefatos são produzidos, marcos são estabelecidos, qualidade é garantida e mudanças são

adequadamente gerenciadas. O processo de software será estudado no próximo capítulo.

1.2.2 Métodos

Os métodos são abordagens estruturadas para o desenvolvimento de software cujo objetivo é facilitar a produção de software. Atendem a um amplo escopo de atividades que incluem análise de requisitos, projeto, implementação, testes etc. Os métodos são baseados em um conjunto de princípios que governam cada área de tecnologia e podem incluir modelos, notações, regras, recomendações de projetos e diretrizes de processos.

1.2.3 Ferramentas

As ferramentas dão suporte à execução de procedimentos relacionados à produção de software. Abrangem um amplo escopo de diferentes tipos de programas e mecanismos usados para suportar as atividades do processo de software, como, por exemplo, ferramentas que auxiliam no planejamento, na edição, na prototipação, na modelagem, na programação, nos testes e na documentação. Quando as ferramentas são integradas, obtém-se um sistema de apoio ao desenvolvimento chamado de *Computer-aided Software Engineering* (CASE).

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar o conhecimento a respeito de Engenharia de Software recomenda-se a leitura de Pressman (2001) e Sommerville (2010).

RESUMO DO CAPÍTULO

Software é a parte programável de um sistema computacional. Produto de software refere-se a software que é produzido para ser entregue a clientes e usuários. As características próprias do software aliadas à demanda crescente por entrega de

sistemas mais complexos em menos tempo trazem grandes dificuldades para o desenvolvimento de software. Para desenvolver um produto de software de qualidade é necessário empreender um projeto segundo os preceitos da Engenharia de Software. A aplicação de engenharia em um projeto implica a adoção de um processo e a utilização de métodos e de ferramentas adequadas.

CAPÍTULO 2

PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

Este capítulo apresenta os conceitos de ciclo de vida de software, processo de software e modelo de processo de software. Vários modelos são abordados, dando-se ênfase aos modelos em cascata e iterativo-incremental. Um processo iterativo-incremental específico é detalhado e, por fim, são descritos um processo e um sistema que serão utilizados ao longo do livro para exemplificar questões relativas ao desenvolvimento de software.

2.1 Processo de Desenvolvimento de Software

Ciclo de vida do software é o período de tempo que se inicia com um conceito para um produto de software e acaba sempre que o software deixa de estar disponível para utilização, envolvendo as fases de desenvolvimento e de manutenção do produto. Dentre os processos do ciclo de vida do software está o processo de desenvolvimento de software.

Processo de desenvolvimento de software ou simplesmente processo de software é o conjunto de atividades e resultados associados que conduzem à produção de um produto de software (SOMMERVILLE, 2010). Envolve um conjunto de atividades técnicas e gerenciais para criar ou alterar um software a partir de requisitos.

Embora essas atividades possam variar, conforme a metodologia e tipo de software que está sendo produzido, um conjunto de atividades técnicas estão presentes em praticamente todos os processos de software. Essas atividades são as seguintes:

- requisitos, também chamada, muitas vezes, de especificação;
- análise de sistemas e projeto de software, muitas vezes tratados como uma única atividade chamada simplesmente de projeto (do inglês *design*);
- implementação, onde ocorre a codificação do software e, por isso mesmo, também chamada de codificação ou desenvolvimento;
- testes ou validação;

- implantação, instalação ou distribuição, na qual o software produzido é disponibilizado para o cliente;
- manutenção, operação ou evolução, na qual o software é mantido operacional recebendo manutenções corretivas ou evolutivas.

Em alguns processos, a divisão entre algumas atividades não é bem clara. Por exemplo, muitos tratam a atividade de análise e de requisitos como uma única atividade. Em outros, a análise é tomada em conjunto com o projeto e há os que dividem a atividade de requisitos em modelagem de negócio e requisitos.

Além das atividades técnicas, também chamadas de atividades fundamentais, muitos processos incorporam atividades gerenciais ou de apoio. Dentre as atividades de apoio, pode-se citar a Gerência de Projetos, a Gerência da Configuração e a Gerência da Qualidade.

2.2 Modelos de Processo de Software

Modelos de processo são representações abstratas de processos de software, sendo também chamados de modelo de ciclo de vida de software, de paradigma de engenharia de software ou de modelo de desenvolvimento.

Para construir software, deve-se adotar uma estratégia de desenvolvimento que englobe o processo, os métodos, as ferramentas e as fases de um projeto de engenharia. Essa estratégia, ou seja, a forma como são tomadas e estruturadas as atividades do processo e a sequência com que estas são executadas é definida pelo modelo de processo adotado.

Existem muitos modelos de processo, e a forma como são denominados e categorizados varia muito conforme os diversos autores. Neste livro serão apresentados quatro modelos, de acordo com a categorização de Sommerville (2010):

- modelo em cascata;
- modelos evolucionários;
- modelos formais;
- modelos orientados a reuso.

2.2.1 Modelo em Cascata

O modelo em cascata foi criado em 1970 e durante muito tempo foi o mais amplamente utilizado. Ele define uma abordagem sequencial para o desenvolvimento de software, tomando as atividades como fases separadas. Após a conclusão de cada fase, passa-se para a fase seguinte. É também chamado de ciclo de vida clássico, modelo progressivo ou modelo sequencial linear. A Figura 1 mostra um desenho esquemático deste modelo.

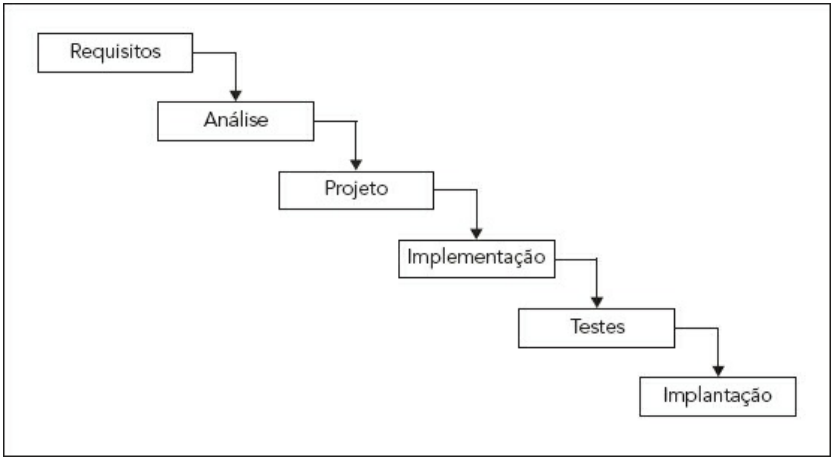


Figura 1 – Modelo em cascata.
Fonte: elaborada pelo autor.

Este modelo tem como principal vantagem a facilidade de gerenciamento do projeto. A principal desvantagem deste modelo refere-se à dificuldade de obter todos os requisitos no início do projeto e de acomodar mudanças depois que o processo se inicia. Além disso, os usuários terão de esperar até a instalação e liberação para ver como o software funciona. Projetos reais raramente comportam-se de maneira sequencial.

2.2.2 Modelos Evolucionários

Para superar as deficiências do modelo em cascata foram desenvolvidos os

modelos evolucionários. Estes determinam que o produto de software passe por sucessivos ciclos de evolução até que esteja pronto para ser entregue ao cliente. Em modelos evolucionários as atividades não são tomadas como fases e nem seguidas sequencialmente. Ao contrário, elas são repetidas a cada ciclo de evolução. Há três tipos de modelos evolucionários: o modelo evolucionário concorrente, o espiral e o iterativo-incremental.

2.2.2.1 Modelo Evolucionário Concorrente

Neste modelo todas as atividades são realizadas ao mesmo tempo. Caracteriza-se pelo desenvolvimento de uma implementação inicial, a qual é exposta à apreciação do cliente. Seu aprimoramento é feito por meio de muitas versões, até que um sistema adequado tenha sido desenvolvido. As atividades de requisitos, análise, projeto, implementação e testes ocorrem concorrentemente. Métodos de prototipação e de *Rapid Application Development* (RAD) podem ser considerados como modelos evolucionários concorrentes.

2.2.2.2 Modelo evolucionário espiral

Este modelo foi proposto por Boehm (1988). O processo é representado por uma espiral que começa no centro e gira em sentido horário. Cada volta representa uma fase no processo, não havendo fases fixas como especificação ou projeto. As voltas são escolhidas dependendo do que é requisitado.

A espiral pode ser divida em três a seis setores, sendo quatro setores o mais comum. Cada setor contempla um conjunto de atividades que são adaptadas a cada projeto. Este modelo tende a se adaptar a projetos grandes e inclui os métodos de prototipação para levantamento de requisitos e de análise de risco. Os riscos são explicitamente avaliados e resolvidos durante todo o processo. Desvantagens: o mesmo não foi amplamente usado e é de difícil gerenciamento.

2.2.2.3 Modelo evolucionário iterativo-incremental

Este modelo caracteriza-se por ser iterativo e incremental. Iterativo porque as atividades do software são iteradas – onde cada iteração equivale a uma minicascata. Incremental porque o desenvolvimento e a entrega são partidos em incrementos que

fornecem parte das funcionalidades requeridas. Quando o desenvolvimento de um incremento é iniciado, os requisitos são "congelados". Requisitos de prioridades mais altas são incluídos nas primeiras entregas. A Figura 2 apresenta a estrutura de um modelo evolucionário iterativo-incremental.

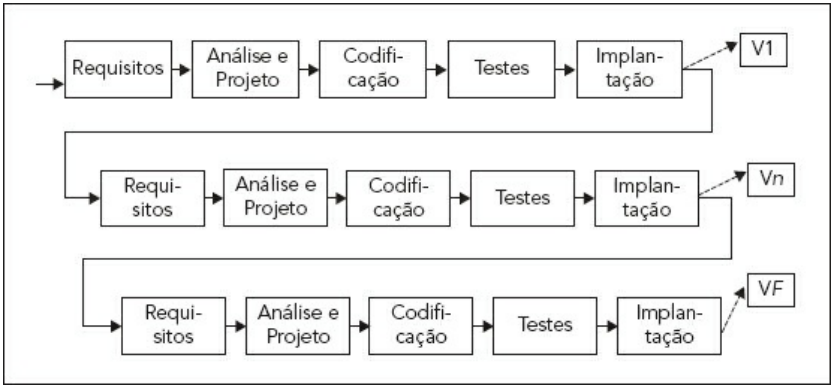


Figura 2 – Modelo evolucionário iterativo-incremental.
Fonte: elaborada pelo autor.

Este modelo traz como desvantagem a dificuldade em se identificar facilidades comuns a todos os incrementos. Entretanto, apresenta inúmeras vantagens, dentre as quais, destacam-se:

- valor ao cliente tende a ser entregue a cada incremento;
- os primeiros incrementos funcionam como um protótipo para ajudar a esclarecer os requisitos para os próximos incrementos;
- baixo risco de o projeto falhar completamente;
- as tarefas de maior prioridade tendem a receber mais testes.

2.2.3 Modelos Formais

Baseiam-se na produção de uma especificação formal do sistema e na transformação dessa especificação, utilizando-se métodos matemáticos para se construir um software. A aplicabilidade se justifica para sistemas críticos,

especialmente aqueles em que uma versão segura deve ser feita antes de o sistema entrar em operação. As atividades deste modelo diferem, em parte, das atividades dos demais modelos, englobando:

- definição de requisitos;
- especificação formal;
- transformação formal;
- teste.

Como vantagem, este modelo oferece maior garantia de que os programas atendam à especificação. Como desvantagens e limitações tem-se que consomem muito tempo, requerem habilidades especiais e treinamento para aplicar a técnica, são de difícil entendimento para leigos (usuário) e há dificuldade para formalizar alguns aspectos do sistema, tais como a interface com o usuário. São exemplos de linguagens para especificação formal VDM e Z.

2.2.4 Modelos Orientados a Reuso

Os modelos orientados a reuso têm como base a existência de um número significativo de componentes reutilizáveis. O processo de desenvolvimento concentra-se na integração desses componentes em um sistema e é, por isso, também chamado de desenvolvimento baseado em componentes. Sistemas são integrados a partir de componentes existentes ou componentes *Commercial-off-the-shelf* (COTS).

Este modelo foi viabilizado em função das tecnologias orientadas a objetos. A orientação a objetos enfatiza a criação de classes que encapsulam dados e os algoritmos que os manipulam. Se projetadas e implementadas adequadamente, as classes podem ser reutilizadas por diversas aplicações. Um conjunto de atividades distintas dos demais modelos, como as citadas seguir, são necessárias para se trabalhar com este modelo:

- identificação de componentes candidatos;
- busca por componentes;
- obtenção de componentes disponíveis;
- construção de componentes indisponíveis;

- integração.

A despeito das diferenças entre as atividades deste modelo e dos tradicionais modelos em cascata e evolucionários, este pode ser usado concomitantemente com os mesmos. As principais vantagens desse modelo advêm do fato de que, ao prover reuso, o tempo e o custo do desenvolvimento são reduzidos, ao passo que a qualidade tende a ser aumentada em função da utilização de componentes altamente testados.

2.3 Modelo em Cascata *versus* Modelo Iterativo-incremental

Dentre os modelos apresentados, o historicamente mais utilizado é o modelo em cascata. Entretanto, nos últimos anos sua aplicação passou a ser questionada e os modelos evolucionários foram apresentados como alternativa. Dos modelos evolucionários, o mais amplamente adotado tem sido o iterativo-incremental.

O modelo em cascata teve sua origem em outras engenharias, mas não se demonstrou muito adequado para a maioria dos projetos de software. Larman (2007) menciona pesquisas que mostram que a recomendação de aplicar o modelo em cascata está associada a altas taxas de falha, menor produtividade e maiores taxas de defeito. Tais pesquisas revelam que um típico projeto de software sofre 25% de modificações nos requisitos, sendo que, para projetos grandes, as taxas sobem para 35% a 50%. Além disso, outros estudos revelam que 45% das características nunca foram usadas, que 19% foram raramente usadas e que cronogramas e orçamentos iniciais, em cascata, variam até 400% em relação ao realizado.

Isto se deve ao fato, já mencionado, de que o desenvolvimento de software é um domínio de alta modificação e instabilidade, no qual não é viável definir os requisitos em detalhe no início do processo de desenvolvimento, como sugere o modelo em cascata. Os requisitos tendem a mudar ao longo do processo e, portanto, faz-se necessário um modelo de processo que se adapte a essas circunstâncias.

O modelo iterativo-incremental tem esta capacidade. Ele divide o projeto de um software em pequenos ciclos de desenvolvimento. Ao final de cada ciclo um conjunto de requisitos novos ou modificados é implementado e uma nova versão do software é disponibilizada. Assim, o software vai tornando-se completo de uma maneira incremental. Vale salientar que todas as metodologias ágeis preconizam a utilização de processos iterativos e incrementais.

2.4 Métodos Ágeis

Métodos ágeis são métodos que encorajam a agilidade, balizando-se pela escala de valores do manifesto ágil e adotando práticas de acordo com os princípios ágeis. Autores e representantes das mais variadas técnicas e metodologias ágeis formaram em fevereiro de 2001 a *Agile Alliance* e publicaram o Manifesto Ágil. Em síntese, este declara que os processos de desenvolvimento de software devem buscar valor em:

indivíduos e interações	sobre	processos e ferramentas,
software funcionando	sobre	documentação compreensiva,
colaboração do usuário	sobre	negociação de contratos e
responder à mudança	sobre	seguir um plano.

O manifesto reconhece o valor nos itens à direita, mas o desenvolvimento ágil valoriza mais os da esquerda. Ser simples é confundido com falta de controle e anarquia, quando na verdade exige muita disciplina e organização. Métodos ágeis reforçam o planejamento constante do projeto, o que minimiza riscos, considerando que o planejamento é mais importante que o plano. Dentre os principais métodos ágeis pode-se destacar:

- Agile Modeling
- Agile Software Development
- Agile Unified Process
- Crystal Clear
- Dynamic System Development Method
- Extreme Programming
- Feature-Driven Development
- Lean Software Development
- SCRUM.

2.5 Processo Unificado

O Processo Unificado (PU) é um processo iterativo-incremental originalmente

proposto por Jacobson, Rumbaugh e Booch em 1999 (JACOBSON, 1999). O Processo Unificado está estruturado em duas dimensões. Horizontalmente, tem-se a dimensão temporal. Na vertical, a dimensão de fluxos de trabalho ou disciplinas. Na dimensão temporal, dois conceitos são importantes: fase e iteração. A Figura 3 mostra a estrutura deste processo.

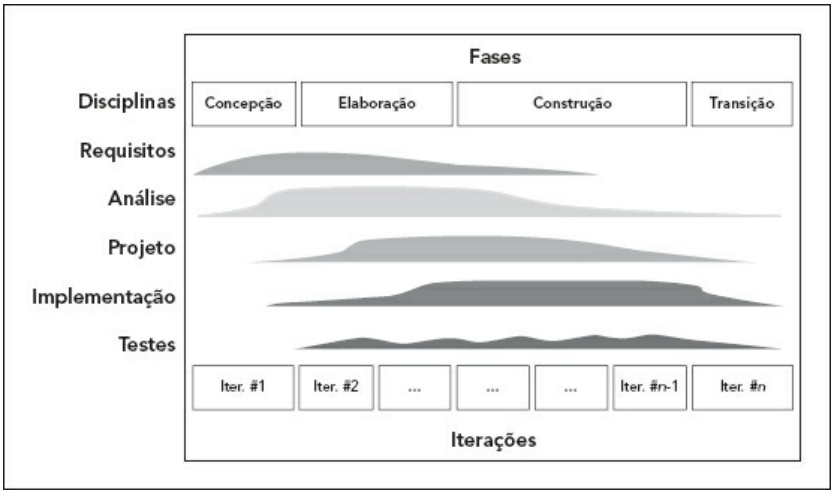


Figura 3 – Processo Unificado (JACOBSON et al., 1999).
Fonte: adaptada pelo autor.

2.5.1 Fases e Iterações

Fase é o período de tempo entre dois marcos importantes de progresso do processo em que determinados objetivos são alcançados, determinados artefatos são produzidos e decisões quanto à passagem para a fase seguinte são tomadas. Quando se passa para a fase seguinte, não há mais retorno à fase anterior. O Processo Unificado compreende quatro fases:

- concepção;
- elaboração;
- construção;

➜ transição.

Durante a fase de concepção é estabelecido o caso de negócio para o software e é delimitado o escopo do projeto. O caso de negócio inclui critérios de sucesso, avaliação de riscos e requisitos do projeto descritos em alto nível de abstração. Nesta fase pode ser criado um protótipo executável para avaliar questões técnicas relevantes ao projeto.

Os objetivos da fase de elaboração são estabelecer uma arquitetura para o software e prover uma base estável para as atividades de projeto e de implementação da fase de construção.

Durante a construção, o software é desenvolvido de uma maneira iterativa e incremental, até que o produto de software esteja completo, pronto para a disponibilização aos usuários. Os objetivos desta fase são refinar os requisitos restantes e completar o desenvolvimento com base na arquitetura estabelecida.

Na fase de transição, o software é instalado no ambiente de produção do cliente, os usuários são treinados e sua utilização é iniciada. Seu objetivo é garantir que o software esteja disponível para a comunidade usuária.

Em cada fase ocorrem iterações. Uma iteração é um ciclo de desenvolvimento completo que perpassa todas as atividades necessárias para a liberação de uma nova versão de software. Observa-se que uma fase pode conter várias iterações.

As iterações não devem ser longas. O recomendável é que cada iteração tenha uma duração fixa entre duas e seis semanas. Ao final de cada iteração deve haver o replanejamento das iterações posteriores. Isto serve para acomodar mudanças referentes a novos requisitos ou alterações nos requisitos já implementados.

2.5.2 Disciplinas (Fluxos de Trabalho)

A dimensão dos fluxos de trabalho do PU define cinco disciplinas, todas de escopo técnico: requisitos, análise, projeto, implementação e testes.

Em cada disciplina está inserido um conjunto de atividades e artefatos correlacionados. Uma atividade descreve as tarefas executadas para criar ou modificar artefatos, juntamente com as técnicas, as diretrizes e os responsáveis pela sua realização, incluindo a utilização de ferramentas para ajudar na sua automação. Um artefato é qualquer documento ou executável produzido, manipulado ou consumido pelo processo, com a finalidade de auxiliar um projeto de desenvolvimento de um software. Em um processo definido, ou seja, explicitado e documentado, os artefatos

podem ser elaborados conforme *templates* padronizados.

As áreas hachuradas representam o esforço dedicado a cada disciplina ao longo das fases. Observa-se que praticamente todas as disciplinas ocorrem em todas as fases, mas com maior ou menor esforço. Contudo, determinadas disciplinas predominam em certas fases. Por exemplo, a disciplina de requisitos concentra seu maior esforço nas fases iniciais, principalmente nas fases de concepção e elaboração, enquanto que o maior volume de implementação ocorre nas fases de construção.

As disciplinas do processo também possuem seus objetivos. O da disciplina de requisitos é entender, capturar e gerenciar os requisitos que o software deverá atender. Os da análise e do projeto são respectivamente analisar o sistema e projetar um software para atender os requisitos mapeados. A implementação busca a codificação do software projetado, enquanto a disciplina de testes objetiva a validação do sistema.

2.5.3 Características do PU

Além de sua estrutura, o PU possui outras características marcantes. Uma delas é que o Processo Unificado tem um forte apelo à modelagem visual, endossando o uso da *Unified Modeling Language* (UML), uma linguagem de modelagem unificada e definida como padrão pelo *Object Management Group* (OMG).

Outra característica peculiar do Processo Unificado é o fato de ele ser dirigido por casos de uso (Figura 4). Por ora, pode-se pensar em casos de uso como interações entre o usuário e o sistema com vistas a cumprir com um objetivo. A questão-chave no PU é que todo o processo gira em torno de casos de uso. Os casos de uso serão usados na disciplina de requisitos para identificar e especificar requisitos funcionais. Na análise, serão utilizados como insumo para a modelagem conceitual. O projeto do software será pensado de forma a realizar os casos de uso definidos que, por fim, serão empregados para validar o sistema.

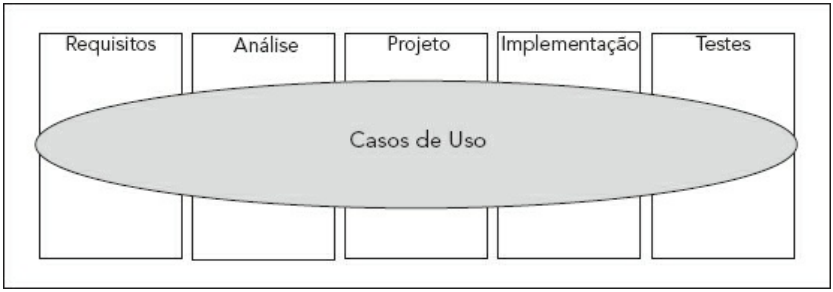


Figura 4 – Casos de uso no processo de desenvolvimento.
Fonte: elaborada pelo autor.

2.5.4 Rational Unified Process (RUP)

O *Rational Unified Process* (RUP) é um processo comercial que estende o Processo Unificado com duas disciplinas técnicas, modelagem de negócio e distribuição e um conjunto de disciplinas de apoio: gestão de projeto, gestão de configuração e gestão do ambiente (KRUCHTEN, 2001). Além disso, a análise e o projeto são tomados como uma única disciplina.

A disciplina de modelagem de negócio tem por objetivo entender a estrutura e a dinâmica (os processos de negócio) da organização na qual um sistema deve ser implantado. A distribuição visa disponibilizar o software produzido aos clientes e usuários. A gestão de configuração controla as modificações e mantém a integridade dos artefatos do projeto. A gestão de projeto descreve as estratégias para a condução gerencial do projeto, incluindo as atividades de gestão de riscos, supervisão e acompanhamento. Já a gestão de ambiente controla o ambiente de trabalho para o desenvolvimento do software.

2.6 Processo e Sistema de Estudo

Para desenvolver os conteúdos, este livro toma como base um processo genérico baseado no PU e um sistema exemplo chamado de Sistema de Gestão de Transportes (SGT).

2.6.1 Caracterização do Processo Base

O livro cobre as disciplinas técnicas de Modelagem de Negócio, Requisitos, Análise, Projeto e Testes. Embora a Modelagem de Negócio não seja uma disciplina definida no PU, ela é contemplada no livro devido a sua importância em projetos de software de sistemas de informação. Por outro lado, a disciplina Implementação não é contemplada, pois seu estudo vai além do escopo deste livro. Da mesma forma, nenhuma disciplina de apoio, tal como Gerência de Projetos, é apresentada.

O início do processo se dá pela fase de Concepção. Na fase de Elaboração, artefatos iniciados da Concepção, tais como Glossário e Modelo de Casos de Uso, devem ser complementados e uma série de outros artefatos referentes à arquitetura do software devem ser desenvolvidos. Na medida em que se avança na fase de Elaboração, diminui a intensidade de esforço dedicado à disciplina de Modelagem de Negócio. O esforço relacionado a Requisitos se mantém ao passo que o esforço dedicado às disciplinas de Análise e Projeto é intensificado. Como o processo é iterativo e incremental, a cada iteração é dedicado esforço à Implementação e Teste dos respectivos requisitos.

2.6.2 Descrição do Sistema Exemplo

Uma descrição geral do sistema exemplo, Sistema de Gestão de Transportes (SGT), é apresentada no Quadro 1. A descrição é incompleta, não contemplando características relevantes de um sistema real. Por outro lado, algumas características são introduzidas artificialmente de forma a viabilizar a explanação de tópicos importantes do conteúdo deste livro.

Quadro 1 – Descrição Sucinta do Sistema de Gestão de Transportes (SGT)

Uma organização mantém uma frota de veículos para atender a necessidade de transporte d materiais e de pessoas vinculadas à empresa (colaboradores e terceiros). São exemplos de transportes levar um diretor ao aeroporto, ou um grupo de funcionários a um evento, buscar convidados visitantes no aeroporto, transportar um consultor diariamente do hotel à organização e vice-versa, leva um equipamento para conserto etc.

É responsabilidade do setor de transporte da organização gerenciar a frota de veículos e controlar manutenção dos mesmos, encaminhando-os à manutenção preventiva de acordo com a quilometragen percorrida, ou à manutenção corretiva caso haja a necessidade de alguma reparação ou conserto.

Para conduzir os veículos, a organização conta com um pequeno quadro de motoristas profissionais sobre os quais mantém, além dos dados cadastrais básicos, informações sobre habilitação infrações cometidas, reclamações recebidas e acidentes.

Atualmente, funcionários da organização telefonam para o setor e solicitam um transporte d pessoas ou de material. No primeiro caso, informam o nome dos passageiros e os horários e endereço

de partida e chegada. No segundo, informam os dados do material. Um auxiliar verifica em um planilha a disponibilidade de veículos e de motoristas para atender a solicitação, agenda o transport reservando um veículo e um motorista, e confirma a solicitação. Além das solicitações, todos o demais controles e cadastros (motoristas, veículos, manutenções, transportes efetuados) são realizados mantidos manualmente pelos auxiliares.

A organização deseja automatizar o sistema desta área através do desenvolvimento de um softwar específico, que será denominado de Sistema de Gestão de Transporte.

O objetivo é que as solicitações passem a ser feitas exclusivamente por funcionários da empres autorizados a solicitar transporte, denominados solicitantes. Um solicitante acessa o sistema, inform os dados do transporte e o sistema, considerando as características do transporte e a disponibilidade d motoristas e veículos, agenda a solicitação emitindo uma ordem de transporte para o motorista alocad e um e-mail para cada passageiro. Caso não seja possível agendar a solicitação (p. ex.: o veícul adequado para o transporte está em manutenção, realizando um transporte ou alocado em outr solicitação), o sistema envia um e-mail para o gerente de transporte. Este tomará as providência necessárias ou iniciará um processo para locação de transporte (táxi ou outra empresa especializada er transporte).

Para agilizar o preenchimento da solicitação e diminuir a incidência de erros, o gerente do setc solicitou que o sistema apresentasse ao solicitante, durante o preenchimento da solicitação, uma list de passageiros e de locais de partida e destino previamente cadastrados. Caso um passageiro ou ur local não esteja cadastrado, o solicitante poderá cadastrá-los durante o processo de preenchimento d solicitação.

O sistema deve gerenciar os transportes efetivamente realizados, dando baixa na respectiv solicitação. É responsabilidade do motorista informar ao sistema a data, a hora e a quilometragem d início e do fim do transporte.

O gerente de transporte solicitou ser notificado por e-mail quando um veículo ating quilometragem que demanda manutenção preventiva.

Fonte: elaborado pelo autor.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para informações sobre gestão de projeto sugere-se a leitura do capítulo sobre a disciplina de gestão de projetos do RUP (KRUCHTEN, 2001), a leitura do livro sobre SCRUM de Cohn (2011) e leitura do *Project Management Body of Knowledge* – PMBOK do PMI (2008). Para aprofundar o conhecimento a respeito do PU ou RUP, recomenda-se a leitura, respectivamente, de Jacobson (1999) e Kruchten (2001). Mais informações a respeito de métodos ágeis podem ser obtidas em www.agilemanifesto.org.

RESUMO DO CAPÍTULO

Ciclo de vida do software é o período de tempo que se inicia com um conceito para um produto de software e acaba quando o software deixa de estar disponível para utilização. Processo de desenvolvimento de software é o conjunto de atividades e resultados associados que conduzem à produção de um produto de software. Modelos de processo são representações abstratas de processos de software. O início do ciclo de vida de um produto de software se dá pela escolha de um modelo de processo para o seu desenvolvimento. Dentre os modelos de processo, o historicamente mais utilizado é o modelo em cascata. Contudo, na maioria dos casos, este modelo não é o mais adequado, conduzindo muitas vezes ao fracasso de um projeto de software. Modelos iterativos-incrementais, tais como o PU e uma gama de métodos ágeis, apresentam-se como alternativa. O PU endossa a modelagem visual através da UML e é dirigido por casos de uso. Apresenta quatro fases (concepção, elaboração, construção e transição) e cinco disciplinas técnicas (requisitos, análise, projeto, implementação e testes). Alguns processos, como o RUP, que é uma extensão do PU, apresentam, além das disciplinas técnicas, disciplinas gerenciais. Dentre as disciplinas gerenciais, destaca-se a de gestão de projeto.

CAPÍTULO 3

UNIFIED MODELING LANGUAGE

Este capítulo introduz a UML, uma linguagem para modelagem de sistemas de software. São apresentados os diagramas, os mecanismos gerais, observações a respeito de sua aplicação e um breve histórico.

A *Unified Modeling Language* (UML) é uma linguagem para modelagem de sistemas de software. O OMG (2010a) define UML como uma linguagem visual para especificar, construir e documentar os artefatos dos sistemas. A UML é a notação padronizada de diagramas que podem ser utilizados para desenhar, com algum texto, representações de software, principalmente software orientado a objetos, em diversos estágios de desenvolvimento. As representações do software não se restringem a aspectos concretos, como classes de software escritas em uma determinada linguagem de programação, componentes de software ou componentes de hardware, mas envolvem também aspectos conceituais, como, por exemplo, processos de negócio e funcionalidades do sistema.

3.1 Diagramas

A UML, a partir de sua versão 2.0, passou a ser composta por 13 diagramas, conforme descreve o Quadro 2. Como muitos modelos foram desenvolvidos com diagramas de versões anteriores da UML é importante destacar que o diagrama de estados passou a chamar-se diagrama de máquina de estados, o diagrama de colaboração a denominar-se diagrama de comunicação e que três novos diagramas foram adicionados à especificação: diagrama de visão geral de integração, diagrama de temporização e diagrama de estrutura composta.

Quadro 2 – Diagramas da UML

Diagrama	Descrição
Diagrama de Atividades	Utilizado para modelar o fluxo de atividades de determinado processo do sistema, podendo ser um processo de negócio, um fluxo de execução de um caso de uso ou até mesmo um fluxo de execução de uma operação de classe complexa.

Diagrama de Casos de Uso	Serve para modelar as interações entre atores e o sistema. Atores são os agentes externos ao sistema, tais como usuários, sensores ou outros sistemas. Visa representar as funções que o sistema deve realizar para atender as demandas dos atores. O diagrama de casos de uso é útil para capturar e comunicar requisitos funcionais.
Diagrama de Classes	É usado para modelar a estrutura de classes do sistema, representando as classes, seus atributos, suas operações e seus relacionamentos. É o mais tradicional e, possivelmente, o mais importante modelo para projeto de softwares orientado a objetos.
Diagrama de Componentes	Serve para modelar os componentes de software (bibliotecas, executáveis, arquivos de configuração, tabelas etc.) que serão utilizados ou desenvolvidos. Pode exibir o conteúdo dos componentes (p. ex.: classes e interfaces) e o relacionamento entre os mesmos.
Diagrama de Comunicação	De forma similar ao Diagrama de Sequência, o Diagrama de Comunicação pode ser empregado para representar ou investigar a troca de mensagens entre objetos. Entretanto, o foco não está na sequência temporal em que as mensagens são trocadas, mas sim, nas conexões entre os objetos diagramados.
Diagrama de Estrutura Composta	Lançado na UML 2.0, descreve a estrutura interna de um classificador (p. ex.: classe ou componente) representando as partes que o compõem e suas interfaces.
Diagrama de Implantação	Representa a infraestrutura física que o software requer para operar. Modela os nodos computacionais (servidores, estações de trabalho, equipamentos de comunicação de dados etc.) e as conexões entre os mesmos, podendo-se destacar os protocolos de comunicação que serão utilizados.
Diagrama de Máquina de Estados	Este diagrama é usado para modelar os estados e as transições de estado de objetos do modelo. Um uso muito comum é o de investigar o ciclo de vida de instâncias de uma classe.
Diagrama de Objetos	Este diagrama representa instâncias de classes do sistema (objetos), com seus valores e relacionamentos, em uma determinada situação em que se queira modelar ou investigar.
Diagrama de Pacotes	Este diagrama é utilizado para representar o conteúdo de um pacote e o relacionamento deste com outros pacotes. Pacotes podem ser usados para agrupar componentes do modelo de acordo com a arquitetura do software ou simplesmente para organizar o modelo.
Diagrama de Sequência	Este diagrama é utilizado para representar ou investigar a ordem temporal da troca de mensagens entre objetos visando à realização de determinado processo do sistema. Este processo poder ser um caso de uso, um cenário de um caso de uso ou uma colaboração entre objetos para executar uma operação de sistema.
Diagrama de Temporização	Este diagrama foi lançado na UML 2.0. Serve para modelar a mudança de estados de um objeto ao longo do tempo.
Diagrama de Visão Geral de Interação	Lançado na UML 2.0, é similar ao diagrama de atividades. Entretanto, os nodos do fluxo não são atividades (nodos de ação), mas sim, qualquer tipo de diagrama de interação (p. ex.: diagrama de sequência ou diagrama de comunicação).

Fonte: elaborado pelo autor.

3.2 Mecanismos Gerais

A UML provê alguns mecanismos gerais que facilitam e complementam a

especificação dos elementos modelados. Dentre os mecanismos gerais, três são mais frequentemente usados: notas explicativas, estereótipos e restrições (*constraints*).

As notas explicativas permitem associar um texto a um elemento do modelo. Para tal, basta criar um elemento gráfico de nota explicativa no diagrama, escrever o texto e associá-lo ao elemento desejado. A Figura 5 apresenta a notação gráfica de uma nota explicativa. No exemplo, associa-se uma nota explicativa a uma classe.

Figura 5 – Notação gráfica para nota explicativa na UML.
Fonte: elaborada pelo autor.

Estereótipos permitem qualificar um elemento do modelo atribuindo a este uma nova característica que seja de interesse destacar sem, contudo, alterar a essência do elemento. Em outras palavras, um estereótipo pode ser usado para estender a semântica de um elemento do diagrama. A UML predefine alguns estereótipos, mas permite a criação de outros conforme o interesse de quem está modelando. Os estereótipos podem ser gráficos ou de rótulo. Os de rótulo são nomes que vêm delimitados por caracteres *guillemets* (“«” e “»”) ao passo que os estereótipos gráficos são representados na forma de ícones. A Figura 6 mostra uma classe com o estereótipo de rótulo predefinido da UML «ator» e sua respectiva representação na forma de ícone.

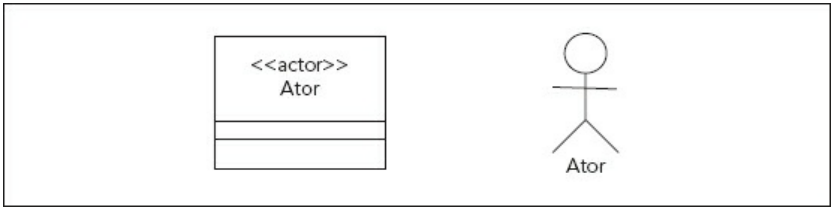


Figura 6 – Notações para estereótipos na UML.
Fonte: elaborada pelo autor.

Restrições especificam condições que devem ser verdadeiras ou impõem limitações à semântica natural de um elemento para que o modelo represente com maior fidelidade o domínio que está sendo modelado. Restrições são expressões ou textos delimitados por chaves e podem ser vinculadas a diversos elementos de um modelo. A UML define a linguagem chamada de *Object Constraint Language* (OCL) que permite especificar restrições. A Figura 7 exibe um exemplo de uma restrição aplicada ao atributo valor da classe Inteiro Positivo, determinando que o mesmo seja positivo.

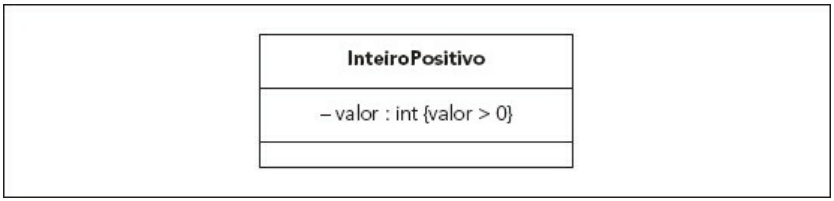


Figura 7 – Notação gráfica de uma restrição na UML.
Fonte: elaborada pelo autor.

3.3 Aplicação da UML

Larman (2007) apresenta três perspectivas de uso para UML: perspectiva conceitual, de especificação e de implementação. Na primeira, diagramas são interpretados como descrevendo coisas em uma situação do mundo real ou domínio de interesse. Na segunda, os diagramas (usando a mesma notação da perspectiva conceitual) descrevem abstrações de software ou componentes com especificações e interfaces, mas sem nenhum comprometimento com uma implementação em particular (p. ex.: C# ou Java). Na terceira, os diagramas descrevem implementações de software em uma tecnologia particular.

Um mesmo elemento gráfico da UML pode ser interpretado de maneira distinta, conforme a perspectiva em que o mesmo está sendo usado. Por exemplo, no PU, quando os símbolos gráficos de classe da UML são desenhados no modelo de domínio, eles são interpretados como conceitos de domínio ou classes conceituais, pois o modelo de domínio mostra uma perspectiva conceitual do sistema. Quando os mesmos símbolos são desenhados no modelo de projeto do PU eles são chamados de classes de projeto, pois o modelo de projeto mostra uma perspectiva de especificação ou implementação do software. As classes de projeto geralmente referem-se a classes

de software, estejam elas ainda em nível de especificação ou já representando a implementação em uma linguagem específica, tal como C# ou Java.

É oportuno comentar que a notação da UML é bastante ampla e completa. Entretanto, é possível modelar a grande maioria dos sistemas utilizando apenas um conjunto limitado de elementos da notação, visto que boa parte da UML é designada para modelagem de situações específicas de diversos domínios.

3.4 Histórico

As primeiras linguagens de programação orientada a objetos surgiram e evoluíram nos anos 1960 e 1970. Na década seguinte, a orientação a objetos começou a difundir-se e passou a ser utilizada pela indústria de Tecnologia da Informação. Muitos métodos orientados a objetos passaram a ser concebidos. Com a rápida proliferação do uso da Orientação a Objetos (OO), na metade da década de 1990 havia quase 50 métodos OO, cada um com seu próprio enfoque, procedimentos, técnicas e notação. Dentre os métodos influentes da época estavam presentes os seguintes:

- o método *Object Modeling Technique* (OMT), de Jim Rumbaugh;
- o método Booch para análise e projeto OO de Grady Booch (BOOCH, 1994);
- o método OOSE (*Object-Oriented Software Engineering*), proposto por Ivar Jacobson (JACOBSON, 1994).

A diversidade de métodos e a ausência de uma notação padrão traziam dificuldades ao desenvolvimento de software OO. Em 1994, Booch e Rumbaugh, com o intuito de combinar seus métodos, criaram o rascunho de um método, o Método Unificado versão 08. Em 1996, Jacobson juntou-se a eles. A ideia era incorporar ao Método Unificado elementos e conceitos dos métodos OOSE e *Objectory*. Entretanto, eles optaram por reduzir o escopo do seu esforço e enfocar em uma notação comum de diagramação em vez de produzir um método comum. Como resultado surgiu, em 1996, a versão 0.9 da UML. Em paralelo, o OMG, convencido por vários vendedores de TI de que um padrão aberto era necessário, passou a contribuir na elaboração da UML. Assim, em 1997, foi lançada a UML 1.0. Desde então, a UML vem sendo evoluída em novas versões. Em 1999, já na versão 1.3, a UML passou a ser mantida pelo OMG. Em julho de 2005, foi lançada a versão 2.0. A UML estabeleceu-se, de fato e de direito, como notação de diagramação padrão para a modelagem orientada a

objetos, sendo hoje amplamente utilizada.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Especificações e outras informações sobre a UML podem ser obtidas em www.omg.org e em www.uml.org. Para mais detalhes a respeito da OCL sugere-se acessar o site www.omg.org/spec/OCL. Uma lista de ferramentas UML pode ser encontrada em case-tools.org/uml.html. Para obter exemplos de diagramas UML sugere-se a leitura do livro Guedes (2009).

RESUMO DO CAPÍTULO

A UML é uma linguagem visual para especificar, construir e documentar os artefatos de sistemas de software, composta, em sua versão 2.0, por 13 diagramas. Além dos diagramas, dispõe de mecanismos gerais, tais como notas explicativas, estereótipos e restrições para facilitar a modelagem. Surgiu do esforço de três autores de métodos orientados a objetos na tentativa que padronizar a modelagem OO frente à proliferação de métodos e notações na década de 1990. Hoje é homologada e mantida pelo OMG.

CAPÍTULO 4

MODELAGEM DE NEGÓCIOS

Este capítulo trata da disciplina de modelagem de negócio. Os principais artefatos do processo produzidos ou iniciados nessa disciplina são apresentados. O uso do diagrama de atividades da UML para modelagem de negócio é demonstrado.

Esta disciplina tem o objetivo de definir o contexto no qual o software irá operar. Trata de compreender a estrutura e a dinâmica da organização, ou seja, seus processos de negócio.

Muitos processos de software não contemplam a disciplina de Modelagem de Negócio, geralmente por entender que ela não compete à Engenharia de Software ou não pertence ao escopo do processo de software. De certa forma, isso é verdade. Entretanto, quando foca-se apenas em entender e implementar os processos de negócio atuais de uma organização, pode-se não perceber que a melhor solução seria a modificação de alguns desses processos. Em sistemas de informação empresariais, esta atividade reveste-se de especial importância. Em outros domínios, como, por exemplo, o de desenvolvimento de aplicativos para computação pessoal, esta disciplina tem menor relevância, sendo suas atividades, em muitos casos, executadas dentro da disciplina de requisitos.

Três principais artefatos são trabalhados nesta atividade:

- documento de visão;
- glossário, que registra o vocabulário comum e específico do projeto;
- modelo de negócio.

4.1 Documento de Visão e Glossário

O documento de visão é um sumário executivo do projeto. Deve ser escrito em linguagem natural e tem por objetivo dar rapidamente uma visão do projeto para quem dela precisar, desde o patrocinador do projeto até um programador recém-chegado à equipe. Este documento pode conter várias seções escritas em texto em formato livre,

dos quais se destacam:

- contexto do negócio/situação atual;
- objetivos do projeto;
- escopo do projeto;
- premissas;
- restrições;
- visão geral do produto.

Na seção contexto do negócio e situação atual deve-se descrever a situação atual, os problemas existentes, os sistemas existentes, as falhas nos processos e os aspectos relevantes para o negócio. Na seção de objetivos do projeto devem ser descritos os objetivos do projeto em relação aos benefícios que o produto a ser desenvolvido irá trazer para o cliente.

O escopo do projeto deve descrever o conjunto de problemas que o software se propõe a resolver, bem como os aspectos que não são cobertos. O primeiro passo para delimitar o escopo do projeto é conhecer quem são os envolvidos (*stakeholders*) e obter o consenso dos mesmos quanto ao problema a ser resolvido. Dentre os envolvidos, podem-se destacar os próprios usuários do software, os representantes do cliente e o patrocinador (*sponsor*) do projeto.

Premissas são elementos pressupostos como verdadeiros para a execução do projeto que estão fora do escopo do mesmo. As premissas são caracterizadas pelas hipóteses de trabalho. O objetivo delas é deixar claro para todas as partes interessadas quais são as suposições na qual o projeto está baseado.

Restrições são elementos impostos para a execução do projeto. Ao contrário das premissas, esses elementos pertencem ao escopo do projeto e impedem a sua execução. As restrições representam condições que limitam o espaço de soluções ou que são impostas sobre o projeto. Essas restrições podem ser de cunho tecnológico, orçamentário e de cronograma, tais como linguagens de programação, bancos de dados, padrões e restrições legais que devem ser adotados e seguidos.

A seção visão geral do produto deve fornecer uma visão de alto nível do produto focada nas necessidades dos interessados e que apresenta as características fundamentais desejadas e a capacidade do produto ser desenvolvido. As integrações entre sistemas também são identificadas. Pode conter elementos, como denominação

do produto, diagrama de contexto e lista de características do produto. O diagrama de contexto pode ser apresentado sob a forma de um diagrama de caso de uso,[1] contendo um único caso de uso representando o sistema e todos os atores que com ele se relacionam.

É importante capturar todo o vocabulário comum e específico do sistema em um glossário. O glossário contém uma lista de termos, com suas definições, formato e regras de validação. Em geral, a construção do glossário tem início na fase de concepção e pode continuar sendo executada nas demais fases, sempre que for necessário definir um novo vocábulo comum ou alterar o glossário.

4.2 Modelo de Negócio

Para a modelagem do negócio, podem-se usar diversos diagramas e notações. Na UML, a forma mais usual é através de diagramas de atividade. Eles mostram os fluxos dos processos de negócio e os principais papéis e objetos de negócio envolvidos. Similares ao diagrama de atividades da UML, mas com recursos adicionais, são os diagramas utilizando a *Business Process Modeling Notation* (BPMN). A BPMN é uma notação homologada pelo OMG e seu uso vem tornando-se cada vez maior.

Outra possibilidade é desenvolver um modelo de negócio composto de modelos de casos de uso de negócio e de modelos de objetos de negócio. Um caso de uso de negócio é diferente de um caso de uso de sistema. No primeiro, o foco está em compreender como um negócio responde a um usuário ou a um evento. No segundo, o foco está nas interações do usuário com o software. Este método tem a vantagem de prover uma fácil derivação desses modelos para os modelos posteriores de análise e projeto.

4.3 Diagrama de Atividades

O diagrama de atividades serve para modelar fluxos de controle de ações e objetos. Ações são unidades atômicas de comportamento, enquanto objetos representam instâncias de classes, geralmente representados em um diagrama de atividades para demonstrar dados que transitam entre ações. Uma atividade contém uma ou mais ações e pode conter outras atividades (subatividades). O fluxo é modelado pela diagramação de nodos de ação, nodos de objetos e nodos de controle. Uma ação pode ser iniciada pela conclusão da ação anterior, pela ocorrência de eventos

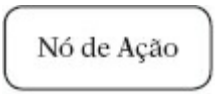
ou pela disponibilização de um objeto.

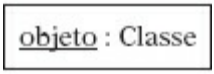
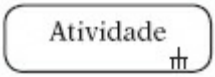
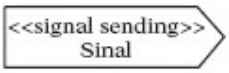
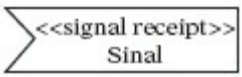
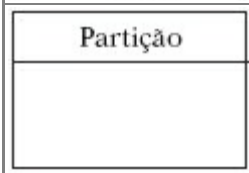
Diagramas de atividade podem ser utilizados para modelar atividades em diversos níveis de abstração, tais como:

- um processo de negócio;
- a lógica de um caso de uso;
- a implementação de uma operação;
- um algoritmo.

O Quadro 3 apresenta os principais elementos da notação para diagramas de atividades.

Quadro 3 – Principais elementos gráficos do diagrama de atividade

Elemento Gráfico	Descrição
	Nó de ação e de atividade – representa uma unidade de comportamento em determinado contexto. Pode ser uma ação quando se modela uma atividade composta por ações, pode ser uma atividade quando se modela um processo composto por atividades ou pode ser um passo computacional quando se modela um algoritmo.
	Fluxo de controle e fluxo de objeto – conectam os diversos nós do modelo representando a transição do controle entre os mesmos. A seta indica o sentido da transição. Um fluxo de controle no Diagrama de Atividades pode conter uma condição de guarda, que é uma expressão lógica representada entre colchetes e que determina quais condições devem ser verdadeiras para que a transição ocorra.
	Nó inicial – é um nó de controle usado para representar o início de um fluxo da atividade representada no diagrama.
	Nó final de atividade – indica o fim de um fluxo da atividade representada no diagrama.
	Nó de decisão – é um nó de controle usado para se escolher um dentre dois ou mais fluxos de controle. Possui um fluxo de entrada e vários de saída. Cada fluxo emergente do nó de decisão deve conter uma condição de guarda. O fluxo escolhido será aquele que satisfizer a condição.
	Nó de bifurcação e de junção – nó de controle utilizado para separar ou reunir fluxos de controle concorrentes. O nó de bifurcação recebe um fluxo de entrada e cria dois ou mais fluxos paralelos. O nó de junção recebe dois ou mais fluxos de entrada e une os fluxos em um único. Ao passo que no nó de decisão apenas um dos fluxos derivados será executado, no nó de bifurcação todos os derivados serão executados. De forma análoga, uma ação seguinte ao nó de junção somente será executada quando todos os

	fluxos concorrentes forem concluídos.
	Final de fluxo – representa o final de um dos fluxos possíveis de uma atividade, mas não o fim da atividade, como no caso do nó final de atividade.
	Nó de objeto – representa um objeto que pode ser utilizado por uma ação. Uma ação pode gerar, alterar ou consultar um objeto.
	Conector – elemento gráfico utilizado para conectar fluxos que precisaram ser divididos para facilitar o aspecto visual do diagrama ou porque sua continuação se dá em outro diagrama. O caractere # deve ser substituído por um identificador, geralmente um número ou uma letra. O mesmo identificador deve ser usado onde o fluxo é interrompido e onde o mesmo continua.
	Ação de chamada de comportamento – este nó invoca a execução de outra atividade, representada em outro diagrama.
	Ação de envio de sinal – é uma ação que representa a ocorrência de um evento e o envio de um sinal para um objeto ou uma ação.
	Ação de aceitação de sinal – é uma ação que representa a espera da ocorrência de um evento. Enquanto o sinal não for recebido, porque o evento que o gera ainda não ocorreu, o fluxo fica parado.
	Partição – eram chamadas em versões anteriores de raias de natação. As atividades/ações de um processo podem ser distribuídas por vários agentes que o executarão. Para representar este tipo de situação, utilizam-se partições. As partições dividem o diagrama de atividades em compartimentos e cada um deles contém as atividades que serão realizadas pelo agente correspondente. As partições podem ser verticais ou horizontais.

Fonte: elaborado pelo autor.

4.4 Modelagem de Negócio com Diagrama de Atividades

Para modelar negócios, um determinado processo pode ser representado graficamente por um diagrama de atividades, facilitando o entendimento e a análise do mesmo. Isto favorece a identificação de atividades que possam ser informatizadas, bem como a proposição de alterações visando à melhoria do processo. Para exemplificar o processo de modelagem, são apresentados dois diagramas de atividade. O primeiro apresenta o diagrama de atividades do processo de negócio para solicitação de transporte tal como o mesmo é executado atualmente, conforme a descrição do sistema exemplo (Figura 8). Posteriormente, é apresentado um diagrama

representando o mesmo processo, porém já contemplando modificações para atender as necessidades e solicitações demandadas (Figura 9).

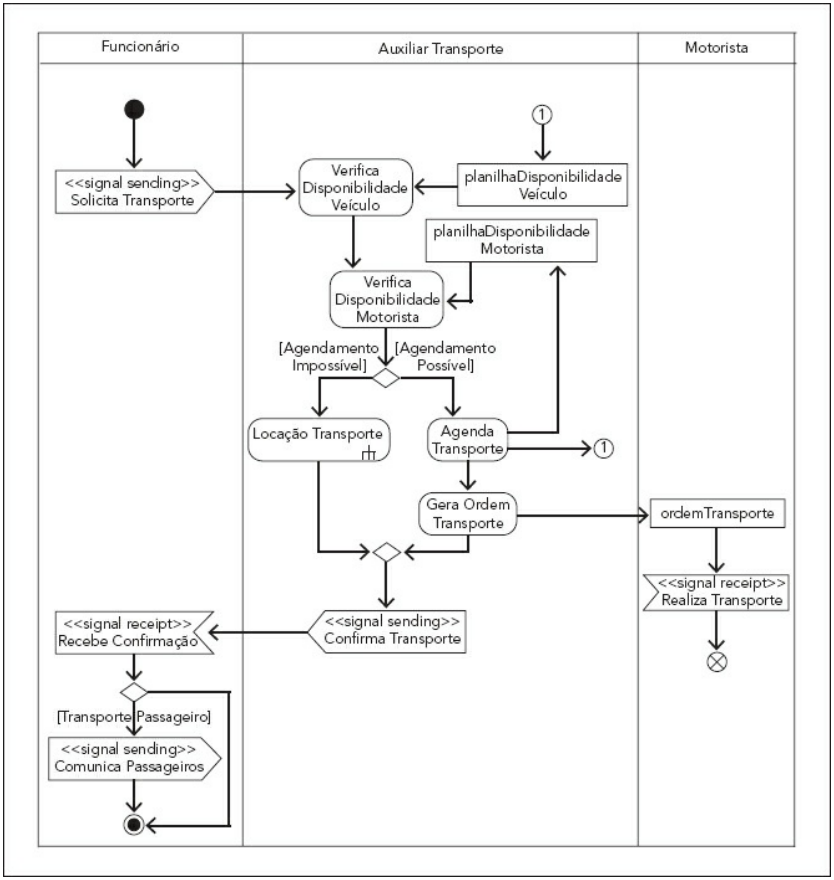


Figura 8 – Diagrama de atividade do processo atual para solicitação de transporte.
Fonte: elaborada pelo autor.

Um objeto (p. ex.: planilhaDisponibilidadeVeiculo), no nível de abstração no qual se está modelando (nível de processo), não é uma instância de uma classe computacional. Trata-se de um objeto de negócio, que armazena valores sobre os quais

se tem alguma necessidade de utilização. Apenas para exemplificar o uso de conectores, o fluxo entre a atividade Agenda Transporte e o objeto planilhaDisponibilidadeVeiculo foi interrompido. No exemplo em questão, o conector de número (1) conecta elementos dentro do próprio diagrama, mas poderia ser utilizado para conectar elementos em diagramas distintos.

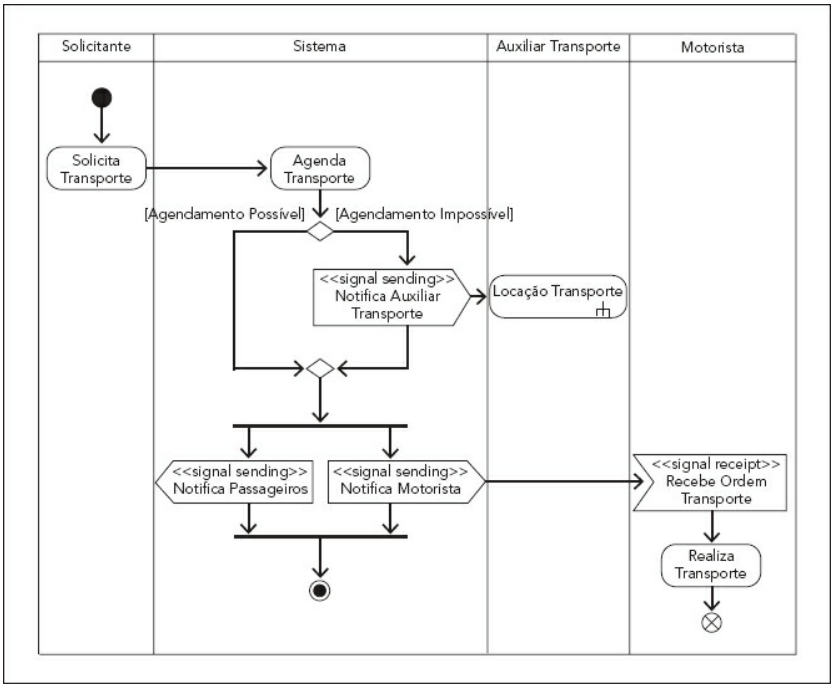


Figura 9 – Diagrama de atividade do processo proposto para solicitação de transporte. Fonte: elaborada pelo autor.

Na medida em que se modelam os processos de negócio, ou mais genericamente, os processos do domínio, o problema a ser resolvido passa a ser compreendido e as necessidades do cliente a ser capturadas. Assim, de certa forma, dá-se início à captura de requisitos do sistema.

É importante perceber que ao modelar o negócio surge a oportunidade de melhorar os processos de uma organização ou domínio em vez de simplesmente

implementá-los como estão. Pode ser que se perceba, inclusive, que os problemas e necessidades do cliente possam ser resolvidos pela aquisição ou integração de outras soluções de software, sem haver a necessidade de se desenvolver nenhuma aplicação de software. Em casos extremos, é possível que a simples modelagem do negócio seja suficiente para atender ao cliente e que nem aquisição, integração e desenvolvimento de software sejam necessários.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar os estudos sobre modelagem de negócios através de modelos de casos de uso de negócio e objetos de negócio recomenda-se a leitura do respectivo capítulo em Kruchten (2001). Para conhecer a BPMN sugere-se consultar os sites www.bpmn.org e www.omg.org/spec/BPMN. A especificação completa do Diagrama de Atividades está disponível em www.omg.org.

RESUMO DO CAPÍTULO

A disciplina de modelagem de negócio tem muita importância para o desenvolvimento de sistemas de informação empresariais, embora seu uso não fique restrito a esses. Sua importância vem da oportunidade de melhorar os processos de uma organização ou domínio em vez de simplesmente implementá-los como estão. Em muitos processos a modelagem de negócio é abordada dentro da disciplina de requisitos. Três artefatos principais são trabalhados nesta atividade: documento de visão, glossário e modelo de negócio. Para realizar a modelagem de negócio pode ser usado o diagrama de atividades da UML ou ainda, modelos na notação BPMN e modelos de casos de uso de negócio e objetos de negócio.

CAPÍTULO 5

REQUISITOS

Este capítulo aborda a disciplina de requisitos. Requisitos são conceituados e categorizados. A Engenharia de Requisitos e suas atividades são apresentadas. Dentre os métodos para se trabalhar com requisitos, este livro foca no de utilização de casos de uso. Assim, os conceitos referentes a casos de uso e modelo de casos de uso são apresentados, bem como o diagrama de casos de uso. Diretrizes e formatos para especificação de casos de uso são explanados e, por fim, comenta-se a respeito de especificações suplementares.

A disciplina de requisitos visa identificar as funções que o sistema deve executar e as restrições sob as quais deve operar. O principal artefato dessa disciplina é o documento de requisitos de software. Às vezes, são produzidos dois documentos: um mais abstrato destinado ao cliente e outro, mais concreto e detalhado, direcionado a quem vai produzir ou fornecer o software. Do ponto de vista do cliente, um requisito é visto como algo que ele necessita. Já, do ponto de vista técnico, é visto como algo que necessita ser projetado.

5.1 Requisitos de Software

Um requisito é uma condição ou capacidade que um sistema deve atender, ou ainda, uma declaração de um serviço ou restrição de um sistema a ser desenvolvido. Requisitos podem ser abordados em três níveis distintos de abstração, cada qual destinado a um público específico:

- requisito de usuário, nível mais elevado, destinado aos clientes e usuários;
- requisito de sistema, nível intermediário, destinado ao cliente e à equipe técnica;
- especificação de projeto de software, nível mais detalhado, destinado exclusivamente à equipe técnica.

5.1.1 Requisito de Usuário

Requisito de usuário é uma declaração de um serviço a ser provido pelo sistema e que diretamente atende a uma necessidade ou solicitação de um usuário. São expressões de alto nível de abstração sobre o comportamento do sistema, também denominadas características do sistema (*system features*). Requisitos de usuário geralmente são escritos em linguagem natural para um público não técnico, podendo valer-se também de diagramas. Um requisito de usuário pode ser expandido em diversos requisitos de sistema.

5.1.2 Requisito de Sistema

Requisitos de sistema estabelecem detalhadamente as funções e as restrições do sistema. São descrições mais detalhadas dos requisitos do usuário, servindo de base para um contrato destinado à implementação do sistema. Às vezes, é chamado de especificação funcional.

A representação dos requisitos do sistema deve ser feita de tal forma que possa servir de veículo de comunicação entre os usuários e os desenvolvedores do software. O uso de linguagem natural não é adequado para descrever requisitos de sistema. Tem-se como alternativas para descrição de requisitos de sistema a linguagem natural estruturada (p. ex.: português estruturado), notações gráficas (p. ex.: diagramas de casos de uso) e especificações matemáticas (p. ex.: métodos formais).

5.1.3 Especificação de Projeto de Software

Especificações de projeto de software traduzem requisitos de sistemas de tal forma que os mesmos possam ser implementados. São descrições abstratas do projeto de software acrescentando mais detalhes à especificação de requisitos do sistema. É um documento orientado à implementação, devendo ser escrito para os engenheiros de software que desenvolverão o sistema.

5.1.4 Requisitos Funcionais, Não funcionais e de Domínio

A expressão FURPS+ é frequentemente utilizada para se referir aos requisitos de sistema. Ela é o acrônimo para Funcionalidade, Usabilidade (*Usability*), Confiabilidade (*Reliability*), Desempenho (*Perfomance*) e Facilidade de Suporte (*Supportability*). O símbolo + refere-se a outros tipos de requisitos, tais como requisitos de segurança, de implementação, de interface, de empacotamento e requisitos legais ou estatutários.

Os requisitos de sistema de software são classificados em:

- funcionais;
- não funcionais;
- de domínio.

5.1.4.1 Requisitos funcionais

Os requisitos funcionais são declarações de funções que o sistema deve fornecer, podendo, eventualmente, declarar explicitamente o que o sistema não deve fazer. Referem-se às funções que o sistema deve ser capaz de realizar. Expressam o comportamento de um sistema.

5.1.4.2 Requisitos não funcionais

Para prover a qualidade desejada ao usuário final, o sistema deve exibir uma vasta variedade de atributos que não são descritos pelos requisitos funcionais. Requisitos não funcionais referem-se às restrições sobre os serviços ou funções oferecidas pelo sistema.

Os requisitos não funcionais não se referem às funções específicas do sistema. Muitos requisitos não funcionais dizem respeito ao sistema como um todo, podendo estar relacionados a propriedades emergentes do mesmo, como, por exemplo, tempo de resposta. A falha em cumprir com um requisito funcional individual pode degradar o sistema, enquanto a falha em cumprir um requisito não funcional pode tornar o sistema inútil.

Requisitos não funcionais são de difícil verificação. Idealmente devem ser expressos quantitativamente, utilizando-se métricas que possam ser objetivamente testadas. Por exemplo, o desempenho pode ser medido por transações/segundo ou tempo de resposta, ao passo que a confiabilidade pode ser medida pela taxa de disponibilidade ou pela taxa de falhas.

Requisitos não funcionais estão muito interligados e é muito frequente que a busca em se cumprir um requisito implique dificuldades para se atingir outro. Por exemplo, para alcançar o desempenho adequado, um algoritmo complexo pode ser implementado ou uma solução proprietária pode ser demandada. Isto, em contrapartida, poderá prejudicar a manutenibilidade e a portabilidade do sistema.

Os requisitos não funcionais nem sempre dizem respeito ao sistema de software a ser desenvolvido, podendo referir-se também a requisitos organizacionais, procedentes de políticas e procedimentos nas organizações do cliente e do desenvolvedor (processo de software, linguagem, padrões de qualidade) e a requisitos externos, procedentes de fatores externos ao sistema e a seu processo de desenvolvimento (questões legais).

5.1.4.3 Requisitos de domínio

Requisitos de domínio são requisitos funcionais e não funcionais derivados do domínio da aplicação do sistema, no lugar de serem obtidos a partir das necessidades específicas dos usuários do sistema. Podem ser novos requisitos funcionais, restringir os requisitos funcionais existentes ou especificar como um requisito funcional deve ser feito.

5.1.5 Documento de Requisitos de Software

O documento de requisitos de software, geralmente chamado de especificação de requisitos de software (SRS – *software requirements specification*), é a declaração oficial do que é exigido dos desenvolvedores de sistema. Deve abranger requisitos de usuário e uma especificação detalhada dos requisitos de sistema.

5.2 Engenharia de Requisitos

Nas últimas décadas diversos estudos demonstraram que a má qualidade dos requisitos é uma das principais causas de fracasso em projetos de software. Como reação, a comunidade científica e profissional de TI focou esforços no estudo dos requisitos de software e concebeu a Engenharia de Requisitos.

A Engenharia de Requisitos é uma subárea da Engenharia de Software que trata do processo de definição dos requisitos de software. O processo clássico de Engenharia de Requisitos apresenta cinco atividades principais: elicitação, análise, especificação, validação e gerenciamento. Pode, ainda, estabelecer como primeira atividade um estudo de viabilidade.

5.2.1 Estudo de Viabilidade

O processo de Engenharia de Requisitos começa com um estudo de viabilidade. Este estudo gera um relatório que recomenda se vale a pena ou não realizar o processo de Engenharia de Requisitos e o processo de desenvolvimento de sistemas.

5.2.2 Elicitação e Análise de Requisitos

Na elicitação busca-se descobrir os requisitos do sistema, normalmente nebulosos e confusos no início do desenvolvimento de um sistema de software. De certa forma, esta atividade está intimamente relacionada à modelagem de negócio. Deve abordar quatro dimensões:

- entendimento do domínio da aplicação;
- entendimento do negócio;
- entendimento do problema;
- entendimento das necessidades e solicitações das pessoas envolvidas.

A atividade de análise de requisitos está muito vinculada à elicitação, pois, na medida em que os requisitos vão sendo descobertos, algum grau de análise sobre os mesmos é realizado. A elicitação e a análise de requisitos envolvem a compreensão do domínio, a coleta, a classificação, a estruturação, a priorização e a validação dos requisitos. O objetivo da análise de requisitos é encontrar possíveis problemas na declaração informal dos requisitos obtidos na elicitação.

Diversas técnicas podem ser utilizadas para a elicitação e análise de requisitos, tais como entrevistas, questionários, reuniões, inspeções, *Joint Application Development* (JAD) e reuniões de *brainstorming*, etnografia e análise de cenários. A obtenção de requisitos com base em cenários pode ser realizada informalmente ou de maneira mais estruturada. Neste caso, destacam-se as técnicas de cenário de eventos, casos de uso e histórias de usuários.

5.2.3 Especificação de Requisitos

A especificação é a atividade na qual os resultados da elicitação e análise de requisitos serão transformados em documentos que organizam os requisitos do sistema. Os requisitos podem ser especificados através de técnicas semiformais, tais como especificação de cenários e prototipação, ou através de métodos formais, como,

por exemplo, utilizando-se a linguagem de especificação formal Z.

5.2.4 Validação de Requisitos

A validação dos requisitos é o processo de verificar os requisitos quanto a sua validade, consistência, completeza, sua viabilidade e sua facilidade de verificação. As revisões de requisitos e a prototipação são as principais técnicas utilizadas para a validação de requisitos. Descobrir um erro durante a validação é muito mais barato que achar o mesmo em fases posteriores do processo de desenvolvimento de software. As seguintes verificações devem ser realizadas:

- de necessidade, que trata de descartar requisitos desnecessários;
- de completude, que investiga a existência de requisitos faltantes;
- de consistência, que busca eliminar ambiguidades e redundâncias;
- de possibilidade, que procura analisar a viabilidade técnica e econômica de um requisito.

5.2.5 Gerenciamento de Requisitos

Mudanças técnicas e relacionadas ao domínio da aplicação são inevitáveis. O gerenciamento de requisitos visa gerenciar e controlar alterações nos requisitos do software para atender tais mudanças e garantir a rastreabilidade dos mesmos, ou seja, o vínculo entre requisitos de usuário, requisitos de sistema e especificações de software. Esta atividade tem maior sentido após a concepção, pois somente no final desta fase é que se tem uma *baseline* de requisitos.

5.2.6 Engenharia de Requisitos no Processo Unificado

Existem diversas técnicas e métodos para executar cada uma das atividades da Engenharia de Requisitos. Contudo, a utilização de casos de uso tem-se demonstrado muito efetiva, pois pode ser utilizada tanto para elicitação de requisitos, quanto para análise e para a especificação.

No Processo Unificado, os requisitos de software são capturados no modelo de casos de uso e no documento de especificações suplementares. No modelo de casos de

uso constarão os requisitos funcionais, ao passo que os requisitos não funcionais serão mapeados para o documento de especificações suplementares. Esses dois artefatos em conjunto formam o documento de especificação de requisitos de software.

5.3 Casos de Uso

Os casos de uso têm sua origem no método *Objectory* (JACOBSON, 1994). Jacobson usava interações para entender requisitos. Os casos de uso surgiram como uma maneira de dar um tratamento mais formal à atividade de capturar e representar essas interações. A definição de caso de uso demanda o entendimento dos conceitos de ator e cenário.

Ator é qualquer entidade externa ao sistema e que interage com este para a execução de uma tarefa. Podem ser pessoas (p. ex.: um usuário ou um operador), outros sistemas, dispositivos (p. ex.: sensores e controladores) ou outras entidades que interagem com o sistema. Cenário é uma interação específica entre um ator e o sistema. A seguir tem-se uma descrição sucinta de um cenário onde o ator, no caso um cliente, realiza uma compra via Internet.

> O cliente vasculha o catálogo de produtos e adiciona os itens desejados à cesta de compras. Quando deseja pagar, ele provê informações para entrega e para pagamento (cartão de crédito) e confirma a compra. O sistema verifica a autorização junto à operadora do cartão e confirma a venda através de uma mensagem.

Caso de uso é uma coleção de cenários relacionados de sucesso e fracasso, que descrevem um ator usando o sistema como meio para atingir um objetivo. Cenário é, portanto, uma instância de um caso de uso. Cenários são também denominados variantes dos casos de uso. Um caso de uso tem um cenário principal que é o cenário de sucesso mais frequente. Este cenário costuma ser apelidado de "dia feliz". No entanto, um caso de uso pode conter muitos cenários alternativos de sucesso, ou até cenários de fracasso onde a intenção do ator acaba por não se realizar da forma pretendida.

Às vezes, é difícil decidir se um determinado conjunto de cenários constitui apenas um ou vários casos de uso. Nesses casos, duas regras podem ser consideradas para a avaliação: um caso de uso corresponde a um único objetivo e um objetivo deve ser plenamente contemplado pelo caso de uso. Por exemplo, em um cenário onde o ator realiza um pagamento em um caixa, as interações referentes à impressão do

cupom fiscal não constituem outro caso de uso, pois a interação somente atende ao objetivo do ator se a impressão do cupom fiscal for realizada. Logo, trata-se de um único caso de uso.

Para fins de priorização e análise, segundo Wazlawick (2004), os casos de uso podem ser organizados em três categorias:

- manutenção de conceitos;
- processos de negócio;
- consultas.

Os casos de uso de manutenção de conceitos contemplam operações elementares de inclusão, alteração, consulta e exclusão de conceitos do domínio de negócio. São também chamados de casos de uso CRUD, do inglês, *Create*, *Retrieve*, *Update and Delete*. Como exemplo, pode-se citar um caso de uso para incluir, alterar, consultar ou excluir um cliente[2].

Os casos de uso de processos de negócio tratam dos principais processos de negócio iniciados ou com a participação dos atores do sistema. É provável que estes processos envolvam muitos conceitos do sistema e que operações elementares de inclusão, alteração, consulta e exclusão sejam demandas para a realização das mesmas. Por exemplo, realizar uma compra é certamente um caso de uso de processo de negócio, pois não pode ser tratado como um simples cadastro de uma compra. Realizar uma compra em um sistema implica, entre outras questões, dar baixa no estoque dos produtos comprados, alterar o valor de uma conta, gerar uma previsão de receita e gerar lançamentos contábeis. Ou seja, muitas outras instâncias de conceitos do sistema são criadas, modificadas, excluídas ou consultadas.

Os casos de uso de consultas referem-se às consultas (impressas ou não) de um sistema de software. Não se trata de uma consulta específica de um conceito (*retrieve*), mas sim, de consultas mais elaboradas que relacionam diversos conceitos, tais como consultas de informações consolidadas, de informações analíticas e de informações estatísticas.

Atores representam papéis de entidades externas interagindo com o sistema. Assim, um papel pode ser desempenhado por muitos usuários e um usuário pode desempenhar papéis distintos. De forma similar, um papel pode executar muitos casos de uso e um caso de uso pode ser executado por muitos papéis.

O ator que obtém valor do caso de uso é chamado de ator principal. Os demais são chamados de atores secundários ou participantes. Os atores podem interagir de

diferentes formas com o sistema e, de acordo com a forma de interação com o sistema, são classificados em quatro categorias:

- iniciador – é aquele que inicia o caso de uso;
- servidor externo – é um ator que provê serviço para o sistema em um caso de uso;
- receptor – é o ator que recebe alguma informação do caso de uso;
- facilitador – é o que dá suporte à interação de outro ator com o sistema.

5.4 Modelo de Casos de Uso

O modelo de casos de uso é o conjunto de todas as especificações de todos os casos de uso do sistema e diagramas associados. As principais atividades para elaborar um modelo de casos de uso do sistema são:

- identificar casos de uso;
- diagramar casos de uso;
- especificar casos de uso.

5.4.1 Identificar Casos de Uso

Para identificar os casos de uso de um sistema deve-se identificar atores e, para cada ator, investigar suas interações com o sistema. É importante produzir uma lista de atores com uma breve descrição das responsabilidades e das características de cada um. A identificação e caracterização destes será útil para a configuração de outros aspectos do sistema, tais como tipos de usuários, segurança de acesso e interface gráfica com usuário. Para identificar os atores envolvidos no sistema pode-se lançar mão das questões relacionadas a seguir.

- Quem usa o sistema?
- Quem mantém ou configura o sistema?
- Quem provê informações para o sistema?

- Quem busca informações no sistema?
- Quais outros sistemas utilizam o sistema?
- Quais outros sistemas são utilizados pelo sistema?

Para identificar os casos de uso deve-se investigar as interações dos atores com o sistema. As seguintes questões podem ser úteis para a identificação dos casos de uso.

- Como cada ator interage com o sistema?
- Quais são as tarefas de cada ator?
- O ator irá criar, armazenar, remover ou ler informações no/do sistema?
- O ator irá informar mudanças externas (eventos) ao sistema?
- O ator precisará ser informado sobre certas ocorrências no sistema (eventos)?

5.4.2 Diagramar Casos de Uso

Na medida em que atores e casos de uso vão sendo identificados, os mesmos podem ser representados em um Diagrama de Casos de Uso para melhor documentar e comunicar as funções que o sistema deve executar. Uma observação importante é que não é necessário diagramar os casos de uso para usá-los. O essencial em um modelo de casos de uso são suas descrições e não os diagramas. O diagrama de casos de uso será visto sem seção posterior.

5.4.3 Especificar Casos de Uso

Na fase de Concepção, casos de uso só precisam ser especificados em alto nível. É nas fases seguintes de elaboração e de construção que os mesmos terão sua especificação completada. Ao término da fase de concepção é importante que cada caso de uso identificado tenha um especificação mínima, contemplando sua identificação e nome (p. ex.: CDU010 – Efetuar Venda), a relação de usuários associados e uma descrição sucinta, contendo o propósito do caso de uso descrito em algumas linhas e fornecendo uma definição da funcionalidade do sistema em alto nível. A seguir é apresentado um exemplo de especificação de caso de uso compatível com a fase de concepção.

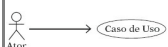
UC10 – Solicitar Transporte
Ator(es): solicitante, Motorista, Passageiro, Gerente de Transporte
Descrição Sucinta: o Solicitante requisita um transporte de materiais ou de passageiros, informando os locais e horários de partida e chegada, o nome dos passageiros ou a especificação do material a ser transportado. O motorista, o(s) passageiro(s), se for o caso, devem ser comunicados do agendamento do transporte. O gerente de transporte deve ser notificado se o agendamento não for viável.

Nas fases de elaboração e construção é feita a especificação detalhada dos casos de uso selecionados para determinada iteração. A especificação detalhada de casos de uso será vista em seção posterior.

5.5 Diagrama de Casos de Uso

O diagrama de casos de uso é um diagrama que mostra um conjunto de casos de uso, atores e seus relacionamentos. Proposto inicialmente por Ivar Jacobson, faz parte da UML desde a sua primeira versão. O Quadro 4 apresenta os principais elementos gráficos da notação de diagrama de casos de uso.

Quadro 4 – Principais elementos gráficos do diagrama de casos de uso

Elemento Gráfico	Descrição
	Ator – podem ser representados todos os atores: principal e secundários. É possível optar também por se representar apenas os atores que iniciam os casos de uso ou apenas os que obtêm valor da execução dos casos de uso. Cada ator recebe um nome.
	Caso de uso – cada caso de uso deve receber um nome. É uma boa prática atribuir uma identificação codificada ao caso de uso. Isto facilita sua rastreabilidade. Outra boa prática é dar nomes aos casos de uso que comecem com um verbo no infinitivo.
	Relacionamento entre ator e caso de uso – é um relacionamento de comunicação que representa a interação de um ator com sistema. O sentido da comunicação pode ser indicado pela presença de uma seta. Pode haver relacionamentos entre atores e entre caso de uso. Estas situações serão vistas posteriormente.
Retângulo englobando os atores e casos de uso	Fronteira do sistema – determina os atores e casos de uso que pertencem ao sistema, separando-os dos demais que por ventura tenham sido modelados.

Fonte: elaborado pelo autor.

A Figura 10 exemplifica a utilização do diagrama de casos de uso. O diagrama

apresenta os atores, os casos de uso e os respectivos relacionamentos obtidos a partir a descrição do sistema exemplo.

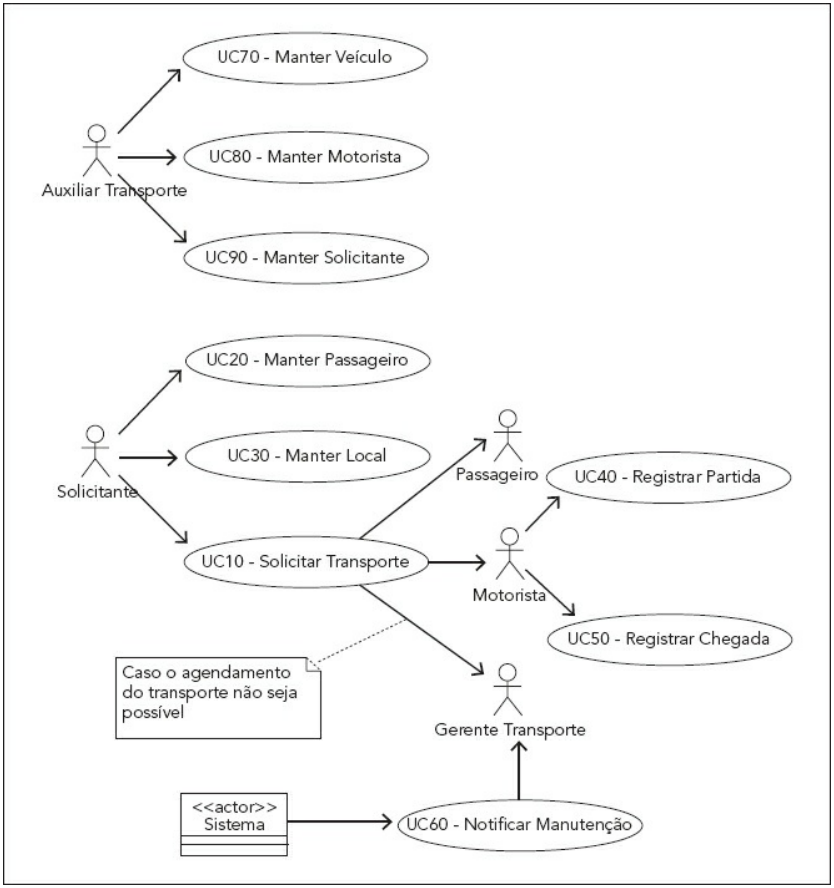


Figura 10 – Diagrama de casos de uso do sistema exemplo.
Fonte: elaborada pelo autor.

5.5.1 Relacionamento entre Casos de Uso e entre Atores

Os primeiros diagramas de caso de uso desenhados durante o desenvolvimento de um sistema geralmente contemplam apenas atores, casos de uso e relacionamentos entre ambos. Isto porque nos momentos iniciais do processo tudo o que se sabe sobre os casos de uso são pequenas descrições. Na medida em que se obtêm mais informações sobre os atores e sobre os casos de uso através de suas especificações detalhadas, relacionamentos entre atores e casos de uso começam a ser identificados. Esses relacionamentos não têm impacto sobre o comportamento ou sobre os requisitos do sistema, sendo simplesmente um mecanismo para melhorar a comunicação, reduzir a redundância e facilitar o gerenciamento.

5.5.1.1 Relacionamento entre casos de uso

Os relacionamentos entre casos de uso e entre atores podem ser de três tipos:

- inclusão;
- extensão;
- generalização/especialização.

Para compreender o relacionamento entre casos de uso é necessário conhecer os conceitos de caso de uso concreto e abstrato e de caso de uso base e de adição. Um caso de uso concreto é iniciado por um ator e realiza o comportamento desejado por ele. Já um caso de uso abstrato é instanciado por outro caso de uso. É um caso de uso de subfunção, que faz parte de outro caso de uso.

Um caso de uso base inclui outro caso de uso, é estendido por outro caso de uso ou é especializado por outro caso de uso. Geralmente são casos de uso concretos. Um caso de uso de adição é uma inclusão, extensão ou especialização de outro caso de uso. Geralmente são abstratos.

5.5.1.2 Relacionamento de inclusão entre casos de uso – «include»

O relacionamento de inclusão é um relacionamento de dependência com o estereótipo «include». Em versões anteriores chamava-se “use”. Representa a fatoração de funcionalidade comum, sendo usado para compartilhar comportamento comum entre casos de uso existentes. Um caso de uso A inclui outro caso de uso B quando uma instância de A inclui o comportamento especificado por B. O caso de uso

A não é completo sem B.

Por exemplo, ao detalhar a especificação do caso de uso "UC70 – Manter Veículo" percebe-se que uma subfunção deste caso de uso é registrar as manutenções do veículo. Seria possível criar um caso de uso específico para esta funcionalidade "UC75 – Manter Manutenções" e informar no diagrama que este faz parte do caso de uso UC70. Neste exemplo, UC70 é um caso de uso concreto, pois é iniciado por um ator, ao passo que o caso de uso UC75 é um caso de uso abstrato. O primeiro é o caso de uso base. O segundo, o de adição. A Figura 11 mostra esta situação.

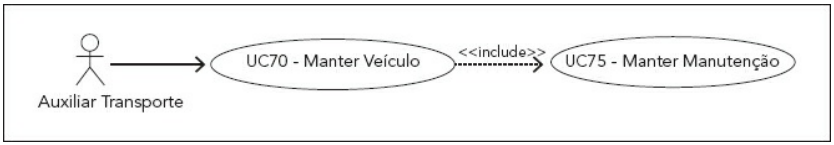


Figura 11 – Exemplo de relacionamento de inclusão.
Fonte: elaborada pelo autor.

Outro exemplo seria o caso em que um cliente pode usar o sistema para realizar três funções: obter extrato, realizar saque e realizar transferência. Em todos esses casos, é necessário que o cliente forneça sua identificação. Em vez de especificar esta funcionalidade três vezes, uma em cada caso de uso, pode-se criar um caso de uso abstrato e relacioná-lo aos três, informando que estes incluem o comportamento do último.

Como diretriz, deve-se usar o relacionamento «include» quando estiver ocorrendo repetição em dois ou mais casos de uso separados, ou quando um caso de uso for muito complexo e longo. Separá-lo em subunidades irá ajudar na compreensão.

5.5.1.3 Relacionamento de extensão – «extend»

O relacionamento de extensão representa a adição de comportamento a um caso de uso já existente. Um caso de uso A estende um caso de uso B quando uma instância de B, em algumas situações, pode ser complementada pelo comportamento especificado por A. O caso de uso B possui um curso completo de transações, ou seja, independe de possíveis casos de uso relacionados via «extend». Pode-se ver um relacionamento «extend» como uma interrupção no caso de uso original B, ponto a partir do qual o caso de uso A será inserido. Esse ponto de interrupção é chamado de

ponto de extensão e pode receber um rótulo para que o caso de uso de adição possa referenciar-se ao caso de uso base. O caso de uso A desconhece se uma interrupção em B irá ocorrer ou não.

Por exemplo, na descrição do sistema exemplo é dito que "caso um passageiro ou um local não estejam cadastrados, o solicitante poderá cadastrá-los durante o processo de preenchimento da solicitação". Nesta situação, o "UC10 – Solicitar Transporte", que é completo por si só, pode ter seu comportamento estendido por outros dois casos de uso, o "UC20 – Manter Passageiro" e o "UC30 – Manter Local". Pode-se diagramar esta situação conforme mostra a Figura 12. Os três casos de uso são concretos, mas o UC10 é base, enquanto os outros dois são casos de uso de adição.

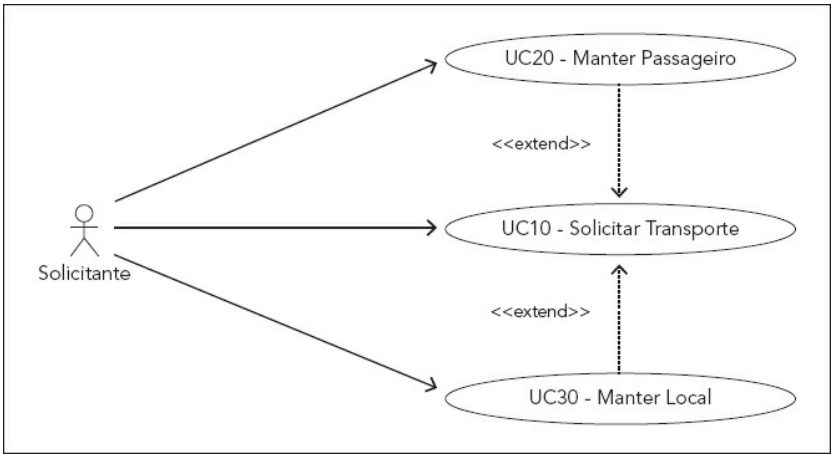


Figura 12 – Exemplo de relacionamento de extensão.
Fonte: elaborada pelo autor.

Como diretriz, deve-se usar o relacionamento «extend» ao se modelar comportamento condicional, opcional ou adicional, ao descrever o tratamento de um evento assíncrono, como quando um usuário pode, a qualquer momento, selecionar ou desviar para uma janela ou página ou quando for indesejável modificar um caso de uso base.

A UML define que relacionamentos de dependência são representados por uma linha tracejada e com uma seta indicando o sentido da dependência. Nos casos de inclusão, o sentido do relacionamento é do caso de uso base para o de adição, pois o

primeiro depende do segundo, visto que sem este o caso de uso base não pode ser considerado completo. Nos casos de extensão, o sentido do relacionamento é do caso de uso de adição para o caso de uso base. Embora ambos possam ser independentes e completos, o sentido da dependência indica que o caso de uso de adição depende do caso de uso base para, em uma determinada situação, ser instanciado.

5.5.1.4 Relacionamento de generalização/especialização

Um caso de uso especializado B pode sobrescrever ou especializar o comportamento do caso de uso base A. Na UML, relacionamentos de generalização/especialização são representados por uma linha contínua com um triângulo em uma das pontas. O elemento tocado pelo triângulo é o mais geral. Os demais são os especializados.

No diagrama da Figura 13 informa-se que existem dois tipos de solicitação de transporte: o de material e o de pessoas. Indubitavelmente, ambos são tipos distintos de solicitação de transporte. Se, além disso, o fluxo principal de cada um deles for muito distinto um do outro, vale a pena tratá-los como casos especiais, ou especializações de um caso de uso mais geral, o caso de uso "UC10 – Solicitar Transporte". Caso contrário, poderia tratar-se de um caso de uso com dois fluxos alternativos. A representação hierárquica neste diagrama permite mostrar que apenas no caso de uso UC12 haverá notificação de passageiros. No diagrama completo mostrado na Figura 10, esta distinção não era possível.

Às vezes, é fundamental representar casos de uso especializados, pois alguns atores poderão estar relacionados a determinadas especializações de casos de uso e outros atores a outras. Isto fornece subsídios importantes para, por exemplo, o mapeamento da segurança de acesso do sistema: quem pode fazer o quê no sistema. O caso de uso UC10 é concreto e base, ao passo que os casos de uso UC14 e UC12 são abstratos e de adição.

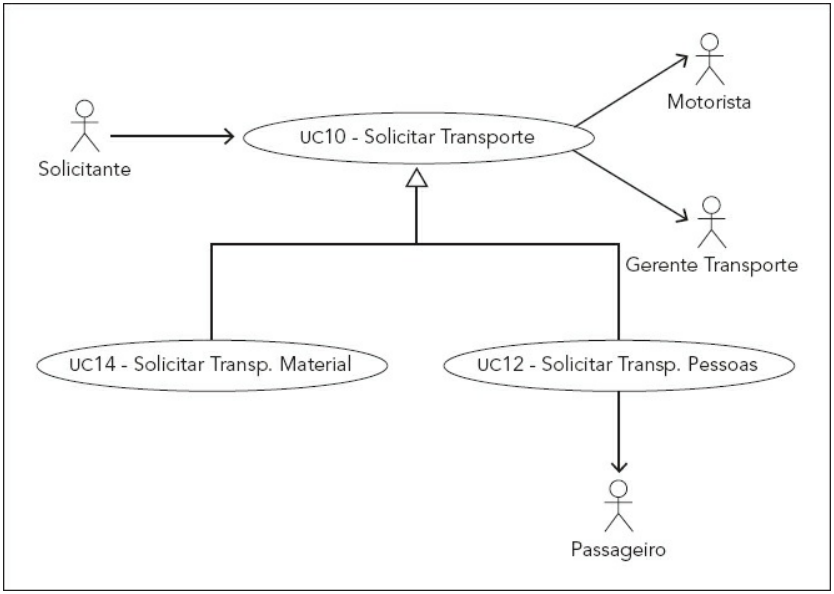


Figura 13 – Exemplo de generalização/especialização de casos de uso.
Fonte: elaborada pelo autor.

Como diretriz, um caso de uso pode ser tratado como especialização de caso de uso mais geral quando tiver um fluxo alternativo suficientemente distinto a ponto de poder ser tratado como outro caso de uso.

Às vezes, é difícil definir se o relacionamento é de inclusão ou de extensão, ou se um caso de uso deve ser especializado ou não. Geralmente, não vale a pena agonizar sobre relacionamentos entre casos de uso.

5.5.1.5 Relacionamento entre atores

O relacionamento entre atores pode ser de dois tipos: comunicação e generalização/especialização. O primeiro trata de situações em que dois atores trocam informação entre si. Como atores são elementos externos ao sistema, modelar ou não este tipo de relacionamento não traz consequência alguma ao sistema. Entretanto, isto pode ajudar a compreender e comunicar melhor o domínio que está sendo modelado.

Já o relacionamento de generalização/especialização entre atores pode ser usado para melhorar a caracterização de atores, informando que determinado ator herda os relacionamentos de outro. Isto pode tornar a leitura do diagrama mais simples na medida em que pode diminuir a quantidade de relacionamentos desenhados entre atores e casos de uso.

Na Figura 14, o diagrama comunica que um Cliente pode fazer uma aplicação (CDU01). O Cliente Vip também poderá fazer uma aplicação, visto que é um tipo de cliente e, portanto, herda os relacionamentos de Cliente. O Cliente Vip também pode fazer o resgate total do saldo diretamente no sistema. Já se o Cliente deseja resgatar o saldo total, não o poderá fazer usando o sistema. Para tal, terá de entrar em contato com o Atendente, e este usará o sistema para fazer o resgate.

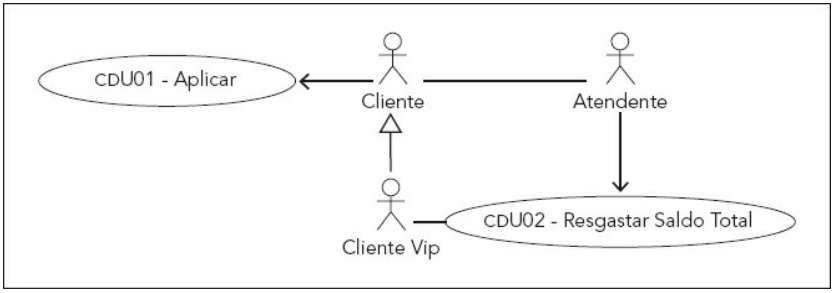


Figura 14 – Exemplo de relacionamento entre atores.
Fonte: elaborada pelo autor.

Como diretriz, os relacionamentos entre atores somente devem ser representados se melhorarem a comunicação e o entendimento do sistema.

5.6 Especificação de Casos de Uso

Uma especificação detalhada de caso de uso descreve, entre outros elementos, a interação entre ator e sistema representando:

- o fluxo principal;
- os fluxos alternativos;
- as exceções e os respectivos fluxos de tratamento.

5.6.1 Especificação de Fluxos e Passos

Fluxos são sequências de passos em uma interação de um ator com o sistema. Exceção é um evento que ocorre na execução de um fluxo e que impede o progresso do mesmo caso não seja tratado. Para garantia da continuidade do caso de uso, toda exceção deve ser tratada.

O fluxo principal é a sequência de passos que representa o principal e mais usual cenário de sucesso de um caso de uso. É também chamado de cenário principal, fluxo normal, fluxo básico, ou "dia feliz". Fluxos alternativos são sequências de passos que representam os demais cenários de sucesso de um caso de uso. Fluxo de tratamento de exceções são sequências de passos para tratamento de exceções nos diversos cenários. A Figura 15 demonstra esses conceitos.

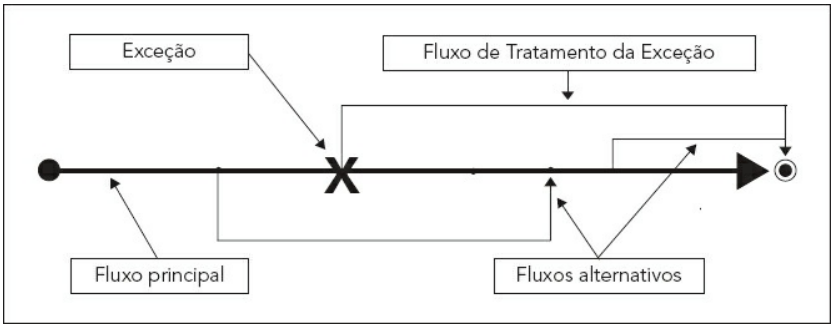


Figura 15 – Fluxos em um caso de uso.
Fonte: elaborada pelo autor.

Para fazer a especificação detalhada de um caso de uso, descreve-se a interação entre ator e sistema, passo a passo, representando o fluxo principal e os fluxos alternativos. Analisa-se cada passo, procurando identificar o que pode dar errado (as exceções) e o que fazer nesses casos (os fluxos de tratamento das exceções).

A seguir é apresentada a descrição do fluxo principal de um caso de uso para processamento de venda em um caixa (PDV). Observa-se que as interações são representadas de forma sequencial e que os passos são numerados. Deve-se evitar o uso de lógica condicional, do tipo: se-então-senão. Essas possibilidades são representadas nos casos de uso através de fluxos alternativos. A numeração dos passos é importante, pois ela serve de referência para a descrição dos fluxos

alternativos e de exceção.

UC01 – Processar Venda

1. Cliente chega à saída do PDV com bens ou serviços para adquirir.
2. Caixa começa uma nova venda.
3. Caixa insere o identificador do item.
4. Sistema registra a linha de item da venda e apresenta uma descrição do item, seu preço e total parcial da venda.
   Caixa repete passos 3 e 4 até que indique ter terminado.
5. Sistema apresenta o total com impostos calculados.
6. Caixa informa total ao Cliente e solicita pagamento.
7. Cliente paga e o sistema trata pagamento.
8. Sistema registra venda completada e envia informações de venda e pagamento para Sistema de Contabilidade e Sistema de Estoque.
9. Sistema emite recibo.
10. Cliente vai embora com recibo e mercadorias.

Neste caso de uso, quatro alternativas para efetuar o pagamento são possíveis:

- em dinheiro;
- com cheque;
- com cartão de crédito;
- com cartão de débito.

Na descrição detalhada de casos de uso, devem-se considerar apenas exceções do domínio ou de negócio, ou seja, não devem ser mencionadas exceções de cunho computacional, como, por exemplo, Null Pointer Exception. No caso de uso "UC01 – Processar Venda", poderiam ser consideradas exceções, entre outras:

- identificador de item inválido;
- cartão de crédito inválido;
- cartão de crédito com saldo insuficiente;
- bobina sem papel.

5.6.1.1 Recomendações

A linguagem utilizada ao descrever os passos de casos de uso deve ser uma linguagem essencial, ou seja, sem considerar aspectos tecnológicos. Por exemplo, as expressões "cliente se identifica" e "usuário escolhe a opção" são preferíveis a, respectivamente, "cliente passa o cartão magnético" e "sistema mostra as opções na tela e o usuário pressiona um botão". Mencionar aspectos de tecnologia em casos de uso pode limitar as possibilidades de projeto ao antecipar decisões técnicas que seriam avaliadas em momento mais oportuno. Portanto, a menos que determinadas soluções técnicas sejam restrições do projeto ou requisitos não funcionais, a descrição dos casos de uso devem evitar mencioná-las.

Deve-se evitar escrever passos que dizem respeito a procedimentos internos do sistema, tais como "o sistema gera um comando SQL …". Ao trabalhar com casos de uso na atividade de requisitos, se está interessado em explorar a interface do sistema com os atores e não no que acontece dentro do sistema.

Os passos podem ser considerados obrigatórios ou complementares. Passos obrigatórios são aqueles em que informação é passada dos atores para o sistema e do sistema para os atores. Podem ser de dois tipos: eventos de sistema, passos em que o ator envia alguma informação para o sistema, e respostas do sistema, passos em que o sistema envia alguma informação para o ator. Por exemplo, considerando o caso de uso "UC01 – Processar Venda", os passos "3 – Caixa insere o identificador do item" e "5 – Sistema apresenta o total com impostos calculados" são obrigatórios, sendo o primeiro considerado um evento de sistema e o segundo uma resposta do sistema. Todo passo que prover informação ao sistema ou receber informação do mesmo deve ser representado. São esses os dados e informações com os quais o sistema terá de lidar. Se não forem identificados e devidamente tratados, muito possivelmente o software que está sendo projetado não cumprirá com seus objetivos.

Passos complementares não são obrigatórios, mas ajudam a entender o contexto dos passos dos casos de uso. Geralmente representam interações entre atores ou esclarecem quando e como o caso de uso inicia e termina. A seguir, alguns exemplos de passos complementares:

- funcionário pergunta o nome do cliente;
- o caso de uso se inicia quando seleciona a opção sacar;
- cliente chega à saída do PDV com bens ou serviços para adquirir;

- cliente vai embora com o recibo e as mercadorias;
- sistema informa que a reserva foi concluída com sucesso.

5.6.1.2 Ligando fluxos alternativos e fluxos de exceções ao fluxo principal

Fluxos alternativos e de tratamento de exceções são representados como extensões ao fluxo principal. As extensões costumam ser representadas separadamente do fluxo principal e, por isso, é necessário conectá-las ao fluxo principal. A descrição das extensões deve possuir os seguintes elementos:

- um identificador que associe a extensão ao seu passo de origem, como, por exemplo, 1a, 1b, 2a;
- descrição/nome da exceção ou fluxo alternativo, como, por exemplo, "ID do Item inválido!";
- passos a serem executados para tratar a extensão;
- passo de finalização, no qual se indica em qual passo do fluxo principal o caso de uso deve continuar após a execução do último passo da extensão. As possibilidades são: retornar ao passo que gerou a extensão, ir para um passo posterior ou outro qualquer e abortar ou reiniciar o caso de uso.

5.6.2 Formatos de Especificação

Existem vários formatos para especificar casos de uso, não havendo nenhum padrão a esse respeito. Isto permite a criação de estilos próprios de descrição. Entretanto, o formato sugerido por Cockburn (2005) tem-se destacado. Para exemplificar a notação de Cockburn, o Quadro 5 apresenta uma descrição esquematizada de um caso de uso hipotético com um fluxo principal de cinco passos, no qual o passo 3, que se refere a uma ação de pagamento, possui três fluxos alternativos.

Há extensões ao passo 2, identificadas, respectivamente, como 2a e 2b. A extensão 2a é uma exceção denominada "Uma exceção ao passo 2!". Ela possui quatro passos de tratamento, sendo que o passo 2 invoca a execução de outro caso de uso. Na extensão 2b, evidencia-se o último passo, no qual é informado que, após a execução do

fluxo de exceção, o fluxo do caso de uso seguirá no passo 4 do fluxo principal.

As extensões 3a, 3b e 3c são os fluxos alternativos para, respectivamente, pagamento em cheque, com cartão e em dinheiro. Observa-se que na extensão 3b é representada uma extensão ocorrendo dentro de outra. Este recurso, entretanto, deve ser usado com parcimônia.

Duas outras características da notação são apresentadas. Quando se deseja expressar que uma extensão pode ocorrer em qualquer passo, o caractere * deve ser utilizado no lugar do número do passo, já que não há um passo ao qual a extensão possa ser atribuída. No exemplo, tem-se a extensão *a, que pode ocorrer a qualquer momento e culmina com o encerramento do caso de uso, como indica o seu último passo. É possível também especificar uma extensão que pode ocorrer em um intervalo de passos, como mostra a última extensão do exemplo, 3-4a.

Quadro 5 – Formato de Cockburn para descrição de casos de uso

Fluxo principal
1. <passo inicial>
2. …
3. Cliente realiza o pagamento
4. …
5. <passo final>

Extensões
*a. Uma exceção que pode ocorrer a qualquer momento!
1. …
2. …
3. O caso de uso encerra

2a. Uma exceção ao passo 2!
1. …
2. Invoca UC02 – Procurar Produto
3. …
4. …

2b. Outra exceção ao passo 2!
1. …
2. O caso de uso retorna no passo 4

3a. Pagamento em cheque
1. …
2. …
3. …

3b. Pagamento com cartão
1. …
2. …

2a. Uma exceção referente ao passo 2 da extensão 3b!
1. …
2. …
3. …
3. …
4. …

3c. Pagamento em dinheiro
1. …
2. …
3. …

3-4a. Uma exceção que pode ocorrer entre os passos 3 e 4 do fluxo principal
1. …
2. …

Fonte: elaborado pelo autor.

5.6.3 Especificação de Outros Elementos

Pode-se descrever no documento de especificação do caso de uso, além dos passos dos diversos fluxos, outras características, quais sejam:

- atores;
- interessados;
- precondições;
- pós-condições;
- requisitos correlacionados;
- opções tecnológicas;
- protótipos de interface gráfica de usuário;
- questões em aberto.

Além de mencionar o ator, pode-se escrever a respeito de outros interessados no caso de uso e sobre quais são seus interesses. A propensão a tomar decisões acertadas

aumenta quando se sabe não apenas o que precisa ser feito, mas sim, o porquê deve ser feito. Por exemplo, no caso de uso "UC01 – Processar Vendas" é possível ter o seguinte ator e os seguintes interessados:

> Ator principal
> Caixa: deseja entrada de pagamento rápida, precisa e sem erros, pois a falta de dinheiro na gaveta será deduzida do seu salário.
>
> Interessados
> Cliente: deseja comprar e receber um serviço rápido. Deseja exibição dos itens e preços inseridos. Deseja um comprovante de depósito.
> Gerente: deseja poder realizar rapidamente operações de correção no caixa.

A especificação de precondições e pós-condições do caso de uso será de grande valia no projeto do software e na validação do mesmo. As precondições são condições do sistema consideradas verdadeiras antes do início do caso de uso. Indicarão o que deve ser testado antes de o caso de uso iniciar a sua execução. O caso de uso não inicia se uma das precondições for falsa. As pós-condições são condições do sistema após a execução do caso de uso. As pós-condições auxiliarão o projeto dos testes. As condições devem poder ser testadas computacionalmente. Por exemplo, a condição "o usuário está de posse da documentação necessária", não é testável. Considerando-se o caso de uso "UC01 – Processar Vendas", as pré e pós-condições seriam:

> Precondição
> Caixa está identificado e autenticado.
>
> Pós-condições
> Venda foi registrada.
> Impostos foram corretamente calculados.
> Contabilidade e Estoque foram atualizados.
> Recibo foi gerado.
> Autorizações de pagamento com cartão foram registradas.

Requisitos não funcionais correlacionados ao caso de uso podem ser descritos no documento de especificação do caso de uso em que ocorrem. Nessas situações, deve-se cuidar para manter a consistência da especificação do sistema, pois um mesmo requisito não funcional pode estar em mais de um caso de uso. A alteração em um pode não repercutir no outro se não existir um mecanismo de rastreabilidade adequado. Um exemplo de requisito não funcional relacionado ao UC01 – Processar Vendas seria "Resposta de autorização de crédito em até 30" em 90% dos casos".

Deve-se primar por uma descrição essencial. Entretanto, para não perder oportunidades de registrar alternativas tecnológicas, estas podem ser registradas em

seção à parte no documento de especificação do caso de uso. Protótipos de interface de usuário também podem ser anexados ao caso de uso. Uma forma usual de representar protótipos de interface de usuário é através de *wireframes*, em que apenas os contornos dos elementos gráficos da interface e textos são representados. Questões em aberto a respeito do caso de uso, que estão sendo discutidas ou aguardam momento oportuno para discussão, podem ser mantidas junto à descrição do mesmo.

5.6.3.1 Exemplo de descrição de caso de uso

O Quadro 6 apresenta a especificação do caso de uso UC10 – Solicitar Transporte no formato proposto por Cockburn.

Quadro 6 – Especificação do caso de uso UC10 – Solicitar Transporte

Caso de Uso: UC10 – Solicitar Transporte

Descrição: um solicitante requisita um transporte de passageiros ou, alternativamente, um transporte d carga.

Atores: solicitante, Gerente de Transporte, Passageiro, Motorista.

Precondições: o solicitante está autenticado e autorizado.

Pós-condições
Em caso de sucesso
O sistema agendou uma solicitação, reservando um veículo e um motorista para o transporte.
O sistema emitiu uma ordem de transporte para o motorista alocado ao transporte. No caso d transporte de passageiros, os mesmos foram notificados por e-mail.
Em caso de insucesso
O sistema notificou o Gerente de Transporte sobre a impossibilidade de agendar uma solicitação.

Fluxo normal

1. O caso de uso se inicia quando o solicitante requisita um transporte.
2. O sistema apresenta uma lista de locais de destino previamente cadastrados.
3. O solicitante escolhe um local de destino.
4. O solicitante informa a hora desejada para chegada.
5. O sistema apresenta uma lista de locais de partida previamente cadastrados.
6. O solicitante escolhe um local de partida.
7. O solicitante informa a hora da saída.
8. O sistema apresenta uma lista de passageiros previamente cadastrados.
9. O solicitante seleciona os passageiros.
10. O solicitante confirma a solicitação.
11. O sistema agenda a solicitação, emitindo uma ordem de transporte para o motorista alocado e,

no caso de transporte de passageiros, um e-mail para cada um deles.
12. O caso de uso se encerra.

Extensões
3a. O local de destino desejado não está cadastrado!
1. O solicitante informa os dados do novo local (UC30 – Manter Local).
6a. O local de partida desejado não está cadastrado!
1. O solicitante informa os dados do novo local (UC30 – Manter Local)
8a Transporte de Carga
1. O solicitante informa os dados da carga a ser transportada.
2. Retorna ao fluxo principal no passo 10.
9a. O passageiro não está cadastrado!
1. O solicitante cadastra um novo passageiro (UC20 – Manter Passageiro)
11a. O sistema não pôde agendar o transporte!

1. O sistema informa que não pôde agendar o transporte.
2. O sistema envia um e-mail para o Gerente de Transporte informando sobre a solicitação pendente.
3. O caso de uso se encerra.

Fonte: elaborado pelo autor.

5.7 Especificações Suplementares

Os requisitos que não foram descritos nos casos de uso podem ser registrados em um documento de especificações suplementares. Pode conter os seguintes tópicos, sem restringir-se a esses:

- atores;
- funcionalidades genéricas;
- requisitos de qualidade (não funcionais);
- regras de domínio.

A especificação dos atores pode contemplar o nome no ator, a descrição do ator, suas principais responsabilidades e a quantidade de usuários que executam tal papel. Por funcionalidade genérica entende-se aquela que é comum a muitos casos de uso ou, eventualmente, não pertence a nenhum deles, como, por exemplo, o registro e tratamento de erros. Os requisitos de qualidade referem-se aos requisitos não funcionais, tais como usabilidade, confiabilidade, desempenho, manutenibilidade e outros. Regras de domínio comuns a vários casos de uso devem ser especificadas. Elas podem ser específicas do sistema ou de um domínio em particular. Por exemplo, na

especificação de um software para simulação de equipamento mecânicos, regras da física podem ser descritas. No caso de um sistema de informação empresarial, as regras de negócio devem ser especificadas.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar o conhecimento sobre engenharia de requisitos sugere-se a leitura de Kotonya & Sommerville (1998). Para qualificar a especificação de casos de uso recomenda-se a leitura de Cockburn (2005). Mais informações sobre a disciplina de requisitos podem ser obtidas em Kruchten (2001).

RESUMO DO CAPÍTULO

Um requisito é uma condição ou capacidade que um sistema deve atender, ou ainda, uma declaração de um serviço ou restrição de um sistema a ser desenvolvido. Essas declarações podem ser destinadas ao usuário final (requisitos de usuário), aos desenvolvedores (especificações de projeto) ou a ambos, como se fosse um contrato (requisitos de sistema). Requisitos de sistema podem ser funcionais ou não funcionais, conforme se refiram a funções que o software deve prover ou a restrições impostas sobre como prover essas funções. A má qualidade dos requisitos tem sido uma das principais causas de fracasso em projetos de software. A Engenharia de Requisitos visa qualificar esse processo. Para tal, sugere as seguintes atividades: elicitação, análise, especificação, validação e gerenciamento de requisitos. O PU sugere o tratamento dos requisitos funcionais através de casos de uso. No PU, o modelo de casos de uso, constituído da especificação dos casos e de diagramas de casos de uso, e o documento de especificações suplementares conterão os requisitos de sistema elicitados, analisados, especificados e validados. O formato de Cockburn tem-se destacado dentre os diversos formatos para especificação de casos de uso.

CAPÍTULO 6

ANÁLISE – MODELAGEM CONCEITUAL

Este capítulo aborda a disciplina de análise, enfocando na modelagem conceitual. Primeiramente, introduz-se a modelagem orientada a objetos. Na sequência, o diagrama de classes da UML é estudado. Por fim, mostra-se como produzir um modelo de domínio e representá-lo através de um diagrama de classes.

Em muitos processos a análise e o projeto são tomados como uma disciplina única. Em processos de outros modelos e em métodos mais antigos, a distinção entre a análise e projeto era mais bem demarcada. Atualmente, os métodos estabelecem uma maneira contínua de trabalhar com a análise e o projeto. Isto faz com que a fronteira entre essas atividades não seja tão clara, o mesmo ocorrendo, muitas vezes, entre a atividade de requisitos e a de análise.

Embora possam ocorrer de maneira contínua em um processo, análise e projeto são atividades essencialmente distintas. Análise é a atividade de decompor o todo (um sistema) em suas partes componentes e examinar estas partes visando conhecer sua natureza, funções e relações. O objetivo é definir e compreender o problema a ser resolvido. Já projeto, é a atividade de adaptar os resultados da análise às restrições impostas pelos requisitos não funcionais e pelo ambiente de implementação visando à implementação do software. O objetivo é propor uma solução para o problema definido na análise.

As principais atividades da análise são a construção do modelo de domínio e a elaboração dos contratos das operações de sistema. A primeira pertence ao escopo do que costuma chamar-se de modelagem conceitual ou análise estática. A segunda refere-se à modelagem funcional ou análise dinâmica.

A modelagem conceitual trata de representar as principais características estáticas de uma entidade de interesse. Em outras palavras, procura modelar os conceitos do domínio em análise, no caso, um sistema. É por esta razão que é também chamada de modelagem de domínio e o artefato nela produzido de modelo de domínio ou de modelo conceitual.

A modelagem funcional trata de representar as principais características dinâmicas de uma entidade de interesse. Na modelagem funcional, busca-se modelar o

comportamento do sistema, identificando e descrevendo as operações que o sistema tem de realizar para executar suas funções.

Para construir o modelo de domínio é preciso adotar uma abordagem específica e utilizar um padrão para diagramação. A abordagem utilizada neste livro é a da orientação a objetos e o diagrama usado é o diagrama de classes. Já para modelar as operações de sistema será empregado o diagrama de sequência.

Um modelo é uma descrição das características estáticas e/ou dinâmicas de uma entidade de interesse, retratada de diversas visões, normalmente por meio de diagramas ou texto. Um modelo apresenta os elementos essenciais da entidade sob uma determinada perspectiva, abstraindo os elementos não relevantes para aquela perspectiva.

6.1 Modelagem Orientada a Objetos

A modelagem de sistemas foi influenciada pelos paradigmas vigentes em cada época. Nos anos 1970, o paradigma vigente era o da orientação a funções, havendo predominância dos métodos estruturados, como, por exemplo, a programação estruturada, projeto estruturado e análise estruturada. Estes métodos perduraram durante as décadas de 1980 e 1990, e ainda hoje são eventualmente utilizados. Nos anos 1980, o paradigma predominante era o da orientação a dados, tendo sido impulsionado pela difusão da Engenharia da Informação, pelo surgimento de bancos de dados comerciais, de linguagens de 4ª geração e de ferramentas case que possibilitavam o rápido desenvolvimento de uma aplicação a partir de uma base de dados estável. Nos anos 1990,[3] dados e funções passaram a ser vistos como um componente único. O paradigma da orientação a objetos foi adotado em larga escala e isto influenciou a maneira de pensar sobre a modelagem de sistemas. Atualmente, a modelagem de sistemas é feita pensando-se em objetos e classes.

O paradigma da orientação a objetos vê o mundo como formado por objetos. Um objeto é qualquer coisa, real ou abstrata, que possui

- uma identidade;
- um estado;
- um comportamento.

No paradigma da orientação a objetos, um sistema de software é modelado como

uma coleção de objetos interconectados que colaboram entre si executando tarefas específicas ou solicitando a execução de tarefas por outros objetos, de forma que as funcionalidades do sistema sejam cumpridas. Um conceito que surge a partir da existência de objetos é o de classe. Classe é uma abstração de objetos descritos pelos mesmos dados e que possuem o mesmo comportamento.

Ao mapear elementos do paradigma de orientação a objetos para o mundo computacional tem-se:

- o objeto como uma variável complexa, composta de uma estrutura de dados e de métodos que os manipulam;
- a classe como um tipo abstrato de dados, um gabarito para criação de objetos.

Para modelar objetos e classes, juntamente com suas estruturas, comportamentos e relacionamentos, a UML dispõe do diagrama de classes. A estrutura é dada pelos atributos de uma classe, ao passo que o comportamento é definido através de operações.

6.2 Diagrama de Classes

O Diagrama de classes é usado para modelar objetos e classes. Descreve os tipos de objetos do sistema, sua estrutura e os relacionamentos estáticos existentes entre eles.

6.2.1 Classe

Uma classe é representada por um retângulo com três compartimentos: nome da classe, atributos e operações. Opcionalmente, pode-se omitir a visualização dos compartimentos de atributos ou de operações. A Figura 16 exibe a notação da UML para classes.

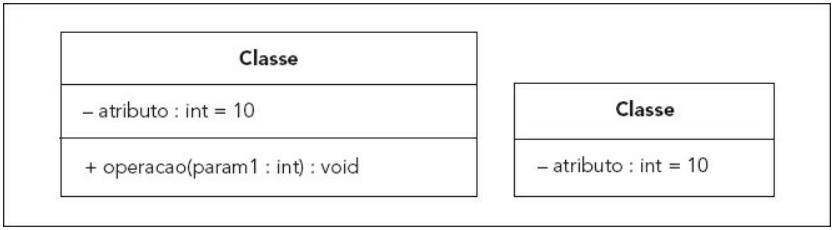


Figura 16 – Notação de classe na UML.
Fonte: elaborada pelo autor.

No primeiro compartimento coloca-se o nome da classe, o qual deve estar no singular e com a letra inicial em maiúscula. O segundo compartimento contém a lista de atributos da classe. Atributo é cada uma das propriedades que descrevem um objeto de uma classe. A notação para um atributo é:

<visibilidade> <nome do atributo> : <tipo> = <valor inicial>

O nome do atributo deve sempre iniciar com letra minúscula. A visibilidade pode ser pública, privada, protegida ou de pacote. Cada tipo de visibilidade é representado por um caractere específico, conforme descrito a seguir:

- + pública – o atributo é visível por qualquer outro objeto;
- – privada – o atributo é visível apenas pelo próprio objeto;
- # protegida – o atributo é visível apenas por objetos de subclasses da classe na qual foi definido;
- ~ pacote – o atributo é visível apenas pelos objetos do pacote.

O tipo do atributo pode ser um tipo básico (p. ex.: int, boolean, char) ou tipos definidos por classes. Nestes casos, podem ser classes de uma linguagem de programação (p. ex.: Integer, Boolean, String) ou classes de tipo de dados do sistema. Tipos de dados do sistema são tipos abstratos de dados do domínio que definem os valores que um atributo pode assumir e as operações válidas para estes valores. Como exemplo, pode-se citar os seguintes tipos de dados: CPF, estado e Sexo. Um valor inicial pode ser definido para o atributo. Quando o objeto for criado, o atributo receberá o valor inicial definido.

Duas características podem ser declaradas com o atributo: derivação e restrição. Um atributo derivado é aquele que pode ser obtido a partir de outros atributos da classe. Sua notação requer uma barra à frente do nome do atributo. Pode-se adicionar também uma restrição ao atributo. As restrições especificam limitações à semântica natural de um elemento, sendo escritas entre chaves. A Figura 17 exemplifica esta situação, estabelecendo que o atributo idade pode ser obtido a partir de outro atributo, no caso o atributo dtNasc. A restrição determina o método de cálculo da idade. Para a atividade de projeto, isto pode sugerir que o atributo idade não seja persistente, isto é, não é armazenado em um banco de dados ou arquivo. Sugere, também, que não se faz necessário projetar uma operação para atribuir valor à idade, visto que esta sempre será computada a partir da data de nascimento. Pensando-se em implementação, não haverá uma operação para atribuir valor à idade.

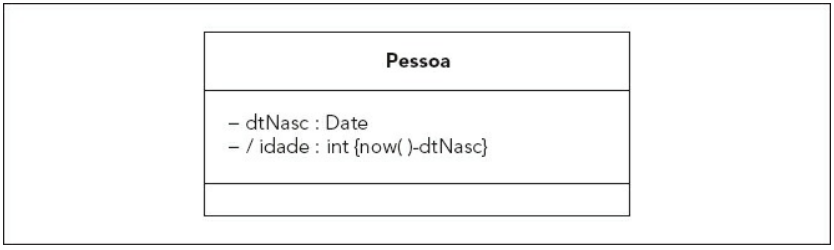


Figura 17 – Atributo derivado e com restrição.
Fonte: elaborada pelo autor.

O terceiro compartimento contém a lista de operações da classe. Uma operação é uma ação executada por um objeto em resposta ao recebimento de uma mensagem. Aqui vale uma observação. Muitos chamam operações de classe de métodos de classe. Esta denominação, embora usual, não é a mais correta. Operação (*operation, method call, method declaration*) é um serviço que pode ser solicitado por qualquer objeto da classe. Método (*method ou method body*) é a implementação de uma operação. Método refere-se ao algoritmo utilizado para realizar uma operação. Uma operação pode ser realizada por meio de vários métodos distintos. Por exemplo, uma operação definida em uma superclasse com três subclasses pode ter três métodos distintos. A notação para uma operação é:

<visibilidade> <nome da operação> (<parâmetros>) : <tipo de retorno>

O nome da operação deve iniciar com letra minúscula e a visibilidade segue o mesmo padrão da visibilidade de atributos. O tipo de retorno pode ser qualquer um dos tipos de dados do modelo, sejam eles tipos básicos ou de classe. Uma operação pode conter uma lista de parâmetros. A Figura 18 exemplifica a notação para operações de classe.

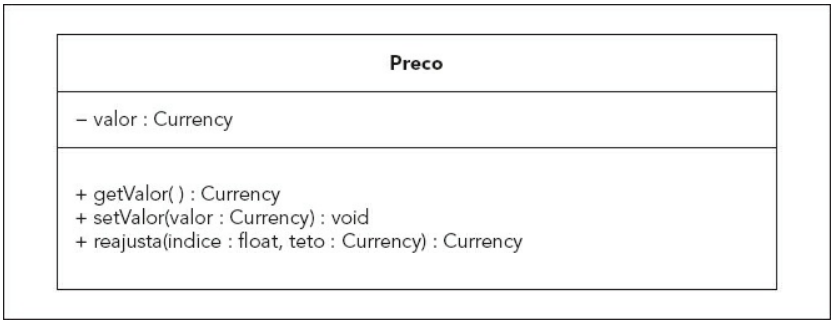


Figura 18 – Notação para operações de classe.
Fonte: elaborada pelo autor.

6.2.2 Objeto

Um objeto é representado por um retângulo com dois compartimentos: nome do objeto e atributos, sendo que o compartimento de atributos pode ser omitido. Existem três formas de designar um objeto: uma, onde se declaram o nome do objeto e o nome da classe a qual o objeto pertence; outra, onde se declara apenas o nome do objeto, sem informar a classe a qual o mesmo pertence e, por fim, a terceira forma, na qual se declara apenas a classe do objeto, sem se informar um objeto específico. Independentemente da forma utilizada, o nome do objeto e/ou da classe vêm sempre sublinhados. No compartimento de atributos, pode-se especificar o valor de cada atributo para cada objeto. A Figura 19 traz exemplos de representações de objetos.

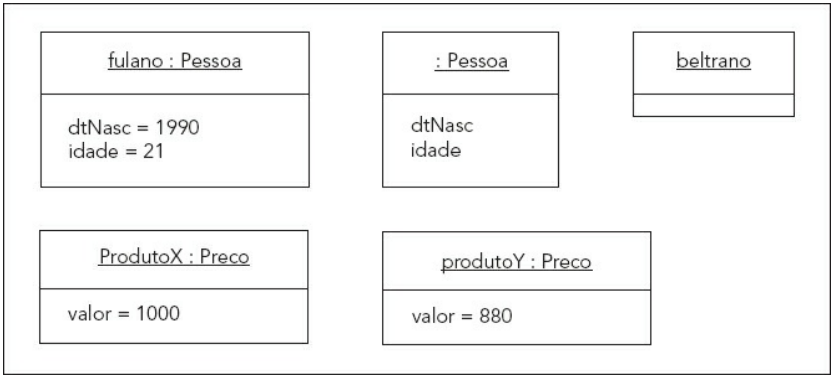


Figura 19 – Notações para objetos.
Fonte: elaborada pelo autor.

Os objetos podem ser representados em diversos diagramas. Entre eles, o diagrama de classe, o diagrama de sequência, o diagrama de comunicação e o próprio diagrama de objetos. O diagrama de objetos geralmente é utilizado para explorar ou validar um contexto ou uma situação em um determinado domínio, representando objetos concretos.

6.2.3 Estereótipos de Classes

Classes podem ser estereotipadas. Ao atribuir um estereótipo a uma classe estende-se a sua semântica natural. A UML predefine alguns estereótipos para classes, mas permite a criação de outros. Os estereótipos para classe predefinidos pela UML são:

- ator (*actor*) – classes que representam alguém ou alguma coisa de interesse para o problema, mas fora da fronteira de automação;
- fronteira (*boundary*) – classes do sistema capazes de coletar dados fornecidos pelos atores ou de fornecer dados destinados a eles;
- controle (*control*) – classes que implementam transações, coordenando as operações de responsabilidade dos objetos de entidades numa única unidade lógica de trabalho;

- entidade (*entity*) – classes que armazenam e manipulam dados que descrevem os objetos da realidade e simulam no sistema o seu comportamento.

Os estereótipos são escritos acima do nome da classe entre os caracteres *guillemets*. Alternativamente, podem ser representados de forma iconográfica. A Figura 20 apresenta classes estereotipadas no formato de texto e no respectivo formato de ícone.

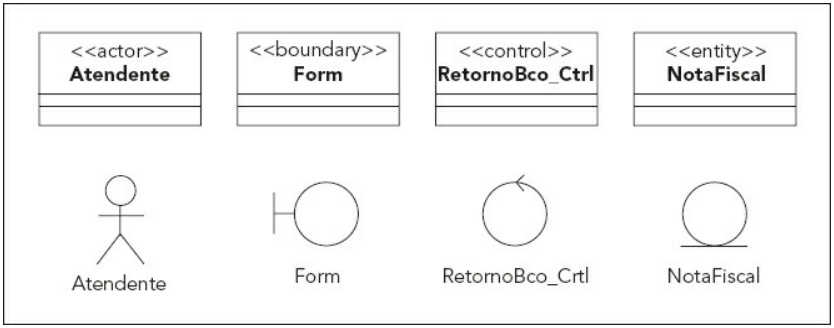


Figura 20 – Estereótipos de classe.
Fonte: elaborada pelo autor.

6.2.4 Relacionamentos

Objetos relacionam-se entre si. A seguir, apresentam-se três tipos de relacionamentos muito importantes para o diagrama de classes:

- associação;
- dependência;
- generalização/especialização.

6.2.4.1 Relacionamento de associação

Uma associação é uma conexão bidirecional entre instâncias de classes. É representada por uma linha unindo as duas classes relacionadas. Cada associação tem duas terminações (*association end*). São relacionamentos do tipo "tem um".

Associações representam conexões estáveis entre dois objetos, como, por exemplo, quando uma classe declara outra como variável de instância. Para comparação, referência a objetos em parâmetros de operações não são modeladas como associações, mas sim como dependências. As associações possuem diversas propriedades:

- nome;
- sentido de leitura;
- papel;
- multiplicidade;
- navegabilidade.

As associações podem receber nomes. O sentido de leitura da associação, representado por uma "seta" hachurada, pode ser desenhado para facilitar a compreensão e leitura do diagrama. A Figura 21 mostra a classe Cliente associada à classe Empréstimo. O nome e sentido de leitura permitem ler esta associação como "cliente realiza empréstimo".

Figura 21 – Associação com nome e sentido de leitura.
Fonte: elaborada pelo autor.

Cada uma das duas terminações pode ser explicitamente nomeada com um rótulo. Este rótulo é chamado de nome do papel e indica o papel que uma classe executa numa determinada associação. Se não for informado, é assumido como nome do papel o nome da classe da terminação destino. A Figura 22 exibe uma associação com especificações de papel em ambas as terminações. Nessa associação, o fornecedor cumpre o papel de contratado, ao passo que a organização cumpre o papel de contratante.

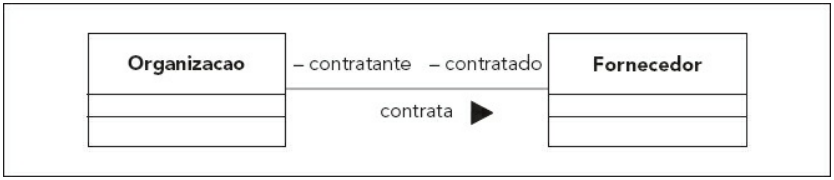


Figura 22 – Associação com papéis.
Fonte: elaborada pelo autor.

A multiplicidade indica o número mínimo e máximo de objetos que podem participar de uma associação. A cada terminação pode ser atribuída uma multiplicidade. A seguir, algumas possibilidades de multiplicidade:

- 0..1 – no mínimo nenhum objeto e no máximo um objeto associado;
- 0..* – no mínimo nenhum objeto e no máximo um número indefinido de objetos associados (representação alternativa: *);
- 1..1 – um e somente um objeto associado (representação alternativa: 1);
- 1..* – no mínimo um objeto e no máximo um número indefinido de objetos associados;
- 5..12 – no mínimo 5 e no máximo 12 objetos associados;
- 5,7,9 – exatamente 5, 7 ou 9 objetos associados.

A Figura 23 demonstra a utilização de associações com multiplicidade definida. A associação pode ser lida como “um cliente pode realizar muitos empréstimos”, ou ainda, “um empréstimo é de um e somente um cliente”.

Figura 23 – Associação com multiplicidade definida.
Fonte: elaborada pelo autor.

A navegabilidade indica em que direção uma associação pode ser percorrida ou, em outras palavras, define se os objetos de uma classe podem acessar instâncias da outra classe do relacionamento. Se a navegabilidade se dá em uma única direção, diz-se que a associação é unidirecional; caso contrário, se diz que ela é bidirecional. A navegabilidade pode não ser necessariamente no sentido de leitura e, geralmente, é definida na fase de projeto, quando se examina a colaboração entre os objetos.

A indicação do sentido de navegabilidade é dada por uma seta (Figura 24). Uma associação sem nenhuma seta tem sua navegabilidade indefinida (Figura 25), ao passo que uma associação com setas em ambas as terminações é bidirecional (Figura 26). Para indicar que uma associação não é navegável em determinado sentido, a terminação não navegável é adornada com um X (Figura 27).

Figura 24 – Navegabilidade de A para B, sendo a navegabilidade de B para A indefinida.
Fonte: elaborada pelo autor.

Figura 25 – Navegabilidade indefinida.
Fonte: elaborada pelo autor.

Figura 26 – Navegabilidade bidirecional.

Fonte: elaborada pelo autor.

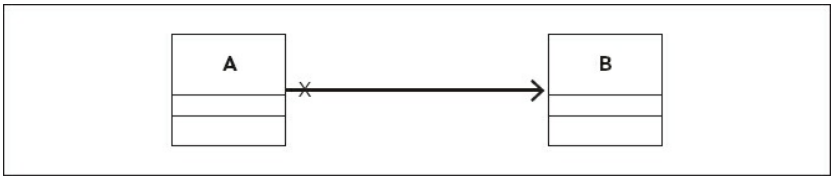


Figura 27 – Navegabilidade de A para B, não havendo navegabilidade de B para A.
Fonte: elaborada pelo autor.

6.2.4.2 Associação reflexiva

Uma associação é reflexiva quando uma instância de uma classe relaciona-se com outra(s) instância(s) da mesma classe. A Figura 28 mostra um exemplo de associação reflexiva. O diagrama pode ser interpretado como "um empregado no papel de supervisor supervisiona muitos empregados no papel de supervisionados" ou ainda como "um empregado no papel de supervisionado tem um e somente um empregado como supervisor".

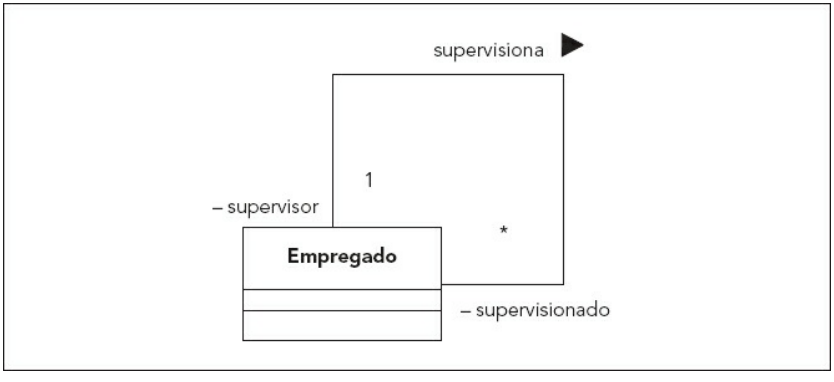


Figura 28 – Associação reflexiva.
Fonte: elaborada pelo autor.

6.2.4.3 Associação múltipla

Uma associação é múltipla quando há mais de uma associação entre as mesmas classes. A Figura 29 exibe o caso de uma associação múltipla. Um Empregado relaciona-se com o Departamento por duas associações. A primeira pode ser interpretada como "um empregado trabalha em um e somente um departamento" ou como "um departamento tem muitos empregados". Já a segunda associação pode ser interpretada como "um empregado é gerente de nenhum ou um departamento" ou ainda, "um departamento tem um único empregado como gerente".

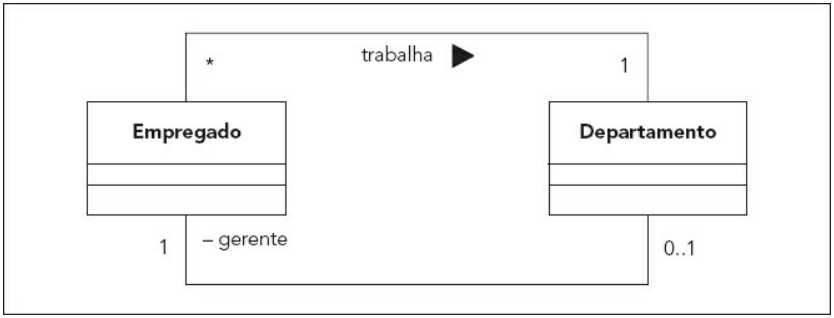


Figura 29 – Associação múltipla.
Fonte: elaborada pelo autor.

6.2.4.4 Classe de associação

É uma associação que tem propriedades semelhantes às de uma classe, tais como atributos, operações e outras associações. O conceito de classe de associação dever ser usado somente quando o objeto oriundo da associação tiver um comportamento próprio. No exemplo da Figura 30, a informação "data de contratação" não é exclusiva do Empregado, tão pouco da Organização. Ela é um atributo de um objeto que só passa a existir quando o Empregado é associado à Organização e um contrato de emprego é criado. O contrato de emprego também não pode existir se não pela associação do Empregado à Organização. A modelagem que utiliza uma classe de associação permite que a associação ContratoEmprego tenha seus próprios atributos, comportamentos e relacionamentos.

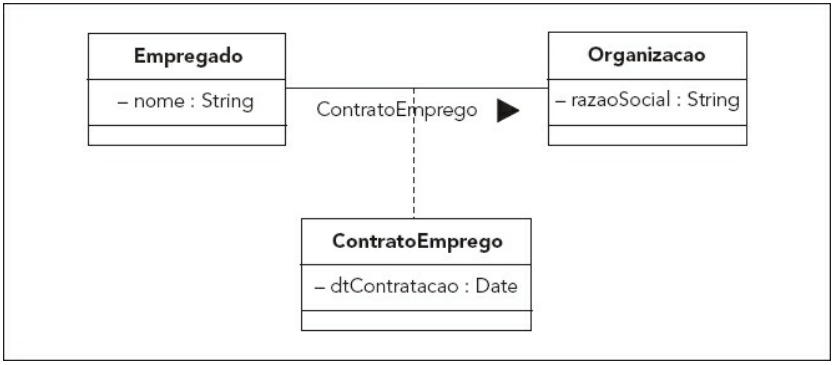


Figura 30 – Classe de associação.
Fonte: elaborada pelo autor.

6.2.4.5 Agregação

Agregações são usadas para modelar associações entre objetos do tipo todo-parte, na qual um objeto, chamado de objeto-todo, agrega ou é composto por diversos objetos chamados de objetos-parte. Agregações podem ser de dois tipos: agregação compartilhada (ou, simplesmente, agregação) e agregação não compartilhada (ou composição).

Agregação compartilhada é um tipo especial de associação na qual um objeto-parte pode estar associado a diferentes objetos-todo. Esta notação é tida como uma espécie de placebo, pois não implica implementação específica, equivalendo à implementação de uma associação normal. Entretanto, pode ser usada para melhorar a comunicação do modelo. São representadas com um losango não hachurado na terminação da classe agregadora. A Figura 31 mostra um exemplo de agregação no qual um Curso agrega muitas Disciplinas.

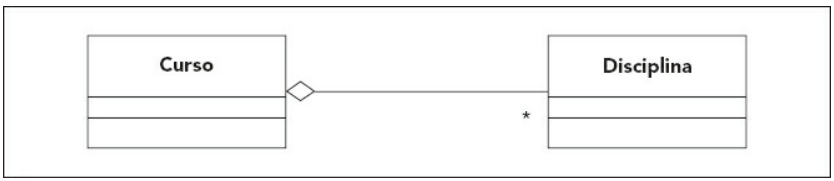


Figura 31 – Agregação.

Fonte: elaborada pelo autor.

6.2.4.6 Composição

Composição é um tipo especial de agregação onde um objeto-parte pertence a um único objeto-todo. É também chamada de agregação não compartilhada. Geralmente objetos-parte são criados e destruídos pelo objeto-todo e os objetos-parte não podem continuar existindo após a destruição do objeto-todo. A Figura 32 traz um exemplo de composição. Um pedido é composto por itens de pedido. Um item de pedido é criado após a criação de um pedido. Se o pedido for excluído, os itens também devem ser. Não faz sentido um item de pedido existir sem estar vinculado a um pedido. Além disso, um item de pedido é exclusivo de um único pedido, não podendo fazer parte de outro pedido. Para comparação, no exemplo de agregação compartilhada, uma disciplina pode pertencer a mais de um curso. Se um curso for extinto, as disciplinas do mesmo podem permanecer vinculadas a outros cursos.

Uma composição é representada por uma associação com um losango hachurado na terminação do objeto agregador. Diferentemente da agregação compartilhada, a composição implica um comportamento bastante específico às classes envolvidas.

Figura 32 – Composição.
Fonte: elaborada pelo autor.

6.2.4.7 Relacionamento de dependência

Um relacionamento de dependência indica que um elemento cliente tem conhecimento de outro elemento fornecedor e que uma modificação no fornecedor pode afetar o cliente. É representado por uma linha tracejada com uma seta partindo do cliente para o fornecedor. Refere-se a relacionamentos do tipo “usa um”.

O uso mais comum dos relacionamentos de dependência no diagrama de classe é na atividade de projeto, não sendo muito utilizados na análise. No projeto, ao se

investigar as interações entre objetos, pode-se identificar classes com operações cuja assinatura ou método usam objetos de outras classes. Esta relação não é estrutural e, por isso, não é representada como uma associação, mas sim como uma dependência.

A Figura 33 mostra o relacionamento de dependência entre a classe RetornoBco_Ctrl e a classe Log. Pode-se inferir que uma operação da classe controladora RetornoBco_Ctrl usa a classe Log para registrar eventos de sua execução, ou seja, a classe RetornoBco_Ctrl é dependente da classe Log. Nesse exemplo, uma alteração na interface pública (conjunto de assinatura das operações de uma classe) da classe Log, no caso a classe fornecedora, pode afetar a classe RetornoBco_Ctrl, que neste caso é a classe cliente.

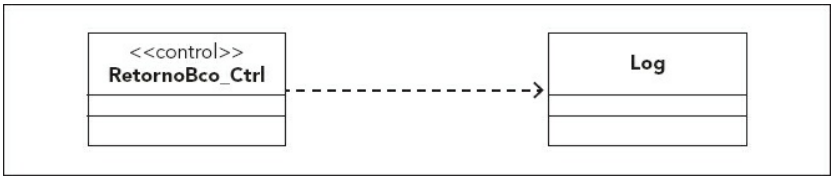


Figura 33 – Relacionamento de dependência.
Fonte: elaborada pelo autor.

O relacionamento de dependência é utilizado nas seguintes situações:

- a classe fornecedora é utilizada como parâmetro de uma operação da classe cliente;
- a classe fornecedora é utilizada como retorno de uma operação da classe cliente;
- a classe fornecedora é utilizada como variável local de uma operação da classe cliente;
- a classe fornecedora é utilizada como uma interface a ser implementada pela classe cliente.

6.2.4.8 Relacionamento de generalização/especialização

É um relacionamento em que cada instância de uma classe mais especializada, chamada de subclasse, é também uma instância de uma classe mais geral, chamada de superclasse. Os objetos de uma subclasse herdam os atributos, o comportamento e os

relacionamentos de sua superclasse. A notação é uma linha contínua conectando a subclasse à superclasse, contendo um triângulo hachurado tocando a superclasse.

A Figura 34 mostra um relacionamento de generalização/especialização. A superclasse é a classe Pessoa, que contém dois atributos: nome e endereço. As duas outras classes, PessoaFisica e PessoaJuridica são subclasses da classe Pessoa e, portanto, herdam os atributos e, se estivessem definidos, o comportamento e os relacionamentos da classe Pessoa. De outro lado, pode-se dizer que as classes PessoaFisica e PessoaJuridica são especializações da classe Pessoa. Para todos os efeitos, pode-se pensar na classe PessoaFisica como contendo os atributos *nome*, *endereço* e *CPF* e a classe PessoaJuridica contendo os atributos *nome*, *endereço* e *CNPJ*. No exemplo, aproveitou-se para mostrar duas formas alternativas de utilizar a notação. Relacionamentos de generalização/especialização são ditos relacionamentos do tipo "é um".

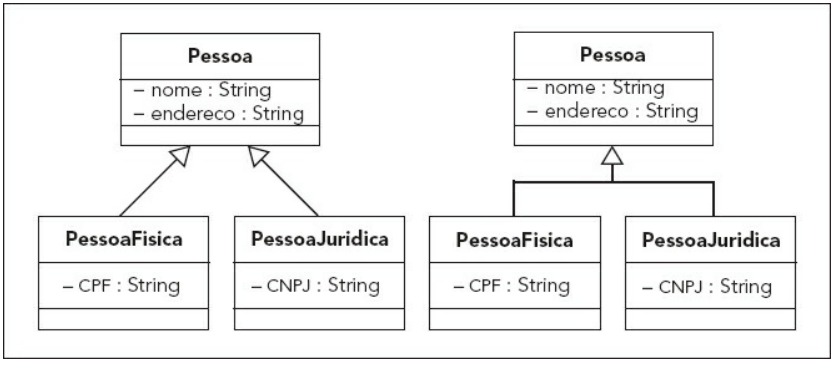


Figura 34 – Relacionamento de generalização/especialização.
Fonte: elaborada pelo autor.

6.2.4.9 Classes Abstratas e Concretas

Classes abstratas são as que não possuem instâncias diretas, mas cujas classes descendentes sim, ao passo que classe concreta é uma classe instanciável. Classes abstratas podem surgir naturalmente ou ser introduzidas artificialmente para facilitar a reutilização do código e são frequentemente usadas para definir métodos a ser herdados pelas subclasses, como mostra a Figura 35.

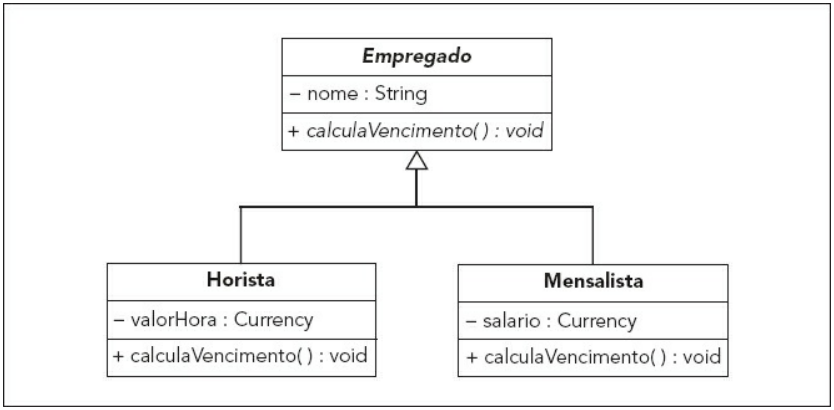


Figura 35 – Classes concretas e classes abstratas.
Fonte: elaborada pelo autor.

No diagrama da Figura 35, a classe Empregado é uma classe abstrata, ou seja, nenhum objeto do tipo Empregado poderá ser criado. Para diferenciar uma classe concreta de uma abstrata, a notação da UML define que o nome da classe abstrata deve vir em itálico. O mesmo ocorre com a operação *calculaVencimento*() da classe Empregado. Ela também é abstrata e por isto também está em itálico. Uma operação abstrata não tem implementação. As classes Horista e Mensalista, por sua vez, são classes concretas, assim como as operações calculaVencimento() em cada uma delas. Em cada uma dessas classes, a operação calculaVencimento() será implementada de acordo com o tipo de objeto a qual pertencer (Horista ou Mensalista).

6.3 Construção do Modelo de Domínio

Essa atividade visa criar um modelo de domínio (também chamado de modelo conceitual), que nada mais é do que a decomposição de um domínio em classes conceituais. Por domínio, entende-se o ambiente em observação, geralmente um sistema.

O modelo de domínio representa os conceitos de um domínio e seus relacionamentos, independentemente de preocupações com implementação e restrições tecnológicas. Ele deve descrever a informação que o sistema necessita gerenciar. É ilustrado com um conjunto de diagramas de classes em que nenhuma

operação é definida. Os seguintes elementos podem ser usados para representar a informação:

- classes, para representar conceitos – informação complexa, que não pode ser descrita meramente por tipos de dados primitivos;
- atributos, para representar informações de tipos de dados primitivos ligadas diretamente aos conceitos;
- relacionamentos que ligam conceitos entre si.

O Diagrama de Classes pode ser usado tanto para desenhar o modelo de domínio, quanto para desenhar o modelo de projeto. A diferença está na perspectiva. O modelo de domínio é um diagrama dos conceitos do domínio, uma representação de classes conceituais do mundo real, não de objetos de software. Está numa perspectiva conceitual. Já no modelo de projeto são representados objetos e classes computacionais. A perspectiva é de especificação e implementação. Embora haja uma certa semelhança, o modelo conceitual também não deve ser confundido com modelo de dados.

O modelo de domínio é único para todo o sistema e vai sendo completado na medida em que novas iterações vão sendo realizadas. As motivações para se criar um Modelo de Domínio são capturar e entender os principais conceitos de um domínio e diminuir o hiato representacional entre os modelos mentais e o de software.

Um cuidado muito importante ao modelar-se um domínio é usar o nível de abstração adequado. A designação de um objeto em uma classe ou outra dependerá do nível de abstração desejado. Quanto maior a quantidade de características de um objeto desconsideradas para sua classificação, maior o nível de abstração. O nível de abstração deve ser ajustado para modelar apenas os elementos necessários (no nível em que os manipularemos) e evitar a presença de elementos desnecessários.

As principais tarefas para compor o modelo de domínio são:

- identificar as classes e atributos;
- identificar os relacionamentos;
- fazer um Diagrama de Classes.

6.3.1 Identificando Classes e Atributos

Existem dois métodos principais para identificar classes, atributos e relacionamentos: um método orientado a responsabilidades e outro orientado a dados. O método orientado a dados procura identificar e analisar a estrutura dos conceitos relevantes para um domínio de negócio. Parte da análise linguística e vale-se de heurísticas para definir classes, atributos e relacionamentos. Dará origem a um modelo conceitual. Este livro foca neste método. O método orientado a responsabilidades não será apresentado.

A análise linguística trata de analisar todos os elementos textuais que possam referenciar informação a ser guardada e/ou processada pelo sistema. Os casos de uso são a principal fonte de análise, pois suas descrições estão repletas de informações que precisam ser representadas no modelo de domínio. Além dos casos de uso, outros documentos como o documento de visão e o glossário, podem ser utilizados.

Ao proceder a leitura dos casos de uso e outros documentos, procura-se identificar no texto substantivos que correspondam a conceitos sobre os quais se tem interesse em manter informação no sistema. Deve-se prestar atenção a sinônimos ou mesmo a palavras ou expressões que representem o mesmo conceito. Por exemplo, as palavras aluno, docente ou estudante podem estar presentes em diferentes documentos, mas referir-se a um mesmo conceito.

Na leitura, deve-se diferenciar itens que correspondam a conceitos complexos dos itens que são caracterizações de conceitos. Os conceitos serão mapeados como classes e suas características como atributos. O uso de uma lista de categorias de conceitos, como a seguinte, pode auxiliar no processo:

- elementos tangíveis;
- funções;
- eventos;
- transações;
- especificações.

Elementos tangíveis englobam conceitos que tenham existência concreta, como um livro, um carro ou uma nota fiscal. Outras vezes, percebemos um elemento através da função por ele exercida e não pelo o que ele é fisicamente. A função refere-se à atuação de um elemento no ambiente em que está inserido. Por exemplo: professor, cliente ou aluno são funções exercidas por entidades físicas, no caso, pessoas. Eventos referem-se a elementos que só podem ser percebidos e caracterizados enquanto certa

ação se desenrola. Por exemplo, um voo, uma aula, um acidente. Transações são eventos que estabelecem uma relação estável entre dois ou mais elementos e que pode resultar no surgimento de outros elementos. Como exemplo, pode citar-se a compra de um imóvel, uma inscrição em um curso e a contratação de um empréstimo. Especificações são elementos que definem características de outros objetos, como a especificação de um carro e a de um microcomputador.

De maneira geral, se não for possível pensar em um substantivo como apenas números ou texto no mundo real, provavelmente este substantivo refere-se a uma classe e não um atributo. Na expressão "voo 123 com destino a Porto Alegre" o substantivo voo deve ser mapeado como uma classe, pois, embora seja designado por um número, não é possível imaginá-lo como simplesmente um número no mundo real. De forma similar, o substantivo destino deve ser mapeado como uma classe, pois se refere a uma cidade. Embora a cidade tenha um nome que pode ser expresso por letras, ela não é, no mundo real, uma cadeia de caracteres.

Ao nomear classes e atributos é aconselhável utilizar os nomes próprios do domínio, evitando-se a "invenção" de nomes ou uso de codificações. Isto ajudará a diminuir o hiato representacional entre o domínio e seu modelo. Por fim, é uma boa prática adicionar ao modelo uma classe controladora que represente o próprio sistema.

Alguns tipos de dados podem der definidos por classes. Representa-se o que, a princípio, poderia ser considerado um tipo de dado primitivo como uma nova classe de tipo de dados no modelo de domínio se:

- o atributo for composto de seções separadas (p. ex.: número de telefone, nome de pessoa);
- existirem operações associadas a ele, como análise sintática ou validação (p. ex.: CPF, que possui a validação dos dígitos de controle);
- o atributo for uma quantidade com uma unidade (p. ex.: preço – R$ 10,00; massa – 20 kg);
- o atributo tiver outros atributos (p. ex.: preço promocional, que pode ter datas de início e de fim).

Em um diagrama de classes, as representações de tipos de dados exibidas na Figura 36 são semanticamente equivalentes.

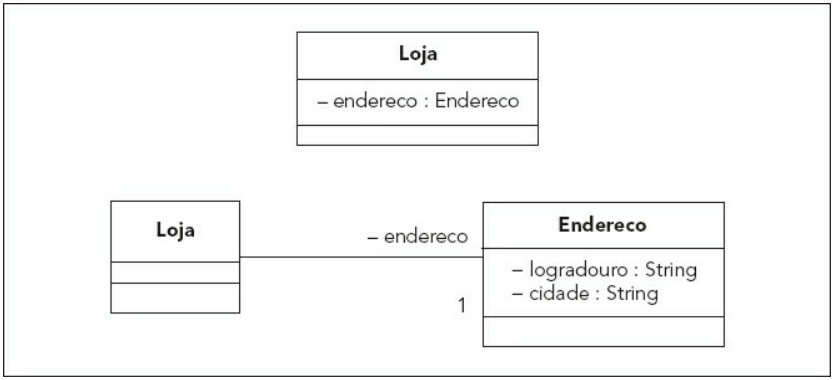


Figura 36 – Tipos de dados definidos por classes.
Fonte: elaborada pelo autor.

6.3.2 Identificando Relacionamentos

Na construção do modelo conceitual, o foco está na descoberta de relacionamentos de associação e de generalização/especialização. Os relacionamentos de dependência podem e devem ficar para o projeto.

6.3.2.1 Identificando associações

Para identificar associações deve-se, para cada conceito, verificar se os seus atributos são suficientes para armazenar toda a informação que o conceito deve prover. Se não forem, deve-se associar o conceito a outros. Por exemplo, em um sistema de gestão de biblioteca, os conceitos usuário e livro possivelmente podem ser caracterizados completamente por seus atributos. Entretanto, o conceito empréstimo precisará necessariamente estar associado a um usuário e a um livro.

Os conceitos que podem existir por si só são chamados de conceitos independentes. Já os conceitos que precisam de outros para existir ou fazer sentido são chamados de conceitos dependentes. Os conceitos independentes, em muitos casos, acabam por ser associados à classe controladora do sistema.

Não se deve representar no modelo conceitual atributos que representam "chaves estrangeiras" como é usual em modelo de dados. A Figura 37 mostra a forma correta e a incorreta de se representar associações em diagramas de classe.

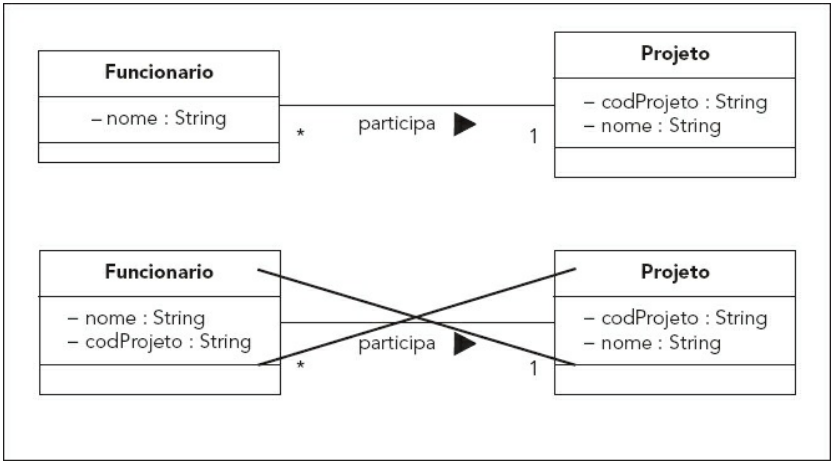


Figura 37 – Modelagem de relacionamentos com "chave estrangeira".
Fonte: elaborada pelo autor.

Deve-se ter cuidado ao considerar casos de uso para mapear associações. Casos de uso descrevem ações de interação entre atores e o sistema, que geralmente referem-se a operações de sistemas e não a associações. Por exemplo, admitindo-se em um sistema a existência dos conceitos funcionário e cliente, modelados cada um como uma classe e que, neste sistema, um funcionário cadastra um cliente. Seria incorreto criar uma associação entre ambos a partir da situação apresentada, visto que nenhuma relação estática é criada entre ambos quando o funcionário cadastra o cliente. O que se tem neste caso é uma operação de sistema, possivelmente modelada como uma interação em um diagrama de casos de uso. Esta associação somente teria sentido se fosse necessário registrar a informação de qual funcionário cadastrou qual cliente. A Figura 38 demonstra esta situação.

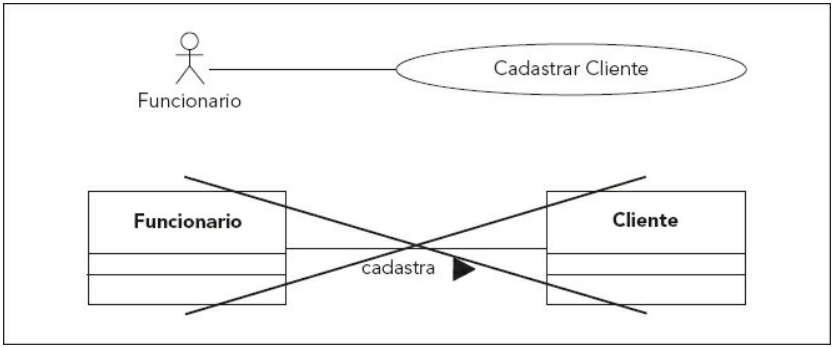


Figura 38 – Modelagem de operações de sistema como relacionamentos.
Fonte: elaborada pelo autor.

Não é uma boa prática atribuir navegabilidade a associações no modelo conceitual. A navegabilidade de uma associação deve ser definida no projeto, quando se investigar a troca de mensagens entre objetos nos diagramas de interação.

6.3.2.2 Identificando Generalizações e Especializações

O relacionamento de generalização/especialização entre conceitos deve ser utilizado para representar uma relação estrutural entre conceitos que possuem atributos, comportamento e relacionamentos comuns. Por relacionamento estrutural, entende-se aquele em que os conceitos relacionados podem ter uma classificação taxonômica definitiva em uma estrutura hierárquica, onde cada nível contém características comuns a todas as classes do nível descendente. Deve-se evitar o uso de generalização/especialização quando o relacionamento entre os conceitos for de papel ou temporal.

Um relacionamento de papel é aquele em que um conceito representa um papel para outro. Por exemplo, uma pessoa pode representar junto a uma empresa o papel de cliente ou de funcionário. Nesse caso, tem-se duas associações distintas entre pessoa e empresa, não havendo uma relação estrutural entre cliente, funcionário e pessoa, embora hajam atributos e relacionamentos comuns. Não se deve, portanto, modelar esta situação com relacionamentos de generalização/especialização, conforme mostra a Figura 39.

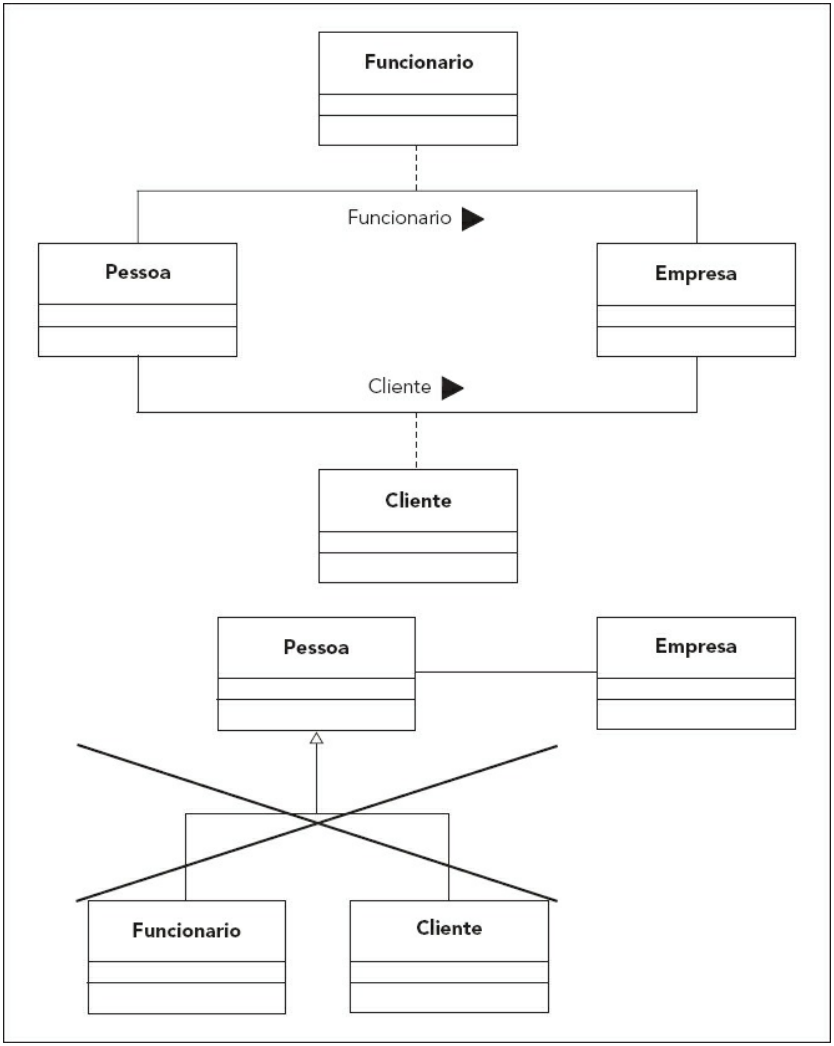


Figura 39 – Modelagem de papel como generalização/especialização.
Fonte: elaborada pelo autor.

Um relacionamento temporal representa relações entre estados de um conceito e o conceito em si, como, por exemplo, a classe pessoa e seus estados civis: solteiro, casado, separado, divorciado e viúvo. Modelar esta situação com relacionamentos de generalização/especialização pode conduzir ao problema de metamorfose, também chamado de classificação dinâmica, no qual um conceito transforma-se em outro (p. ex.: uma pessoa solteira torna-se uma pessoa casada). A Figura 40 demonstra esta situação.

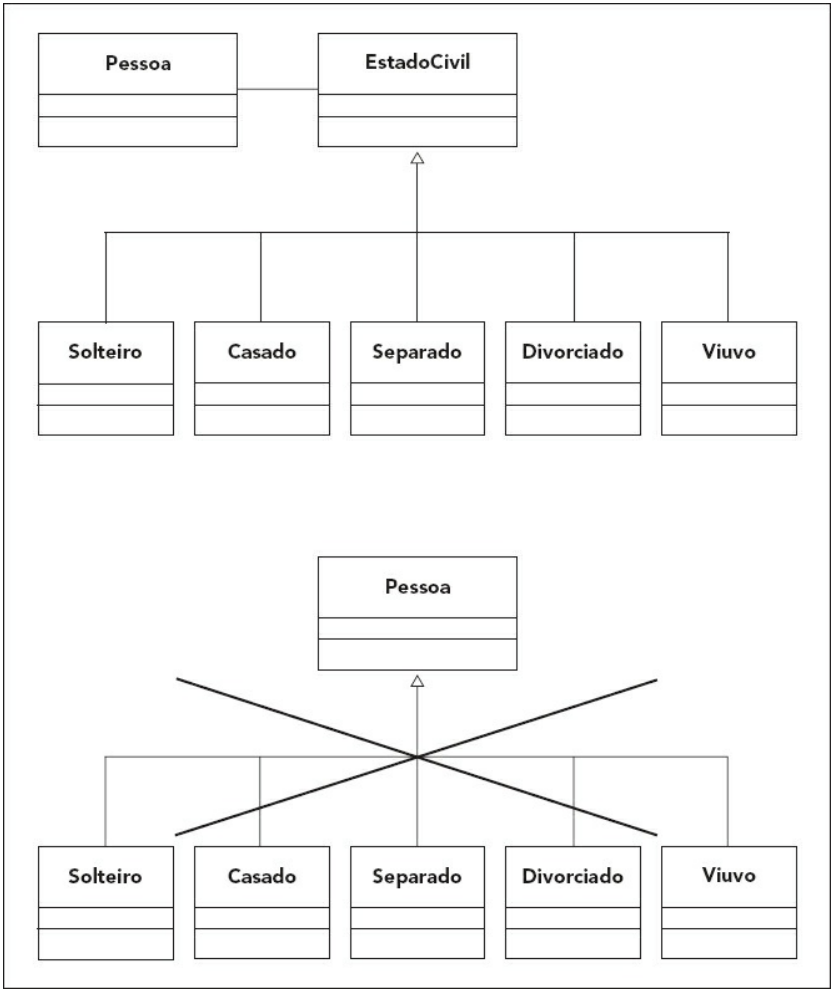


Figura 40 – Modelagem de estados como generalização/especialização.
Fonte: elaborada pelo autor.

6.3.3 Modelo de Domínio do Sistema Exemplo

Este tópico exemplifica a criação de um modelo de domínio a partir da descrição do sistema exemplo. Conforme visto, a construção de um modelo de domínio com o método orientado a dados procura identificar e analisar a estrutura dos conceitos relevantes para um domínio, valendo da análise linguística e de heurísticas para definir classes, atributos e relacionamentos. Neste exemplo, a análise baseou-se exclusivamente na descrição do sistema. Na prática, todos os artefatos textuais devem ser considerados, mas, principalmente, a descrição dos casos de uso. A Figura 41 apresenta o modelo de domínio do sistema SGT.

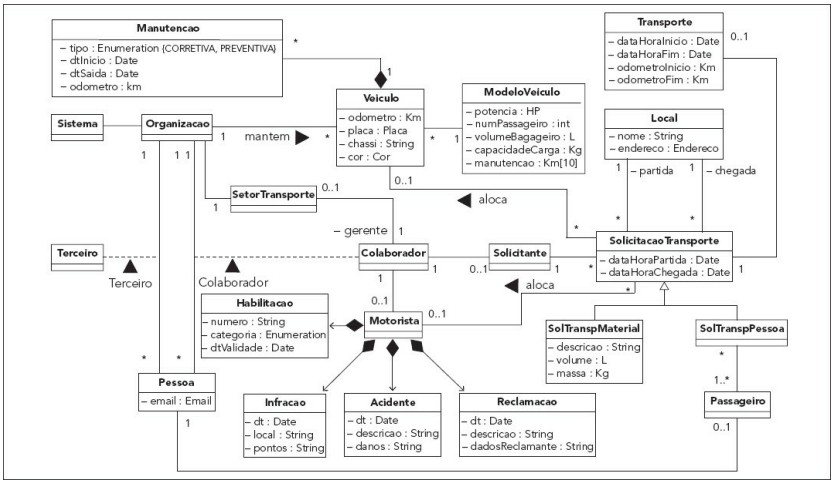


Figura 41 – Modelo de domínio do sistema exemplo.
Fonte: elaborada pelo autor.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar o conhecimento sobre técnicas de modelagem conceitual recomenda-se a leitura de Larman (2007) e Wazlawick (2004). Para uma visão geral sobre o método de modelagem orientado a responsabilidades sugere-se a leitura de Bezerra (2002). Mais detalhes sobre a notação de Diagramas de Classe podem ser obtidos em Guedes (2009).

RESUMO DO CAPÍTULO

A análise foca no entendimento do sistema de forma a viabilizar que uma aplicação de software possa ser projetada para o mesmo. Duas abordagens complementares de análise são importantes: modelagem conceitual e modelagem funcional. A modelagem conceitual visa produzir um modelo de domínio, que contempla as classes conceituais do sistema, seus atributos e seus relacionamentos. A construção de um modelo de domínio pode valer-se da análise linguística dos documentos e artefatos produzidos no processo, principalmente as especificações dos casos de uso. O modelo de domínio deve ser representado graficamente com um diagrama de classes da UML, no qual nenhuma operação de classe é apresentada.

CAPÍTULO 7

ANÁLISE – MODELAGEM FUNCIONAL

Este capítulo enfoca a modelagem funcional da disciplina de análise. Inicialmente, aborda-se a atividade de elaboração dos contratos de operações de sistema. Para tal, mostra-se como identificar as operações de sistema usando-se um diagrama de sequência e como especificar seus contratos. Posteriormente, é apresentado o diagrama de máquina de estados e sua aplicação para investigar o ciclo de vida de objetos do domínio.

Outra atividade da análise é a elaboração dos contratos das operações de sistema. Ela faz parte da modelagem funcional ou dinâmica. A construção do modelo de domínio foca na identificação e modelagem dos dados que o sistema necessita gerenciar. Contudo, é necessário, também, analisar como o sistema obtém e fornece essas informações.

7.1 Operações de Sistema

Em certo grau de abstração, um sistema pode ser visto como uma caixa preta que reage a eventos externos. Os eventos externos, chamados de eventos de sistema, são disparados por atores e tratados por operações de sistema. Operações de sistema são, portanto, operações ativadas a partir de um evento de sistema.

Cada operação de sistema identificada deve ser especificada na forma de um contrato, no qual se descreve o que a operação deve fazer, quais são seus compromissos e quais são os compromissos de quem solicita sua execução. Essas operações serão posteriormente projetadas utilizando-se Diagramas de Comunicação e implementadas em uma classe controladora do sistema ou do caso de uso específico. As operações de sistema, em conjunto, vão corresponder à totalidade das funções do sistema. Elas irão compor a interface do sistema com o mundo externo, ou seja, a interface pública do mesmo.

Casos de uso descrevem como os atores interagem com o sistema de software. Durante essa interação, um ator gera eventos para o sistema e esses eventos invocam operações de sistema e por elas são tratados. Modelam-se essas interações com Diagramas de Sequência, analisando-se os passos das descrições dos casos de uso e

desenhando os respectivos diagramas. Portanto, antes de iniciar a modelagem das operações de sistema, é necessário conhecer o diagrama de sequência da UML.

7.2 Diagrama de Sequência

Programas orientados a objetos operam através de trocas constantes de mensagens entre objetos que colaboram na consecução de um determinado propósito. Diagramas de interação descrevem como objetos colaboram em algum comportamento. Existem dois tipos de diagramas de interação:

- diagrama de sequência;
- diagrama de comunicação.

O diagrama de sequência facilita a observação da ordem temporal das trocas de mensagens ao passo que o diagrama de comunicação facilita a observação de como os objetos estão estaticamente conectados. A Figura 42 traz a representação do trecho de código a seguir, através de um diagrama de sequência e de um diagrama de comunicação, evidenciando as diferentes abordagens entre os dois tipos de diagramas.

```
public class A
{
private B meuB = new B();
public void opUm()
{
meuB.opDois();
meuB.opTres();
}
//....
}
```

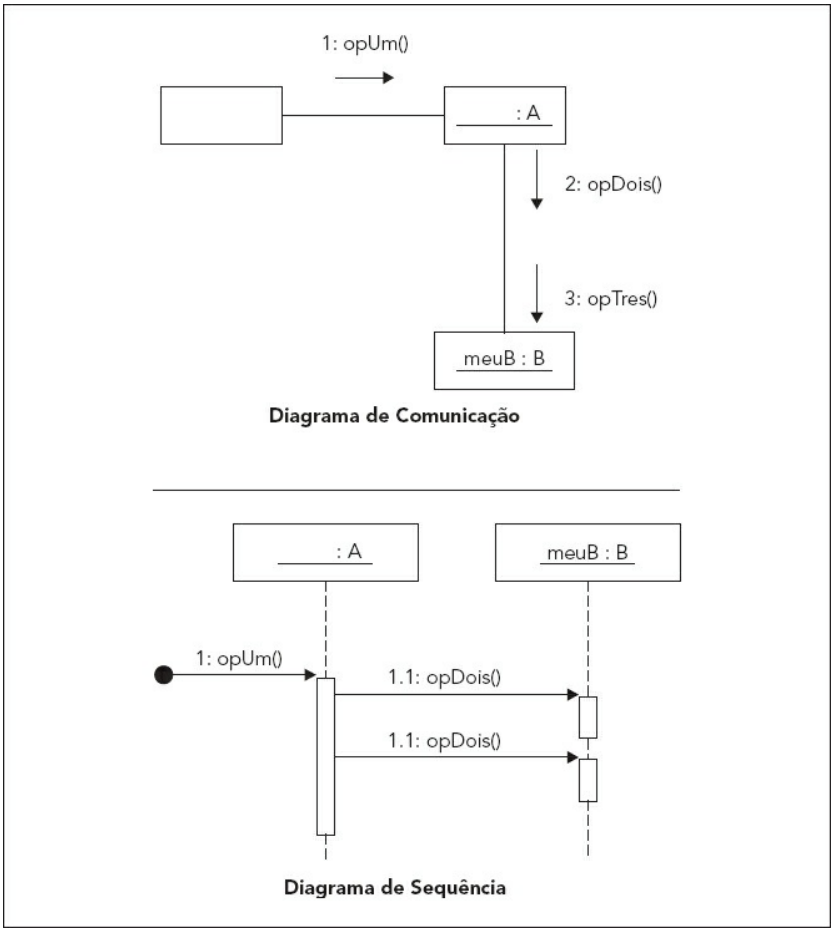


Figura 42 – Diagramas de comunicação e de sequência.
Fonte: elaborada pelo autor.

Diagramas de interação não são bons para uma definição precisa do comportamento. Para modelar um comportamento qualquer em detalhe, deve-se usar o diagrama de atividade. Os diagramas de interação devem ser usados quando se deseja observar o comportamento de vários objetos dentro de um contexto. Se a necessidade

for observar um objeto através de vários contextos, deve-se utilizar o diagrama de máquina de estados.

Algumas ferramentas CASE criam de forma automática um diagrama de comunicação a partir de um diagrama de sequência e vice-versa. Um diagrama de comunicação pode conter mais detalhes sobre os relacionamentos entre os objetos e é mais utilizado na atividade de projeto.

O diagrama de sequência representa a sequência temporal das mensagens trocadas por um conjunto de objetos em um determinado escopo. A ordem é de cima para baixo. O diagrama de sequência não é um fluxograma. Os principais elementos de um diagrama de sequência são:

- objetos;
- linhas de vida dos objetos;
- mensagens trocadas entre objetos.

As mensagens são trocadas entre objetos e não entre classes. Por esta razão, os diagramas de interação representam objetos e não classes. A ordem dos objetos no diagrama é arbitrária. A notação de um objeto no diagrama de sequência é a mesma do diagrama de classes ou de objetos. No diagrama de sequência podem ser representados objetos de qualquer tipo de classe, seja ela um ator, uma classe de fronteira, controladora ou de entidade.

As linhas de vida representam o tempo desde a criação/instanciação de um objeto até sua destruição. É representada por uma linha vertical tracejada. Um retângulo vertical sobre a linha de vida do objeto representa o tempo durante o qual um objeto fica responsável pelo fluxo de controle e equivale ao tempo em que determinada operação permanece ativa (em execução).

Cada mensagem é representada por uma seta entre linhas de vida de dois objetos. As mensagens podem ser disparadas da esquerda para direita ou vice-versa. Cada mensagem tem um nome e pode incluir alguns argumentos, um valor de retorno e uma condição de guarda. Uma mensagem é enviada apenas quando a condição de guarda for verdadeira. Automensagem é uma mensagem enviada de um objeto para ele mesmo.

As mensagens podem ser síncronas ou assíncronas. Mensagem síncrona é aquela em que o objeto remetente espera o objeto destino finalizar sua tarefa para então continuar a execução. Sua notação é uma seta com ponta fechada e hachurada. Mensagem assíncrona é aquela em que o objeto remetente não precisa esperar pelo objeto destino. A linha de execução pode continuar sem que haja uma resposta ao

remetente. A notação é uma seta com ponta aberta.

A Figura 43 mostra os principais elementos de um diagrama de sequência. Observa-se que a mensagem1() tem dois argumentos, respectivamente, arg1 e arg2 e uma variável de retorno ret. Essa mensagem somente será enviada se a condição de guarda [x==y] for satisfeita.

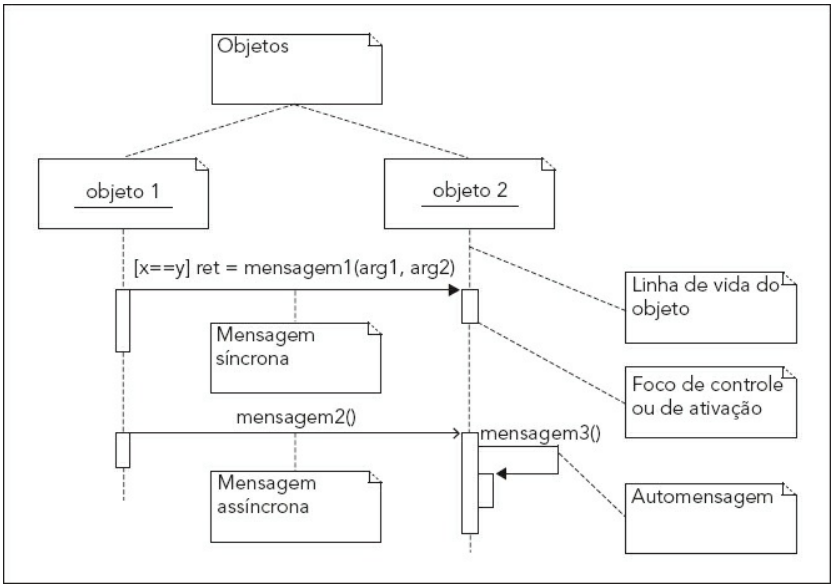


Figura 43 – Diagrama de sequência.
Fonte: elaborada pelo autor.

É possível denotar o retorno de mensagens com uma notação alternativa, utilizando-se uma seta tracejada. O retorno não pode ser considerado um tipo de mensagem e deve ser utilizado apenas quando ajudar a compreensão do diagrama. Existem dois tipos especiais de mensagens: mensagem de criação e de destruição. As mensagens de criação determinam a criação (início da linha de vida) de um objeto e são marcadas com estereótipo «create». Mensagens de destruição determinam a destruição (fim da linha de vida) de um objeto e são marcadas com estereótipo «destroy». A Figura 44 exibe essas notações.

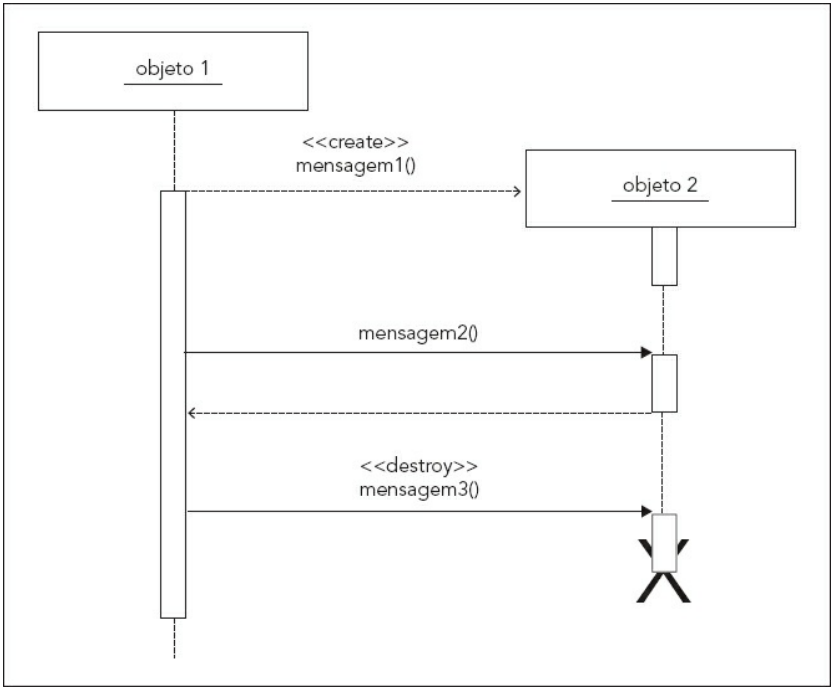


Figura 44 – Mensagens de criação e de destruição.
Fonte: elaborada pelo autor.

A UML 2.0 implementou melhorias em sua notação para representar iterações e fluxos alternativos. Para indicar que um determinado trecho de fluxo será iterado ou está sujeito a uma condição, deve-se delimitá-lo com o elemento gráfico chamado de fragmento combinado. Um fragmento combinado pode ter um operador e condições de guarda. Existe uma série de operadores, dentre os quais se destacam o operador *loop* e o operador *alt*.

O operador *alt* implica que, dentre os diversos fluxos pertencentes ao fragmento, será executado aquele que satisfizer a respectiva condição de guarda. O operador *loop* implica que o fragmento será iterado de acordo com a respectiva condição de guarda. A Figura 45 mostra o uso de fragmentos combinados. Há um fragmento com operador *loop* determinando que todas as mensagens dentro dele serão iteradas para cada conta.

Dentro deste fragmento, há um fragmento com operador *alt*. A seção do fragmento contendo a mensagem2() será executada se o saldo for negativo. Caso contrário, será executada a seção contendo a mensagem3().

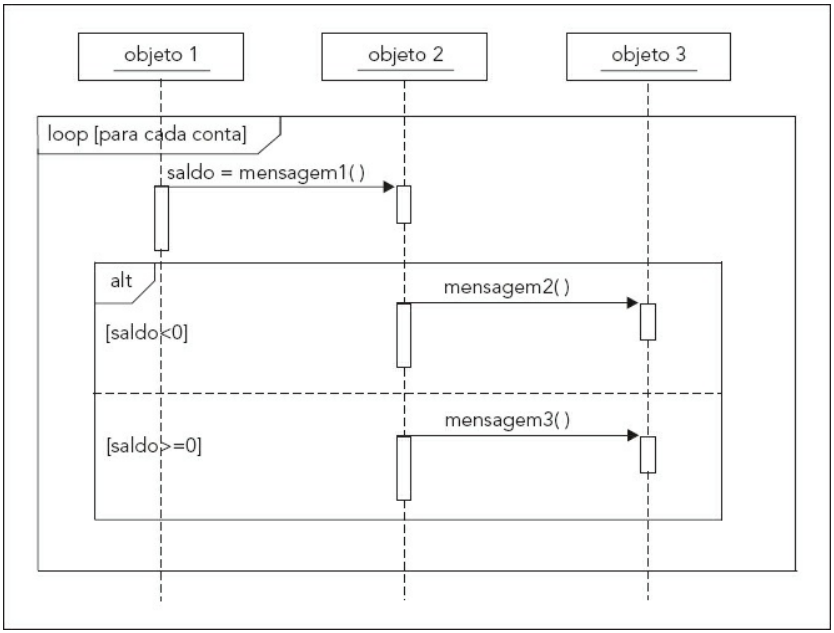


Figura 45 – Fragmentos combinados do diagrama de sequência.
Fonte: elaborada pelo autor.

7.3 Especificação de Operações de Sistema

As operações de sistema podem ser identificadas analisando-se os passos obrigatórios dos casos de uso, pois nesses passos a informação é passada para o sistema ou dele é obtida. Para modelar operações de sistemas, pode-se utilizar o diagrama de sequência da UML.

Considerando-se uma provável arquitetura em camadas, e que a disciplina de análise não considera ainda os objetos internos do sistema, pode-se desenhar três objetos no diagrama de sequência: um objeto representando o ator, um representando

a interface do sistema com o ator (camada de apresentação) e outro representando a camada de domínio. Um ator pode comunicar-se apenas com a camada de apresentação e esta com a camada de domínio.

Na modelagem de operações, o fluxo de informação pode se dar entre atores (comunicação entre atores), dos atores para o sistema (eventos do sistema) e do sistema para os atores (respostas do sistema). Os eventos de sistema podem ser informativos ou de controle. Eventos informativos passam dados para o sistema e implicam, usualmente, operações com parâmetros. Eventos de controle passam uma informação de controle e não necessariamente dados. São usados para iniciar ou finalizar uma ação e usualmente implicam operações sem parâmetros.

Três formas alternativas podem ser usadas para modelar as operações de sistema. Na forma geral, faz-se um diagrama de sequência para cada caso de uso, considerando-se todos os possíveis cenários. Esta forma apresenta como desvantagem o fato de que, se o número de cenários for grande, o diagrama pode ficar muito complexo. Na forma de cenário, faz-se um diagrama de sequência para cada cenário. A desvantagem dessa forma é que se pode gerar redundâncias de fluxos que são comuns a vários cenários. Uma forma alternativa é fatorar o caso de uso em fluxos não redundantes. A desvantagem é que pode ser necessário olhar vários diagramas para visualizar um cenário.

A Figura 46 exibe o diagrama de sequência para o cenário de sucesso de transporte de passageiros do caso de uso UC10 – Solicitar Transporte. Três objetos são representados: um objeto da classe Solicitante com o estereótipo de ator, representando um usuário solicitante; um objeto da classe Form com o estereótipo de fronteira, representando um formulário de interface gráfica com o usuário; e um objeto da classe UC10_Ctrl com estereótipo controlador responsável por tratar os eventos relacionados ao caso de uso UC10. O objeto da classe Form é componente da camada de apresentação e o objeto da classe UC10 é componente da camada de domínio. O primeiro tem a função de receber as entradas de dados e eventos do ator e repassá-las à camada de domínio através de operações de sistema, bem como apresentar os resultados das operações ao ator. O segundo é o objeto da camada de domínio que se relaciona com a camada de apresentação através de uma interface que especifica diversas operações de sistema. Em vez de um objeto de uma classe referente a um caso de uso, poderia-se utilizar um objeto de uma classe controladora representando todo o sistema. Optou-se pela primeira alternativa por se entender que isto pode representar um avanço em direção ao projeto de software, pois, ao final da identificação das operações de sistema de todos os casos uso, estas já estariam atribuídas às classes

controladoras de cada caso de uso.

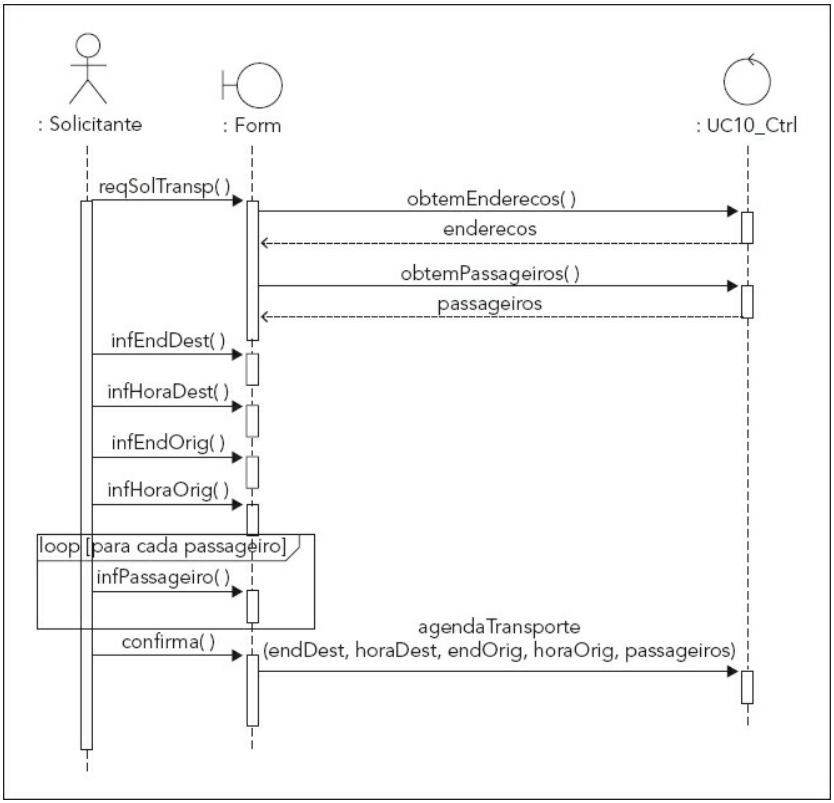


Figura 46 – Diagrama de sequência do caso de uso UC10 – Solicitar Transporte.
Fonte: elaborada pelo autor.

Observa-se na Figura 46 que, tão logo o ator acesse o formulário de solicitação de transporte, a camada de apresentação envia duas mensagens à camada de domínio. A essas mensagens corresponderão a duas operações de sistema que têm por finalidade obter os endereços e passageiros cadastrados e repassá-los à camada de apresentação de forma popular às respectivas listas no formulário.

Na sequência, o diagrama apresenta seis mensagens enviadas do ator à camada de

apresentação. As cinco primeiras são eventos informativos no quais o ator passa informação ao sistema. A presença de um fragmento combinado indica que a mensagem infPassageiro() será repetida para cada passageiro a ser transportado. A sexta mensagem refere-se a um evento de controle, no qual nenhum dado é passado do ator ao sistema, mas sim, a indicação de que o solicitante confirma a solicitação de transporte. Com a confirmação, a operação de sistema agendaTransporte é executada. Tem-se, então, que, para tratar o cenário em questão do UC10, será necessário o projeto e a implementação de três operações de sistema: obtemEnderecos(), obtemPassageiros() e agendaTransporte(). Um passo anterior ao projeto, entretanto, será a especificação da operação através de um contrato.

7.3.1 Especificando as Operações

Operações de sistema devem ser especificadas na análise. Cada operação deve ter um nome, parâmetros e um contrato. Essas operações serão posteriormente implementadas em uma classe controladora (classe do sistema ou de caso de uso). Os contratos devem ser especificados como expressões interpretáveis em termos dos elementos do modelo conceitual. Cada contrato de operação pode conter os seguintes elementos:

- objetivo;
- precondições;
- pós-condições;
- exceções;
- resultados.

Objetivo é a descrição sucinta do objetivo da operação. Precondições definem o que deve ser verdadeiro na estrutura da informação armazenada para que a operação possa ser executada. As precondições podem ser de garantia de parâmetro ou de restrição complementar. Uma precondição de garantia de parâmetro determina quais condições devem ser satisfeitas em relação aos parâmetros (p. ex.: o cliente passado como parâmetro existe). Uma restrição complementar determina uma regra do domínio que deve ser satisfeita (p. ex.: o aluno está matriculado).

Pós-condições estabelecem o que uma operação de sistema muda na estrutura da informação armazenada. As modificações que uma operação pode produzir na

estrutura da informação armazenada se restringem, basicamente, às seguintes possibilidades:

- criação de uma instância de um conceito – nesse caso, as associações deste conceito devem ser estabelecidas;
- destruição de uma instância de um conceito – nesse caso, todas as associações ligadas à instância do conceito devem ser eliminadas;
- alteração de atributo de uma instância de um conceito;
- criação de relacionamentos entre instâncias de conceitos;
- destruição de relacionamentos entre instâncias de conceitos.

Exceções são situações de erro ou falha que não podem ser verificadas antes de se iniciar a execução da operação e impedem o prosseguimento correto da mesma, não podendo ser tratadas sem a intervenção do usuário. Resultados indicam quais são as informações que uma operação de consulta retorna para serem disponibilizadas em uma interface. Uma operação de consulta é uma operação de sistema que simplesmente recupera informação armazenada, sem modificar a mesma. O Quadro 7 apresenta o contrato da operação de sistema agendaTransporte().

Quadro 7 – Especificação do contrato da operação de sistema agenda Transporte()

Operação: agendaTransporte

Parâmetros:

- endDest: – endereço de destino do transporte
- endOrig – endereço de partida do transporte
- horaDest – hora de chegada ao destino
- horaOrig – hora de chegada ao local de partida do transporte
- passageiros – lista de passageiros que serão transportados

Objetivo:

- agendar um transporte e comunicar os envolvidos sobre o agendamento.

Precondições:

- os endereços de destino e partida são endereços válidos;

- a hora de chegada e partida são horários válidos;
- os passageiros da lista estão cadastrados no sistema.

Pós-condições:

- uma solicitação foi criada no sistema e associada ao solicitante;
- os atributos horaIni e horaFim foram alterados para horaOrig e horaDest;
- foi associado à solicitação um local de partida considerando endOrig;
- foi associado à solicitação um local de chegada considerando endDest;
- foi associado à solicitação um veículo considerando a quantidade de passageiros e a disponibilidad do mesmo;
- foi associado à solicitação um motorista com disponibilidade de horário para atender a solicitação com habilitação para o tipo de veículo;
- foi associado à solicitação cada um dos passageiros considerando a lista de passageiros recebida como parâmetro;
- foi emitido um e-mail para cada passageiro informando: hora e local de partida, hora e local de chegada, o nome e o celular do motorista;
- foi emitida uma ordem de transporte para o motorista informando: hora e local de partida, hora e local de chegada, e o nome e o celular de cada um dos passageiros;
- foi emitido um e-mail para o gerente do setor de transporte informando o identificador da solicitação pendente.

Exceções:

- o sistema não pôde agendar o transporte!

Fonte: elaborado pelo autor.

7.4 Diagrama de Máquina de Estados

Objetos têm estado. Eventos internos ou externos podem fazer os objetos mudarem de estado. Quando um objeto muda de estado diz-se que ele realizou uma transição de estado. Os estados e as transições de estado de um objeto constituem o seu ciclo de vida. O diagrama de máquina de Estados modela os estados dos objetos de uma classe, podendo ser usado para investigar o ciclo de vida de objetos.

A investigação do ciclo de vida de objetos está relacionada à modelagem dinâmica. Todo evento que causa uma transição de estado de um objeto deve ser tratado pelo sistema de software através de operações de classe. Um objeto em determinado estado

não poderá transitar para outro estado se não existir uma operação de classe correspondente.

7.4.1 Estado

Um estado é uma situação na vida de um objeto durante a qual ele satisfaz alguma condição ou realiza alguma atividade. O estado de um objeto é determinado por seus atributos e suas associações. Existem três tipos de estado:

- ordinário;
- inicial;
- final.

Estado ordinário representa um estado de ação em que o objeto está executando algo, ou um estado estático, no qual o objeto espera pela ocorrência de um evento. Um estado de ação geralmente é nominado com um verbo no gerúndio. Um estado ordinário não possui subestados Os estados são representados por retângulos com cantos arredondados divididos em duas seções. Na seção superior é colocado o nome do estado. Na seção inferior são colocadas as atividades internas do estado.

O estado inicial determina o ponto de partida do modelo. Só pode haver um estado inicial. Quando aplicado à modelagem de um objeto, representa o início do ciclo de vida do mesmo. É representado por um círculo hachurado. O estado final determina os pontos finais do modelo. Pode haver mais de um estado final. Quando aplicado à modelagem de um objeto representa o fim do ciclo de vida do mesmo. É representado por um círculo não hachurado envolvendo outro, menor e hachurado. A Figura 47 exibe a notação para o estado inicial, o estado ordinário e o estado final.

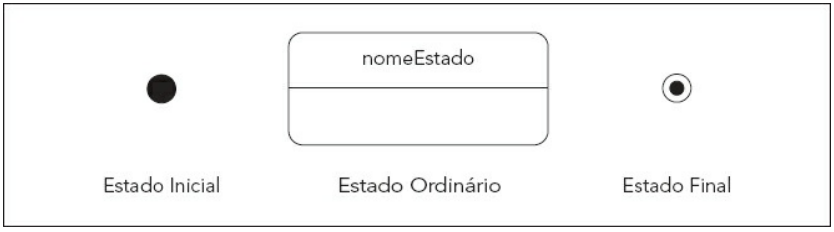


Figura 47 – Notação de estado inicial, ordinário e final.
Fonte: elaborada pelo autor.

7.4.2 Transição

Os estados estão associados a outros pelas transições. Uma transição é representada por uma seta conectando dois estados. O sentido da seta indica o sentido em que a transição ocorre. Uma transição pode ter como estado subsequente o estado original, sendo chamada de autotransição. A Figura 48 apresenta um diagrama de máquina de estados para um objeto da classe Solicitação-Transporte. O diagrama comunica que uma solicitação recém-criada assume o estado de Agendada. Depois de agendada, ela pode ser cancelada ou realizada, assumindo os respectivos estados. Uma vez cancelada ou realizada, a solicitação não poderá mais assumir outro estado. Uma solicitação agendada, mas que foi reagendada, volta a assumir o mesmo estado, caracterizando assim uma autotransição.

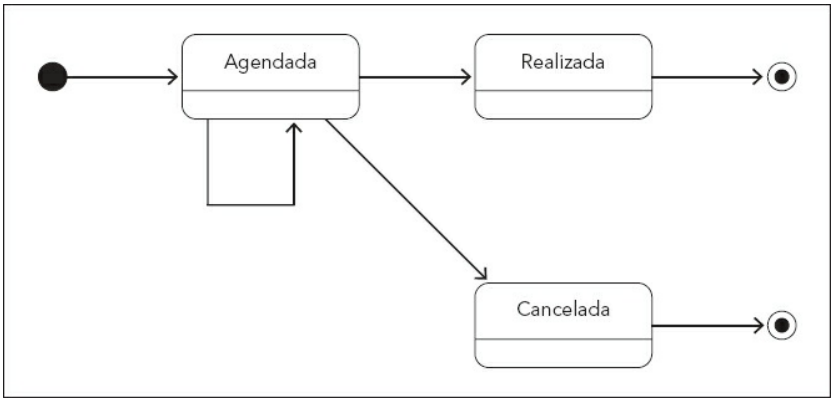


Figura 48 – Diagrama de máquina de estados de um objeto da classe SolicitacaoTransporte.
Fonte: elaborada pelo autor.

Transições podem ser rotuladas com a expressão:

evento (lista de parâmetros) [condição de guarda] / ação

Evento é algo que acontece em algum momento no tempo e que pode modificar o estado de um objeto. Os eventos podem ser dos seguintes tipos:

- evento de chamada – recebimento de uma mensagem de outro objeto;
- evento de sinal – recebimento de uma mensagem assíncrona;

- evento temporal – passagem de um intervalo de tempo predefinido;
- evento de mudança – uma condição que se torna verdadeira.

Um evento temporal é especificado com a cláusula *after*. Por exemplo, a expressão *after*(1") associada a uma transição significa que a transição será disparada um segundo após o objeto entrar no respectivo estado. Um evento de mudança é especificado com a cláusula *when*. Por exemplo, a expressão *when*(valor==0) associada a uma transição significa que a transição será disparada quando o atributo valor do objeto modelado se tornar zero.

Eventos podem ter parâmetros que serão repassados para o objeto receptor. Condição de guarda é uma expressão lógica representada entre colchetes. Uma transição, para a qual foi especificada uma condição de guarda, é disparada somente se o evento associado ocorrer e se a condição de guarda for verdadeira. Uma condição de guarda pode verificar o valor dos atributos do objeto, as associações do mesmo e o valor dos parâmetros do evento.

Ao transitar de um estado para outro, um objeto pode realizar uma ou mais ações. Uma ação é uma expressão que pode ser definida em termos dos atributos, das operações, das associações de classe ou dos parâmetros do evento. É representada na linha de transição após uma barra ("/"). Uma ação é executada somente se a transição for disparada.

O digrama da Figura 49 demonstra o uso de transições rotuladas. Este exemplo aprofunda o estudo do ciclo de vida de um objeto de SolicitacaoTransporte. Tem-se que uma solicitação recém-criada assume o estado Nova. Após o processo de agendamento automático ela pode ir para o estado Agendada. Caso o agendamento seja impossível, ela passa para Pendente e a ação notificarGerente deve ocorrer durante esta transição de estado. Uma solicitação pendente pode passar para agendada através de um agendamento manual, evento no qual uma data e hora são parâmetros. Após o transporte ser realizado, a solicitação passa para o estado Realizada. Solicitações pendentes há mais de 30 dias são automaticamente canceladas.

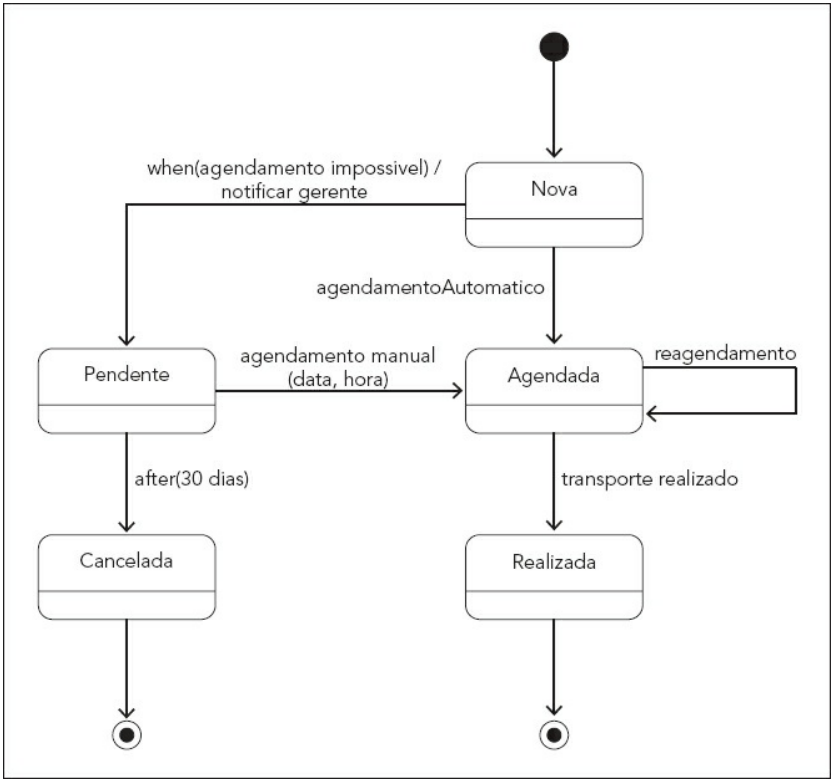


Figura 49 – Diagrama de máquina de estados com transições rotuladas.
Fonte: elaborada pelo autor.

Usa-se o recurso de notação pseudoestado de escolha quando a transição de um estado pode levar a diversos estados de acordo com a condição de guarda. Um pseudoestado de escolha é representado por um losango ou por um círculo vazado de onde partem as transições possíveis. Já o pseudoestado de junção pode ser usado para unir diversos fluxos em um único fluxo ou dividir um único fluxo em vários. Um pseudoestado de junção é representado por um círculo hachurado similar ao de estado inicial, porém um pouco menor. A Figura 50 mostra um sistema de controle de temperatura. A cada segundo – cláusula *after*(1") – o sistema avalia a temperatura e

assume um novo estado de acordo com o especificado nas condições de guarda.

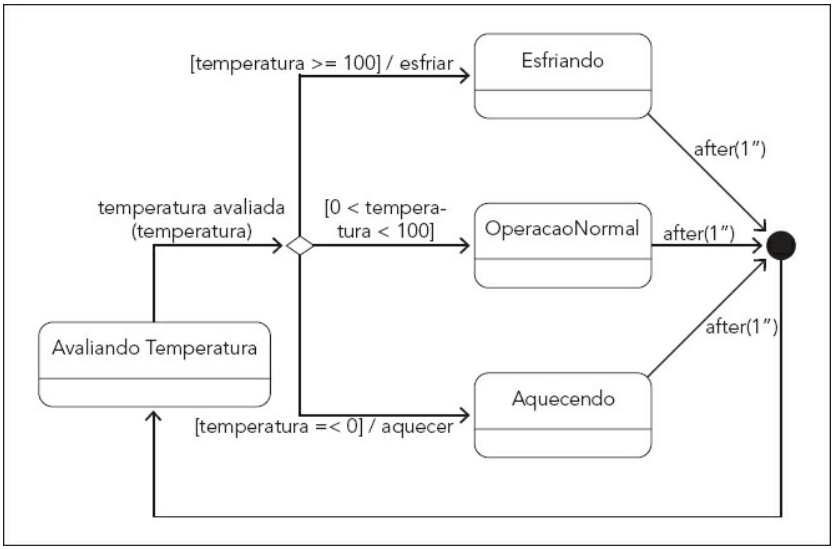


Figura 50 – Pseudoestado de escolha e de junção.
Fonte: elaborada pelo autor.

Podem existir estados aninhados dentro de outro estado. Um estado que contém outros estados é dito composto. Todos os estados dentro de um estado composto, chamados de subsestados, herdam qualquer transição deste último. A representação de um estado composto é similar ao do estado ordinário, executando-se pelo fato de que na seção inferior são representados os subestados. A Figura 51 mostra o diagrama de máquina de estados de um sistema de alarme. O sistema, enquanto no estado programado, pode estar, ao mesmo tempo, em um de três subestados: esperando, soando e soneca.

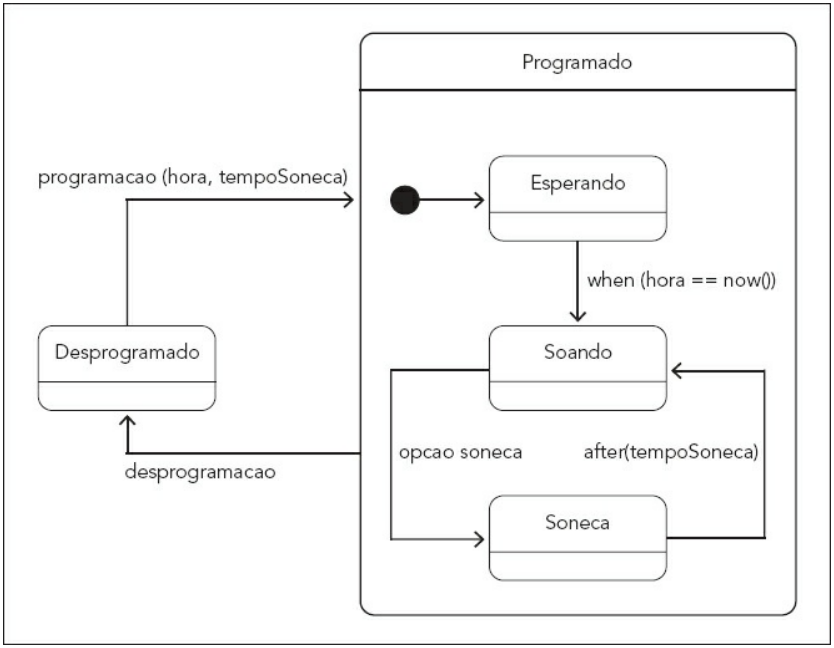


Figura 51 – Estado composto.
Fonte: elaborada pelo autor.

Estado de submáquina é equivalente a um estado composto. Entretanto, os subestados não são exibidos. Em contrapartida, é exibido no canto inferior direito um símbolo indicativo de que este estado possui subestados. A Figura 52 exibe a notação para um estado de submáquina.

Figura 52 – Estado de submáquina.
Fonte: elaborada pelo autor.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar o conhecimento sobre técnicas de modelagem funcional recomenda-se a leitura de Larman (2007) e Wazlawick (2004). Mais detalhes sobre a notação para diagramas de sequência e diagrama de máquina de estados podem ser obtidos em Guedes (2009).

RESUMO DO CAPÍTULO

Na modelagem funcional, busca-se modelar o comportamento do sistema, identificando e descrevendo as operações que o sistema tem de realizar para executar suas funções. O diagrama de sequência pode ser usado para identificar essas operações, que devem, posteriormente, ter seus contratos especificados. Outro aspecto comportamental importante que pode ser analisado é o ciclo de vida dos objetos do domínio, ou seja, seus estados e transições, que podem ser modelados empregando-se o diagrama de máquina de estados.

CAPÍTULO 8

PROJETO

Este capítulo enfoca o projeto de uma solução para o sistema. O projeto envolve a definição da arquitetura do software, que engloba arquitetura física e a arquitetura lógica, e o detalhamento técnico da solução. Arquitetura física e arquitetura lógica são definidas e os diagramas da UML que podem ser usados para representá-las são trabalhados. Assumindo-se uma arquitetura em camadas, o que é bastante comum em projetos de software, são apresentadas as atividades de detalhamento técnico da camada de domínio e do serviço de persistência da camada de serviços.

8.1 Introdução ao projeto de software

Uma vez que o sistema foi analisado e o problema relativamente bem definido, é hora de começar a projetar uma solução para este. As principais atividades da análise são a construção do modelo de domínio e a elaboração dos contratos de operações de sistema. Estes artefatos são insumo para o projeto, que visa definir a arquitetura lógica e a arquitetura física do software. O projeto engloba duas questões principais: a definição da arquitetura de software e o detalhamento técnico da solução.

8.1.1 Definição da arquitetura de software

A arquitetura de software refere-se ao conjunto de decisões significativas sobre a organização de um sistema de software, envolvendo decisões sobre a arquitetura física e a arquitetura lógica do sistema.

Por arquitetura física entende-se a organização em larga escala dos nodos computacionais e conexões de um sistema como, por exemplo, servidores e redes de comunicação. A UML provê o diagrama de implantação para representar a arquitetura física do sistema. Este diagrama será estudado posteriormente. Já por arquitetura lógica, entende-se a organização em larga escala das classes de software em pacotes, subsistemas e camadas.

Um estilo comum para arquitetura lógica de sistemas de software é o da

organização em camadas. Camada é um agrupamento de granularidade muito grossa de classes, pacotes ou subsistemas que tem responsabilidade coesiva sobre um aspecto do sistema. Camadas são organizadas de modo que as mais altas solicitem serviços às mais baixas, mas, normalmente, não ao contrário. Softwares podem ser projetados em duas, três, quatro ou mais camadas. Pode-se, contudo, falar de três camadas principais:

- interface de usuário, que é a camada mais externa/alta e que contém janelas, botões, linhas de comando para interação do usuário com o sistema;
- camada de domínio, que é a camada intermediária, onde ocorre o processamento da lógica específica do sistema;
- camada de serviços, a mais basilar, que trata de questões como persistência, acesso a outros sistemas, *logging* e *mailing*, entre outros serviços comuns em sistemas de software.

Pacote, na UML, é um mecanismo de agrupamento geral que pode ser usado para agrupar vários artefatos de um modelo, tais como classes conceituais, classes de software e casos de uso. Pode ou não equivaler ao conceito de pacote em linguagens de programação. Subsistema é um pacote que pode realizar interfaces (operações), ou seja, é um pacote executável. Geralmente, congrega um conjunto de classes.

A UML provê um diagrama para representar a arquitetura lógica em grande escala. Trata-se do diagrama de pacotes. Neste diagrama representam-se os pacotes e os subsistemas. A notação gráfica para um pacote é um retângulo com uma aba no canto superior esquerdo, equivalente ao ícone de pasta em muitos sistemas operacionais (Figura 53). Subsistemas são representados como pacotes contendo na aba o estereótipo «Subsystem» ou ícone representativo.

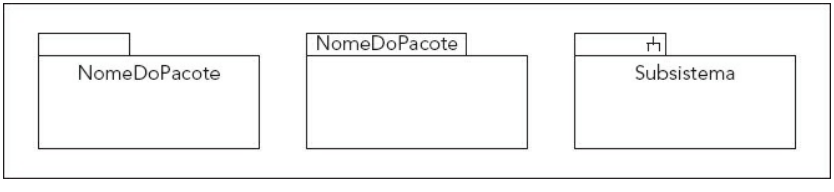


Figura 53 – Notação de pacotes e subsistemas na UML.
Fonte: elaborada pelo autor.

Os elementos constituintes de um pacote podem ser representados dentro do mesmo. É possível relacionar pacotes através de dependências. Uma dependência é representada por uma seta com a linha tracejada. A ponta da seta indica o sentido da dependência. As relações de dependência entre pacotes e subsistemas podem delimitar as camadas. A Figura 54 mostra um exemplo de Diagrama de Pacotes, evidenciando o conteúdo interno dos pacotes e a relação de dependência entre as camadas de interface, domínio e serviço.

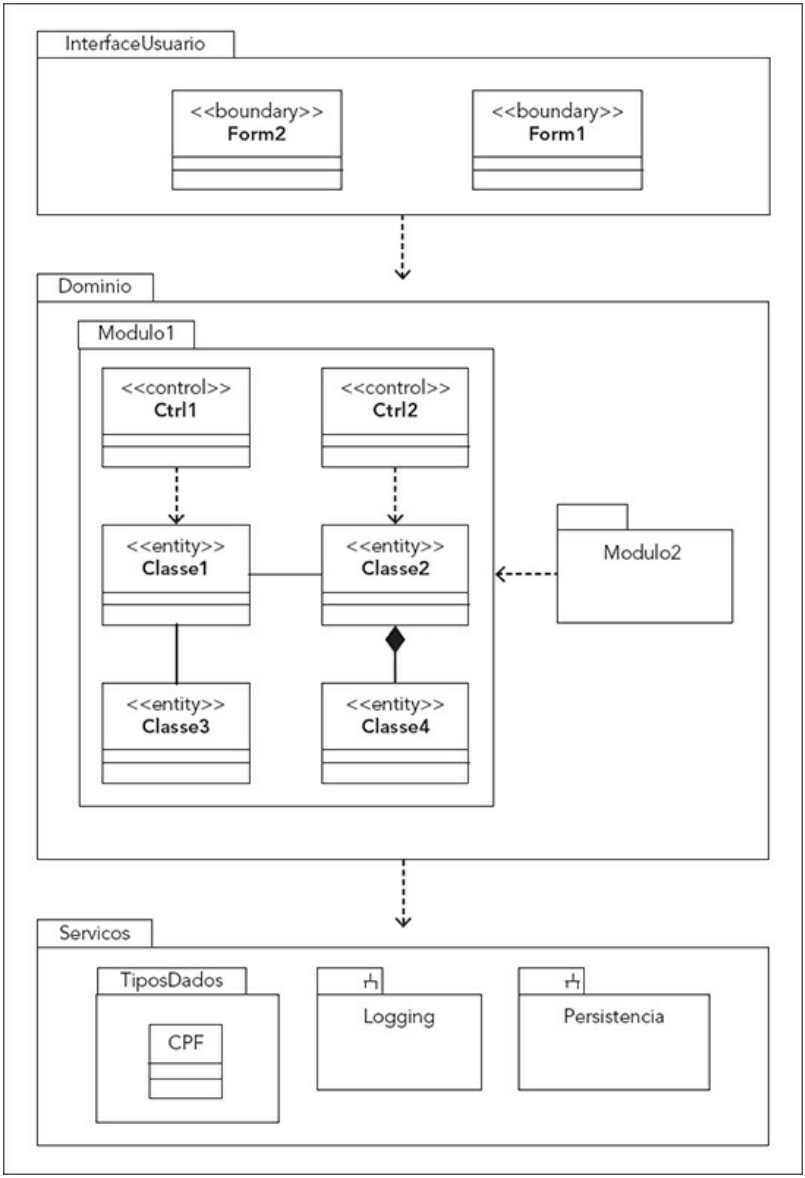

InterfaceUsuario
<<boundary>>
Form2
<<boundary>>
Form1
Dominio
Modulo1
<<control>>
Ctrl1
<<control>>
Ctrl2
<<entity>>
Classe1
<<entity>>
Classe2
Modulo2
<<entity>>
Classe3
<<entity>>
Classe4
Servicos
TiposDados
CPF
Logging
Persistencia

Figura 54 – Diagrama de pacotes.
Fonte: elaborada pelo autor.

8.1.2 Detalhamento técnico da solução

O detalhamento técnico do projeto vai depender, em parte, da arquitetura definida para o projeto. Assim, se for definido que o software será estruturado em três camadas, deverá ser feito o detalhamento técnico de cada uma delas.

Diversos diagramas da UML podem ser utilizados para representar o detalhamento técnico da solução: o diagrama de classes, já conhecido, o diagrama de comunicação, que evidencia a conexão entre objetos a partir das trocas de mensagens entre os mesmos, e o diagrama de componentes, que mostra os componentes de software, suas interfaces e as dependências entre os mesmos.

A utilização desses diagramas para o detalhamento técnico da solução será vista em seguida. Antes, contudo, é necessário compreender três conceitos importantes em projeto de software:

- visibilidade;
- acoplamento;
- coesão.

8.1.3 Visibilidade, acoplamento e coesão

Para um objeto emissor enviar uma mensagem a um objeto receptor, o receptor deve ser visível ao emissor, ou seja, o emissor deve ter algum tipo de referência ou apontamento para o objeto receptor. Em outras palavras, para um objeto A enviar uma mensagem para um objeto B, B deve ser visível para A. A visibilidade pode se dar de várias maneiras:

- visibilidade por atributo – B é um atributo de A;
- visibilidade por parâmetro – B é um parâmetro de uma operação de A;
- visibilidade local – B é um objeto local em uma operação de A;
- visibilidade global – B é globalmente visível (p. ex.: declaração de variável global).

Um software bem projetado deve ter como características um fraco acoplamento e uma forte coesão. São, portanto, objetivos do projetista diminuir o acoplamento e aumentar a coesão.

Acoplamento refere-se ao grau de conexão e interdependência entre elementos. Quanto mais os elementos forem dependentes entre si, tão maior será o acoplamento entre eles. Para diminuir o acoplamento, deve-se minimizar a quantidade de ligações de visibilidade entre as classes (por atributo, por parâmetro, local e global). O modelo de domínio tem um grau natural de acoplamento que representa o acoplamento mínimo de um sistema. Deve-se evitar a criação de novas ligações de visibilidades além das que são intrínsecas ao modelo de domínio.

A Figura 55 exemplifica a questão do acoplamento. Suponha-se que o diagrama da esquerda represente as conexões naturais de um sistema. Se, inadvertidamente, em tempo de projeto, forem criadas novas conexões entre as classes, pode-se chegar a uma nova configuração do software, como a representada no diagrama da direita. Este diagrama tem um grau de acoplamento maior que o primeiro. O problema do acoplamento alto, conforme demonstra o exemplo a seguir, é que agora qualquer modificação na Classe 1 demandará a revisão das Classes 2, 3 e 4. É certo que algumas novas conexões terão de ser criadas, mas algumas diretrizes devem ser seguidas para que não se criem conexões desnecessárias.

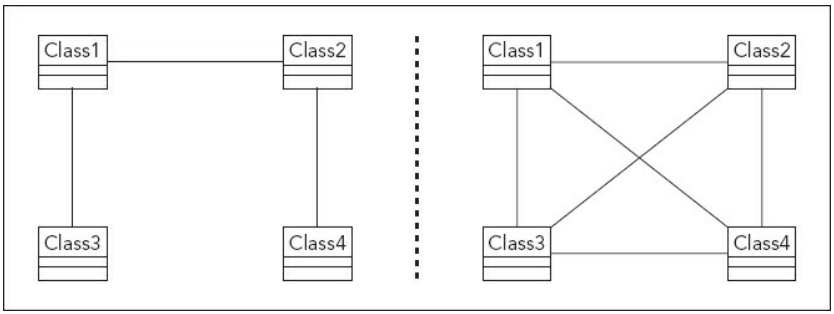


Figura 55 – Acoplamento baixo e acoplamento alto.
Fonte: elaborada pelo autor.

Coesão é a medida de quão forte e genuinamente relacionados são os elementos de um agregado. Deve-se evitar que as classes se tornem complexas demais por assumirem responsabilidades que não são suas, evitando, por exemplo, a criação de

classes que representem mais de um conceito. A Figura 56 mostra uma classe muito pouco coesa, visto que incorpora em si diversos conceitos que deveriam estar presentes em outras classes. Comparando-se a estrutura desta com a representação da mesma na Figura 41, é possível perceber a diferença no grau de acoplamento entres ambas as modelagens.

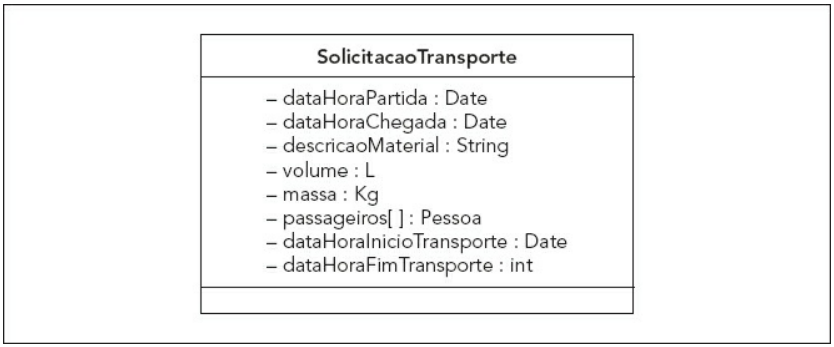


Figura 56 – Coesão baixa.
Fonte: elaborada pelo autor.

8.2 Padrões de projeto

Algumas diretrizes para um projeto de qualidade estão mapeadas sob a forma de padrões de software. Padrões descrevem problemas comuns relacionados ao desenvolvimento de software e o núcleo da sua solução. Pode-se usar o núcleo ou a ideia geral da solução para resolver diversos problemas concretos similares. O uso de padrões de projeto ajuda a melhorar a qualidade do projeto do software, diminuindo o acoplamento e aumentando a coesão. Além disso, acelera e qualifica o desenvolvimento de um produto de software através da reutilização de soluções amplamente experimentadas e sabidamente funcionais e eficientes.

Existem diversas categorias de padrões, tais como padrões arquiteturais (FOWLER, 2006), padrões de projeto (GAMMA, 2000; BRAUDE, 2005) e idiomas. Padrões arquiteturais são padrões para serem utilizados no esboço inicial do projeto do software, especificando as estruturas fundamentais da aplicação (arquitetura lógica). Padrões de projeto (*design patterns*) são utilizados na fase final do esboço, refinando ou estendendo a arquitetura fundamental do software. São aplicados na

etapa do detalhamento do projeto. Idiomas são utilizados na etapa de implementação, para escrever um padrão de projeto em uma linguagem específica.

Os padrões mais difundidos são os de projeto. Existe uma infinidade deles. Está fora do escopo deste livro aprofundar o estudo em particular dos vários padrões de projeto. É apresentada apenas uma visão geral, de forma que o conceito seja entendido, possibilitando que, na medida do necessário, se busque e se aplique padrões adequados na resolução de problemas de projeto.

Neste livro destacam-se os padrões GRASP (*General Responsibility Assignment Software Patterns*), padrões GoF (*Gang of Four*) e o padrão MVC (*Model-View-Controller*).

8.2.1 Padrões GRASP

Os padrões GRASP são utilizados para definir a atribuição de responsabilidades das classes. O Quadro 8 apresenta uma descrição sucinta do problema que cada um dos nove padrões GRASP se propõe a resolver e a respectiva solução.

Quadro 8 – Padrões GRASP (LARMAN, 2007)

Padrão	Problema	Solução
Especialista na informação	Qual é o princípio geral para atribuição de responsabilidades a objetos?	Atribuir a responsabilidade ao especialista na informação, ou seja, a classe que tem a informação necessária para satisfazer a responsabilidade.
Criador	Quem deve ser o responsável pela criação de uma nova instância de uma classe?	Atribuir à classe B a responsabilidade de criar uma instância da classe A se uma das seguintes condições for verdadeira: → B contém A ou agrega A por composição; → B registra A; → B usa A de maneira muito próxima; → B tem os dados iniciais de A, que serão passados para A quando este for criado.
Acoplamento baixo	Como apoiar dependência baixa, baixo impacto de modificação e aumento de reuso?	Atribuir responsabilidades de modo que o acoplamento permaneça baixo.
		Atribuir a responsabilidade a uma classe que

Controlador	Qual é o primeiro objeto, além da camada de IU que recebe e coordena uma operação de sistema?	representa uma das seguintes escolhas: ➔ o sistema ou ➔ um cenário de um caso de uso do qual ocorre o evento do sistema.
Coesão alta	Como manter os objetos bem focados, inteligíveis e gerenciáveis e, como efeito colateral, apoiar o acoplamento baixo?	Atribuir uma responsabilidade de forma que a coesão permaneça alta. Uma classe com coesão baixa faz muitas coisas não relacionadas.
Polimorfismo	Como tratar alternativas com base no tipo? Como criar componentes de software interconectáveis?	Quando alternativas ou comportamentos variam segundo o tipo (classe), deve-se atribuir a responsabilidade pelo comportamento aos tipos para os quais o comportamento varia, usando operações polimórficas.
Fabricação	Qual objeto deve ter a responsabilidade quando não se quer violar a coesão alta e o acoplamento baixo, mas as soluções oferecidas pelo especialista não são apropriadas?	Atribuir um conjunto de responsabilidades altamente coeso a uma classe artificial ou de conveniência que não represente um conceito do domínio do problema – algo inventado.
Indireção	A quem se deve atribuir a responsabilidade de maneira a evitar acoplamento direto entre dois ou mais objetos? Como desacoplar os objetos, de modo que o acoplamento baixo seja apoiado e o potencial de reuso permaneça mais alto?	Atribuir a responsabilidade de ser o mediador entre outros componentes ou serviços a um objeto intermediário, para que eles não sejam diretamente acoplados. O intermediário cria uma indireção entre os outros componentes.
Variações protegidas	Como projetar objetos, subsistemas e sistemas de modo que as variações ou as instabilidades nesses elementos não tenham um impacto indesejável sobre outros elementos?	Identificar pontos de variação ou instabilidade previsível e atribuir responsabilidades para criar uma interface estável em torno deles.

Fonte: elaborado pelo autor.

8.2.2 Padrões GOF

GOF é acrônimo para "*Gang of Four*", fazendo menção aos quatro autores de uma obra de referência sobre padrões de projeto: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Segundo a obra, os padrões de projeto podem ser classificados quanto ao propósito e quanto ao escopo. O Quadro 9 mostra os padrões GOF segundo suas categorizações. Quanto ao propósito, ou seja, o que o padrão faz, podem ser categorizados em de criação, estruturais e comportamentais. Os padrões de criação se encarregam do processo de criação de objetos, os estruturais definem como será composta a classe ou objeto e os comportamentais caracterizam como será a interação ou responsabilidade das classes ou objetos. Já quanto ao escopo, ou seja, se

o padrão se aplica à classe ou ao objeto, podem ser categorizados em de classe, que estabelecem relacionamentos entre classes, e de objeto, que estabelecem relacionamentos entre objetos.

Quadro 9 – Padrões GOF (GAMMA et al., 2000)

		Propósito		
		De Criação	Estrutural	Comportamental
Escopo	Classe			Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: adaptado pelo autor.

Gamma (2000) exemplifica a utilização dos padrões no projeto de um editor de texto com interface WYSIWYG (*What You See Is What You Get*). O desenvolvimento de um editor traz consigo diversos desafios de projeto que precisam ser superados. Para resolver estes problemas, uma boa ideia é não "reinventar a roda" e avaliar a aplicação de padrões de projetos. Problemas comuns em projetos de editores de texto poderiam ser resolvidos pela aplicação dos seguintes padrões de projeto:

- Composite – para representar a estrutura física do documento;
- Strategy – para permitir diferentes algoritmos de formatação;
- Decorator – para adornar a interface do usuário;
- Abstract factory – para suportar múltiplos padrões de interação;
- Bridge – para permitir múltiplas plataformas de sistemas de janela;

- Command – para desfazer operações de usuário;
- Iterator – para acessar e percorrer estruturas de objetos;
- Visitor – para permitir um número qualquer de capacidades analíticas sem complicar a implementação da estrutura do documento.

Para que os padrões possam ser utilizados, eles precisam ser descritos em um formato padrão. Um padrão de documentação, contendo os tópicos descritos no Quadro 10, foi sugerido por Gamma (2000).

Quadro 10 – Padrão de documentação de um projeto (GAMMA, 2000)

Nome e classificação do padrão	Nome deve ser sucinto e significativo. Classificação conforme apresentado no Quadro 9.
Intenção e objetivo	O que faz? Quais os princípios e a intenção? Que tópico ou problema trata? Essa descrição deve ser curta.
Também conhecido como	Relacionar outros nomes se existir.
Motivação	Apresentar um cenário que ilustra o problema e como as estruturas de classe e objetos solucionam o problema.
Aplicabilidade	Descrever quais as situações em que o padrão pode ser aplicado. Descrever como se pode reconhecer essas situações.
Estrutura	Mostrar uma representação gráfica das classes que compõem o padrão (UML).
Participantes	Classes e objetos que participam no padrão e suas responsabilidades.
Colaboração	Como os participantes colaboram para executar suas tarefas.
Consequências	Informar como o padrão executa seus objetivos, quais os custos, benefícios e resultados são decorrentes de sua utilização.
Implementação	O que é necessário conhecer para implementar o padrão. Sugestões ou técnicas. Considerações específicas sobre uma linguagem.
Exemplo de código	Partes do código que ilustram a implementação.
Usos conhecidos	Exemplos já implementados em outros sistemas.
Padrões relacionados	Quais padrões estão relacionados com este. Quais as diferenças importantes entre eles.

Fonte: elaborado pelo autor.

8.2.3 Padrão MVC

O padrão Modelo-Visão-Controlador (MVC) foi originalmente um padrão de pequena escala de SmallTalk-80 para tratar com objetos de dados relacionados (modelos), dispositivos de interface gráfica de usuário (visões) e manipuladores de eventos do mouse e teclado (controladores). Recentemente, o termo MVC foi adotado também para o nível arquitetural de larga escala. O modelo refere-se à camada de domínio. A visão refere-se aos componentes visuais da camada de interface gráfica de usuário (IGU) e os controladores são os objetos de fluxo de trabalho desta mesma camada. O princípio da separação Modelo-Visão estabelece que o modelo de objetos (domínio) não deve ter conhecimento direto dos objetos da visão.

8.3 Projeto da camada de domínio

O projeto da camada de domínio é a mais fundamental das atividades de projeto. Ele envolve dois trabalhos principais: projetar a colaboração de objetos para realizar contratos de operações de sistema e elaborar o diagrama de classes do projeto (DCP).

8.3.1 Projeto de colaboração de objetos

O projeto de colaborações visa identificar e modelar as trocas de mensagens entre objetos para realizar os contratos das operações de sistema mapeadas. As mensagens trocadas darão origem às operações de classes. Para realizar este trabalho, podem-se usar dois tipos de diagramas de interação da UML: o diagrama de comunicação e o diagrama de sequência.

O diagrama de sequência facilita a observação da ordem temporal das trocas de mensagens, enquanto o diagrama de comunicação facilita a observação de como os objetos estão estaticamente conectados. Ambos podem ser usados para projetar a colaboração entre os objetos. Entretanto, como o diagrama de comunicação foca nas conexões estáticas entre objetos, evidenciando o acoplamento entre os mesmos, ele torna-se mais vantajoso para esta atividade, visto que o projetista precisa estar atento em minimizar o acoplamento. Os principais elementos gráficos do diagrama de comunicação são exibidos na Figura 57.

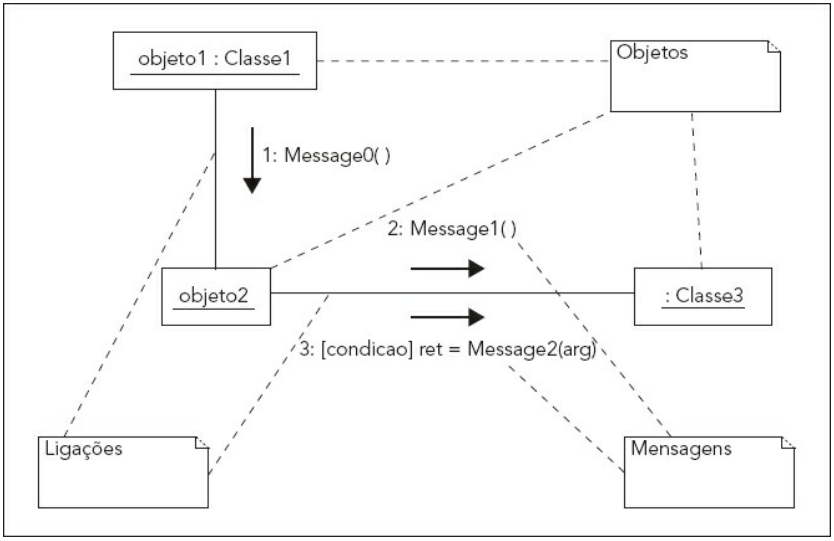


Figura 57 – Diagrama de comunicação da UML.
Fonte: elaborada pelo autor.

As mensagens em uma colaboração podem ser de dois tipos: básicas e delegadas. As mensagens básicas são aquelas para as quais a própria classe tem a responsabilidade de implementar uma operação. As delegadas são as que devem ser repassadas para outra classe. As mensagens básicas são as seguintes:

- criação de instância;
- modificação de atributo;
- consulta de atributo;
- criação de associação;
- consulta de associação;
- destruição de associação;
- destruição de instância.

A Figura 58 mostra a representação das mensagens de criação de instância,

modificação de atributo, consulta de atributo e destruição de instância.

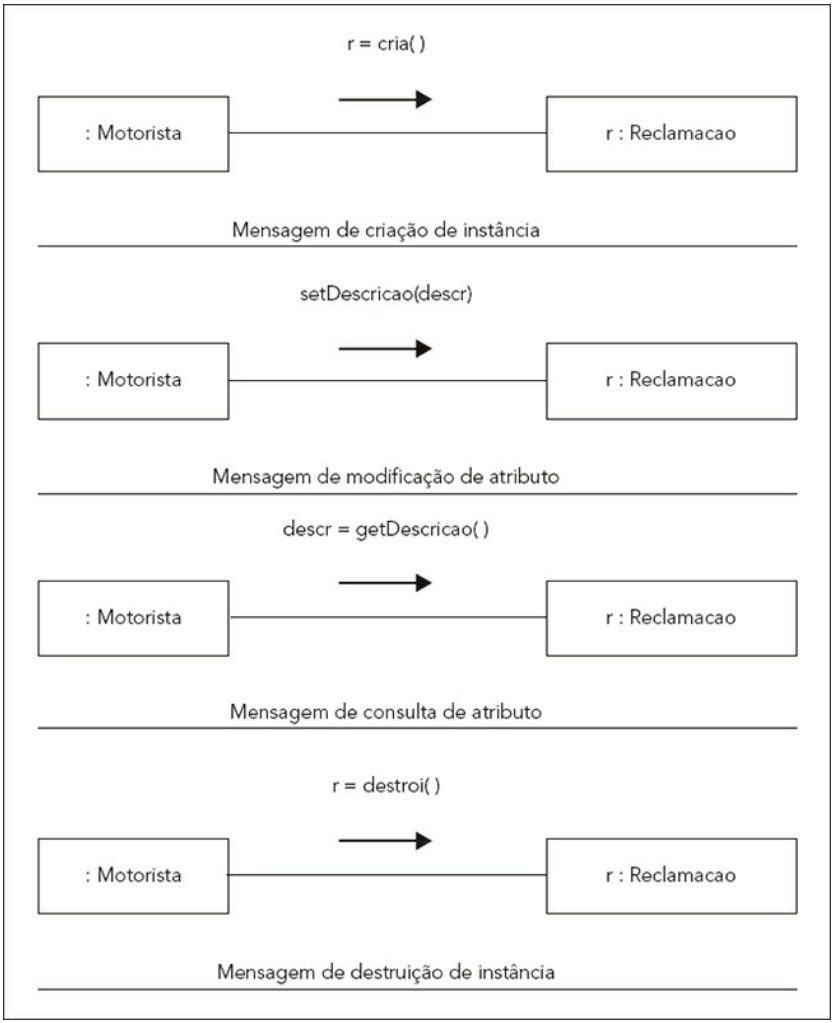


Figura 58 Mensagens de criação e destruição de instâncias e de consulta e modificação de atributos.
Fonte: elaborada pelo autor.

As mensagens para criação, consulta e destruição de associações entre instâncias variam conforme a multiplicidade da associação. A Figura 59 mostra estas mensagens para associações com multiplicidade de uma para nenhuma ou uma instância (0..1), enquanto a Figura 60 mostra as mensagens para associações com multiplicidade de uma para muitas instâncias (*). Como base para os exemplos, considerou-se as associações entre Colaborador e Solicitante (0..1) e entre Solicitante e SolicitacoesTransporte (*) da Figura 41.

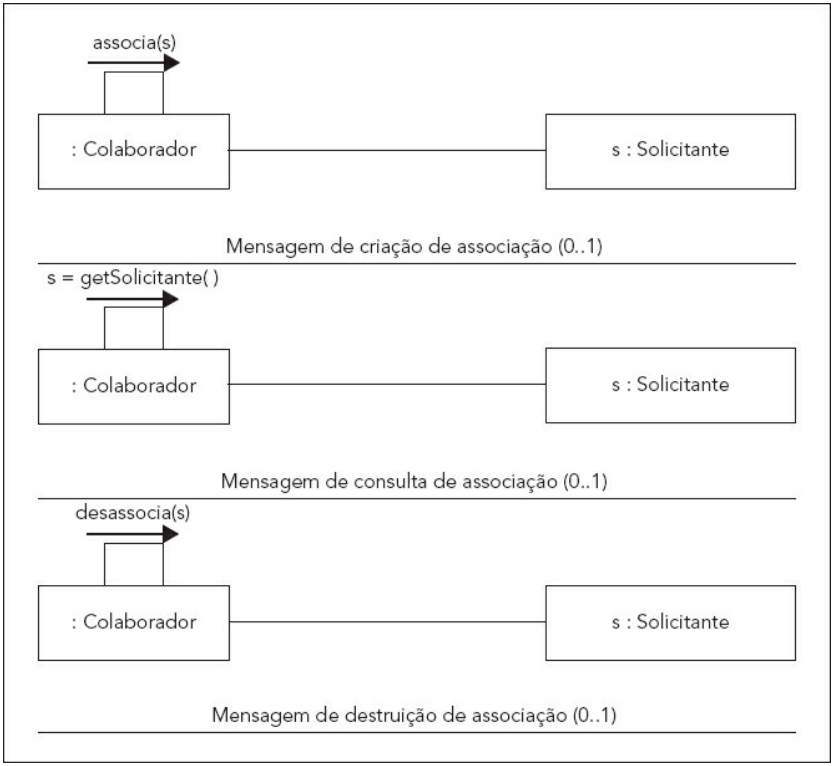


Figura 59 – Mensagens para criação, consulta e destruição de associações (0..1).
Fonte: elaborada pelo autor.

A Figura 61 apresenta o diagrama de comunicação para a operação

agendaTransporte() e exemplifica o uso de mensagens delegadas. Deve-se passar adiante a responsabilidade de realizar uma operação quando o objeto que detém o fluxo de execução não possui visibilidade direta para o objeto que deve executar a operação. Ao delegar uma mensagem, as linhas de visibilidade devem ser respeitadas.

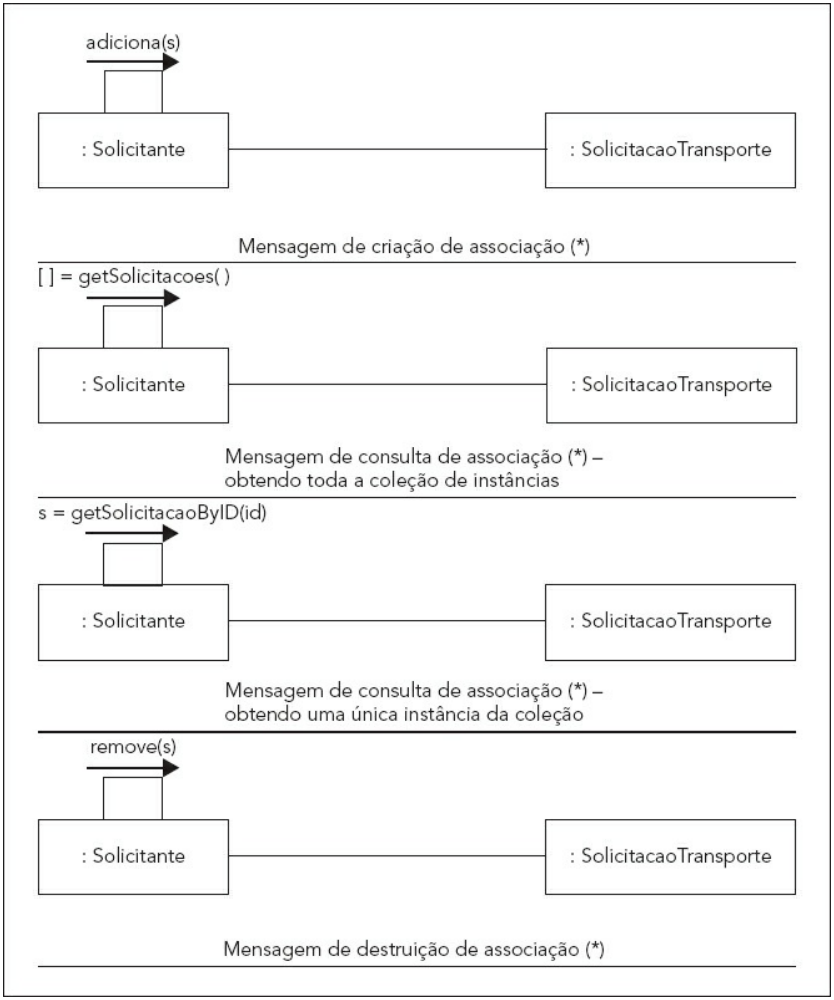


Figura 60 – Mensagens para criação, consulta e destruição de associações (*).
Fonte: elaborada pelo autor.

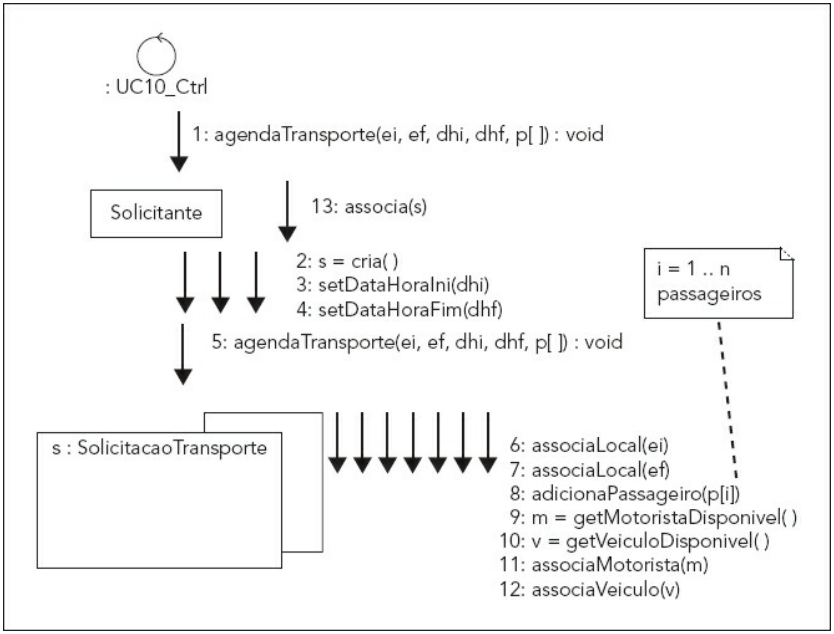


Figura 61 – Diagrama de comunicação da operação agendaTransporte().
Fonte: elaborada pelo autor.

8.3.2 Elaboração do diagrama de classes do projeto (DCP)

A elaboração do diagrama de classes de projeto toma por base o modelo de domínio desenvolvido na análise. Neste, diversas modificações são realizadas de forma a acrescentar detalhes importantes para a implementação. O modelo que antes era meramente conceitual passa a conter detalhes técnicos que vão permitir a implementação do mesmo em uma linguagem de programação. Entre os detalhamentos que podem ser feitos estão:

- adição/alteração de classes do mundo computacional, inclusive as derivadas de

aplicação de padrões de projeto;

- adição de operações de classe identificadas no projeto de colaborações;
- adição, alteração ou detalhamento de atributos de classes, especificando tipos, valores iniciais, visibilidade e restrições;
- adição de relacionamentos de dependência conforme as conexões evidenciadas pelo projeto de colaborações;
- adição de navegabilidade nas associações, conforme a visibilidade evidenciada pelo projeto de colaborações.

A Figura 62 apresenta o diagrama de classes do projeto já contendo as operações delegadas e outras operações de classe identificadas na análise de colaboração para realização da operação de sistema agendaTransporte(). As mensagens básicas não foram representadas, pois são predefinidas e sua representação pode poluir visualmente o modelo. Por simplificação, os parâmetros das operações de classe foram omitidos. Como todas as mensagens se deram sobre linhas de visibilidade já existentes, nenhum relacionamento de dependência foi adicionado, exceto o entre as classe UC10_Ctrl e Solicitante.

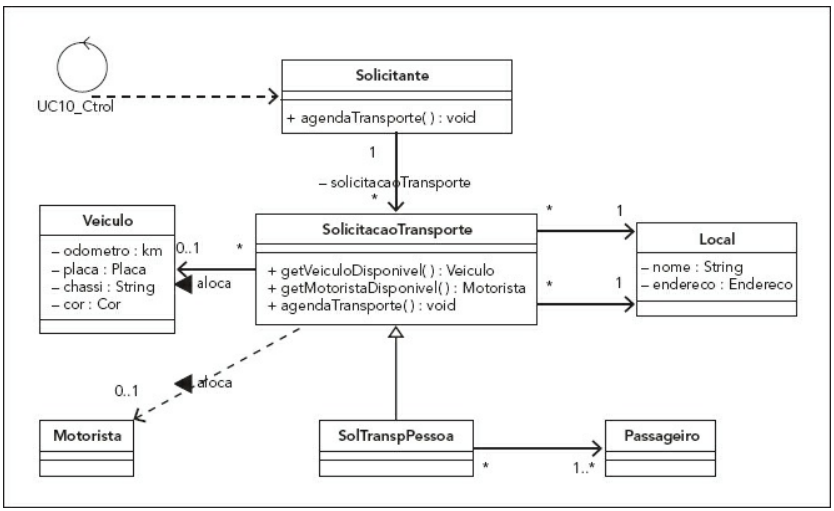


Figura 62 – Diagrama de classes do projeto (DCP) baseado no diagrama de comunicação da operação

agendaTranspote ().
Fonte: elaborada pelo autor.

8.4 Projeto da camada de serviços

Um determinado conjunto de funcionalidades está presente na maioria dos softwares. Essas funcionalidades não se referem a um domínio específico, tendo geralmente uma implementação padrão, acessível por uma interface pública e bem definida. Por exemplo, muitos softwares precisam registrar a execução de suas operações, acessar um serviço de email ou um serviço de autenticação, armazenar e recuperar informações em bancos de dados. Essas funcionalidades podem compor uma camada de servios, que responde a requisições da camada de domínio. Dentre os serviços mais comuns e necessários está o de persistência.

8.4.1 Serviço de persistência

Linguagens de programação orientadas a objetos lidam apenas com objetos essencialmente transientes, ou seja, residentes em memória. Contudo, às vezes há a necessidade de usar objetos criados durante a execução de uma aplicação em execuções futuras da mesma, ou ainda, em outras aplicações. Neste ponto, entra o serviço de persistência que tem a função de garantir que os objetos sejam salvos em um dispositivo de memória secundária (sistema de arquivos ou SGBDs OO, relacionais ou híbrids), e que de lá sejam recarregados quando necessário.

Em uma arquitetura em camadas, operações e consultas da camada de interface com usuário são realizadas por meio da camada de domínio. Operações da camada de domínio que necessitam criar, consultar, alterar ou excluir objetos armazenados em memória secundária não o fazem diretamente, mas através de requisições ao serviço de persistência.

O serviço de persistência tem um aspecto estrutural e um comportamental. O primeiro trata da estrutura para o armazenamento dos dados (p. ex.: mapeamento OO-ER). O segundo trata do salvamento e carga de objetos. Para esta função existem duas abordagens possíveis: inserir nos pontos adequados o código necessário para salvar e carregar objetos, ou controlar o salvamento e carregamento de objetos usando a própria arquitetura do sistema. Esta segunda alternativa demanda a utilização de um *framework* de persistência.

O projeto da camada de persistência pode ser facilitado pela utilização de um

framework para persistência. *Frameworks* possibilitam a geração automática de uma camada de persistência a partir da camada de domínio. Pode-se construir ou adotar um *framework*. Atualmente, muitos *frameworks*, comerciais ou livres, estão disponíveis. Dentre eles, destacam o Hibernate e OJB – ObJectRelationalBridge.

8.4.2 Mapeamento OO-ER

Uma questão muito presente quando se trata de serviço de persistência é a do mapeamento OO–ER. A questão vem do fato de que a maioria dos SGBDs utilizados em projetos de software orientados a objetos são relacionais. Como existe uma diferença semântica natural entre o modelo OO e o modelo baseado em tabelas de um banco de dados relacional, se faz necessário um mecanismo de mapeamento entre os dois modelos.

8.4.2.1 Mapeamento de classes e de atributos

Normalmente uma classe conceitual do DCP é mapeada para uma tabela relacional. Cada instância de uma classe corresponde a uma linha na respectiva tabela. É possível mapear uma classe para mais de uma tabela. Classes temporárias não são transformadas em tabelas relacionais. Classes controladoras podem ou não ser transformadas em tabelas, dependendo se possuem ou não atributos.

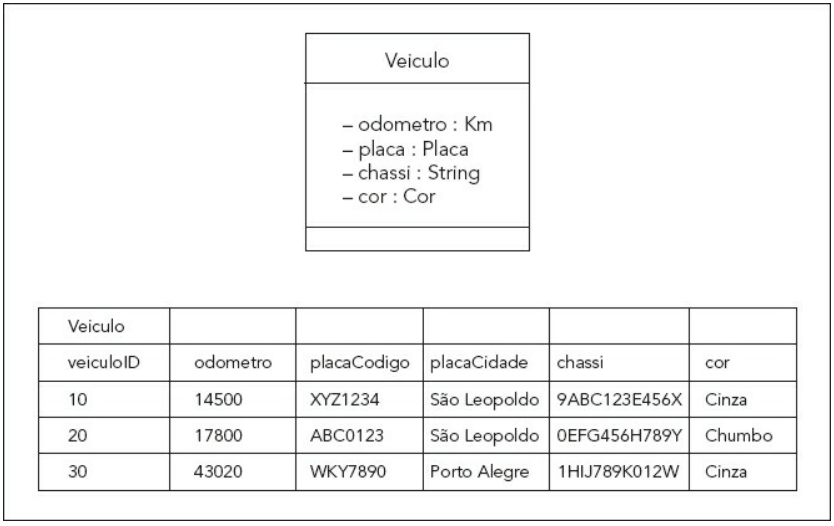
Cada atributo de uma classe pode ser mapeado para nenhuma, uma ou muitas colunas. Atributos sem estrutura são mapeados para uma coluna. Atributos com estrutura podem ser mapeados para *n* colunas (p. ex.: endereço). Atributos derivados não são mapeados para colunas.

Cada objeto persistente deve receber um atributo de identificação, sem qualquer significado para a camada de domínio. Este atributo será chave primária da tabela que armazena o objeto. Nomes usuais para este tipo de atributo são ID, PID (Persistence ID) ou OID (Object ID). Não devem ser usados para este fim os atributos que naturalmente identificam os objetos, tais como códigos e outros identificadores. Isto porque estes atributos variam em tipo e estrutura, o que dificulta um tratamento padronizado para a persistência. A Figura 63 exibe o mapeamento da classe Veiculo.

8.4.2.2 Mapeamento de relacionamentos

Relacionamentos de dependência não são mapeados para persistência.

Relacionamentos de generalização/especialização serão estudados posteriormente. O mapeamento de relacionamentos de associação entre classes varia segundo a multiplicidade da associação. Associações com classes controladoras não são transformadas em tabelas.

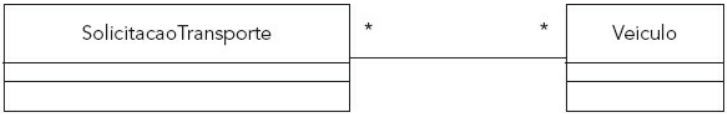


Veiculo					
veiculoID	odometro	placaCodigo	placaCidade	chassi	cor
10	14500	XYZ1234	São Leopoldo	9ABC123E456X	Cinza
20	17800	ABC0123	São Leopoldo	0EFG456H789Y	Chumbo
30	43020	WKY7890	Porto Alegre	1HIJ789K012W	Cinza

Figura 63 – Mapeamento de classe e atributos para tabela e colunas.
Fonte: elaborada pelo autor.

*8.4.2.3 Mapeamento de relacionamentos de associações * para **

Para associações com multiplicidade * para * cria-se uma tabela associativa com os IDs das tabelas relacionadas. A Figura 64 mostra um exemplo, assumindo-se uma modelagem ligeiramente diferente da apresentada na Figura 41 para a associação entre a classe Veiculo e a classe SolicitacaoTransporte. No caso do exemplo assumiu-se que uma solicitação de transporte pode ser atendida por mais de um veículo.

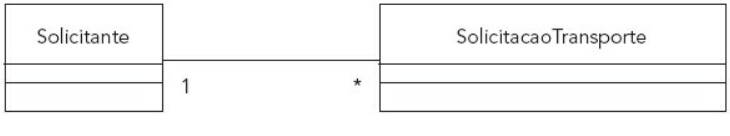


SolicitacaoTransporte_Veiculo	
solicitacaoTransporteID	veiculoID
100	10
200	20
300	10
300	20
400	10

Figura 64 – Mapeamento de associações com multiplicidade * para *.
Fonte: elaborada pelo autor.

*8.4.2.4 Mapeamento de relacionamentos de associações 1 para **

Para associações com multiplicidade 1 para * cria-se uma tabela associativa com os IDs das tabelas relacionadas. A coluna da tabela associativa correspondente ao ID da classe do lado * será do tipo *Unique Key* (UK). A Figura 65 apresenta um exemplo.



Solicitante_SolicitacaoTransporte	
solicitanteID	solicitacaoTransporteID (UK)
10	100
20	200
10	300
20	400
10	500

Figura 65 – Mapeamento de associações com multiplicidade de 1 para *.
Fonte: elaborada pelo autor.

8.4.2.5 Mapeamento de relacionamentos de associações 1 para 1, 1 para 0..1 e 0..1 para 0..1

Para associações com multiplicidade 1 para 1, 1 para 0..1 ou 0..1 para 0..1 cria-se uma tabela associativa com os IDs das tabelas relacionadas. Ambas as colunas da tabela serão do tipo *Unique Key* (UK). Alternativamente, pode-se usar chave estrangeira (*Foreing Key* – FK) em uma das tabelas associadas. Esta forma, contudo, é menos recomendável. A Figura 66 traz um exemplo da primeira alternativa e a Figura 67 da segunda.

Colaborador_Solicitante	
colaboradorID (UK)	solicitanteID (UK)
100	10
200	*null*
300	20
400	*null*
500	*null*

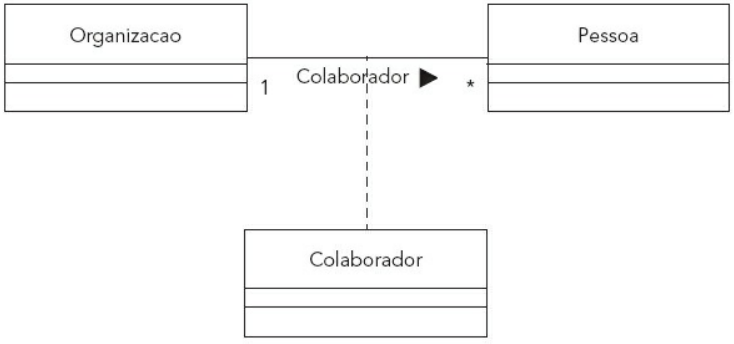
Figura 66 – Mapeamento de associação com multiplicidades 1 para 1, 1 para 0..1 ou 0..1 para 0..1.
Fonte: elaborada pelo autor.

Colaborador		
colaboradorID (UK)	outras colunas colaborador	solicitanteID (UK)
100		10
200		*null*
300		20
400		*null*
500		*null*

Figura 67 – Mapeamento de associação com multiplicidades 1 para 1, 1 para 0..1 ou 0..1 para 0..1 com FK.
Fonte: elaborada pelo autor.

8.4.2.6 Mapeamento de relacionamentos de classes de associação

Para mapear relacionamentos que envolvam uma classe de associação, deve-se criar uma tabela com colunas mapeando os atributos da classe de associação e com colunas para mapear a associação. Como a classe de associação pode ter suas próprias associações, é necessário criar um atributo ID para suas próprias instâncias. A Figura 68 apresenta o mapeamento de uma classe associativa.



Colaborador			
colaboradorID	pessoaIID	organizaçãoID	outras colunas colaborador...
100	10	1	
200	20	1	
300	40	1	
400	70	1	
500	90	1	

Figura 68 – Mapeamento de classe associativa.
Fonte: elaborada pelo autor.

8.4.2.7 Mapeamento de relacionamentos de generalização/especialização

O mapeamento de relacionamentos de generalização/especialização pode se dar de três formas alternativas:

- partição vertical;
- partição horizontal;
- partição única.

A Figura 69 apresenta as três possibilidades. Os exemplos consideram o diagrama base demonstrado na mesma figura. Com a estratégia da partição vertical, cada classe da estrutura de generalização/especialização é mapeada para uma tabela, cada qual com seu ID. As tabelas referentes às classes especializadas devem conter o ID correspondente ao da tabela referente à classe mais geral. Na partição horizontal, cada ramo da estrutura de generalização/especialização transforma-se em uma tabela, que deverá conter todos os atributos das classes do respectivo ramo. No caso da partição única, todos os atributos da estrutura são atribuídos a uma única tabela. Os atributos específicos de classes especializadas devem ser mapeados para colunas que possam conter valor nulo. Um atributo adicional deve ser criado para informar a qual classe determinada dupla da tabela pertence.

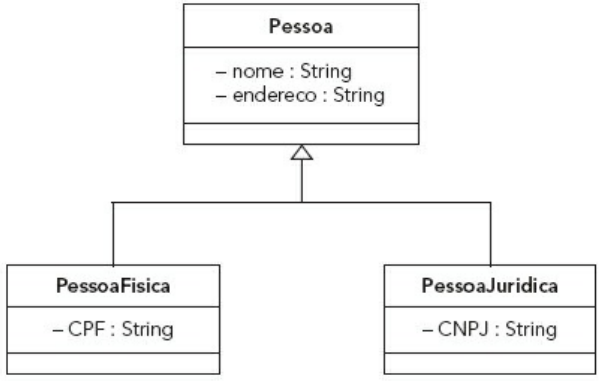


Diagrama base

Pessoa
pessoaID
nome
endereco

PessoaFisica
pessoaFisicaID
cpf
pessoaID

PessoaJuridica
pessoaJuridicaID
cgc
pessoaID

Partição Vertical

PessoaFisica
pessoaFisicaID
nome
endereco
cpf

PessoaJuridica
pessoaJuridicaID
nome
endereco
cgc

Partição Horizontal

Pessoa
pessoaID
nome
endereco
cpf (*null*)
cgc (*null*)
tipo {FISICA, JURIDICA}

Partição Única

Figura 69 – Mapeamento de relacionamentos de generalização/especialização.
Fonte: elaborada pelo autor.

8.5 Projeto da camada de interface com usuário

A camada de interface com usuário pode ser dividida em duas subcamadas. Uma, chamada de camada de apresentação, contém classes referentes aos elementos gráficos da interface e tem por função receber dados e instruções do usuário e apresentar informações ao mesmo. A outra, chamada de camada de aplicação ou de camada da lógica de apresentação, tem por responsabilidade controlar a operação da interface.

O primeiro passo para o projeto da camada de interface com usuário é verificar os requisitos não-funcionais que dizem respeito à interface com usuário e definir os dispositivos de entrada e saída. Devem ser considerados os dispositivos de entrada (leitoras de barras, cartões, RFID, sensores, câmeras e dispositivos biométricos) e os dispositivos de saída (monitores, impressoras e atuadores). Além disso, é preciso fazer definições a respeito do estilo de interface, como, por exemplo, se de linha de comando ou de janelas, e neste último caso, se utilizando cliente rico (cliente-servidor) ou cliente pobre (Web).

No caso de um sistema baseado em janelas, o próximo passo é definir as janelas do sistema. Para tal, deve-se analisar os casos de uso. Em princípio, a cada caso de uso corresponderá uma janela e uma classe controladora. As primeiras poderão fazer parte da subcamada de apresentação. As classes controladoras poderão compor a subcamada de aplicação. Analisando-se os casos de uso é possível definir os elementos de entrada de dados, de saída de resultados e de operação da interface.

Os leiautes das janelas podem ser desenhados através de *wireframes* (desenho estilizado dos contornos dos elementos que compõem uma interface gráfica) e adicionados à especificação do caso de uso. Um aspecto importante, principalmente em sistemas Web, é definir as possibilidades de navegação entre as janelas. A UML não dispõe de um diagrama específico para este fim, mas o diagrama de estados pode cumprir esta função.

O projeto gráfico das janelas e do sistema como um todo é uma atividade ampla e multidisciplinar, que vai muito além do escopo deste livro. Contudo, alguns pontos serão destacados. Todo o sistema deve compartilhar de um mesmo projeto gráfico. Assim, cores, formatos de campos, rótulos, mensagens, agrupamento de elementos nas interfaces, estilos de navegação e de operação, entre outros diversos elementos, devem ser padronizados.

A aceitação da interface depende MAIS da qualidade de suporte a algumas tarefas e MENOS da quantidade de funções suportadas. Utilidade é a adequação das funções do sistema às tarefas do usuário. Usabilidade é a adequação do suporte que o sistema fornece às tarefas do usuário. Algumas recomendações podem ser seguidas no projeto de interfaces:

- delimitar regiões para os elementos gráficos (menus, mensagens de status, etc.);
- agrupar elementos relacionados, distanciar elementos distintos;
- pedir informações em uma sequência lógica;
- destacar o foco corrente na tela;
- projetar interfaces objetivas – não desviar atenção do foco;
- projetar interfaces simples – poucos objetos e pouca densidade;
- evidenciar ao usuário o tipo de erro que ele cometeu e orientar sobre o caminho a seguir;
- oferecer ao usuário um modo de cancelar e reiniciar uma transação;
- oferecer um mecanismo de ajuda (*help*);
- oferecer um mecanismo de perdão (*undo*);
- prover valores *default* nas entradas;
- oferecer uma forma de acompanhar o progresso de uma transação (p. ex.: barra de progresso);
- usar sons e cores com parcimônia;
- projetar interfaces WYSIWYG.

Outra questão que pode ser considerada no projeto da camada de interface é a da segurança de acesso. A segurança tem de ser garantida pela camada de domínio. Entretanto, é conveniente, dentro do possível, controlar o acesso também via interface. O objetivo é evitar exceções de segurança. É preferível não permitir que o usuário chegue à tela de inclusão, a dar uma mensagem de erro depois de ele ter informado todos os dados e confirmado a operação. A segurança pode ser definida em quatro níveis:

- acesso às janelas – por exemplo, o ator pode acessar a janela de cadastro de clientes;
- acesso a operações – por exemplo, embora o ator tenha acesso à janela de cadastro de clientes, ele somente poderá consultar clientes;
- acesso a atributos – por exemplo, embora o ator tenha acesso à janela de cadastro de clientes para consulta, ele poderá ver o nome e o endereço, mas não poderá ver a remuneração;
- acesso a instâncias – por exemplo, o ator somente consegue consultar os clientes da sua loja.

8.6 Componentes

Componente é uma unidade de software que pode ser utilizada na construção de vários sistemas e que pode ser substituída por outra unidade que tenha a mesma funcionalidade e interface. Componentes de software podem ser compostos por uma única classe, por um grupo de classes ou, até mesmo, por um subsistema. Quanto maior a quantidade de elementos encapsulados em um componente diz-se que maior é a sua granularidade. Uma arquitetura organizada em camadas e constituída de componentes contribui para a qualidade do produto de software, tornando-o mais robusto e flexível. Para definir a arquitetura do software baseada em componentes, duas questões são importantes: quais classes serão alocadas a quais componentes e quais componentes serão alocados a quais servidores (disposição física dos componentes em nós de processamento). O diagrama de componentes pode ser utilizado para ilustrar a primeira situação, enquanto o diagrama de implantação pode ser usado para representar a distribuição física dos componentes.

8.6.1 Diagrama de componentes

Um diagrama de componentes mostra os vários componentes de software de um sistema e suas dependências. Os elementos que compõem este diagrama são os componentes, as interfaces e os relacionamentos de realização e dependência.

Na UML, componentes são considerados classificadores. Classificador é um conceito genérico da UML para denotar uma classe, um subsistema ou um componente. Um classificador possui comportamento, ou seja, implementa um

conjunto de operações. Quando um classificador implementa uma ou mais operações especificadas em uma interface, diz-se que ele realiza a interface. Um classificador pode realizar uma ou mais interfaces. Interface é um conjunto de especificações de serviço. Tem semelhança com uma classe abstrata, pois não gera instâncias, mas difere de classes abstratas por não conter estrutura interna (atributos e associações). Um serviço é composto por sua especificação e por seu método. A especificação define o que o serviço realiza. Contém as informações necessárias para que o serviço seja realizado e o resultado de sua realização. Método é o modo de realizar um serviço e, dada uma especificação, pode haver diversos métodos para realizá-lo.

A Figura 70 mostra um diagrama de componentes. Nele, um conjunto de classes foram agrupadas em um componente denominado Motorista e outro conjunto de classes em um componente chamado de SolicitacaoTransporte. Este componente depende do primeiro e realiza duas interfaces: Registrável e I_SolicitacaoTransporte. A interface I_SolicitacaoTransporte é utilizada pela classe UC10_Ctrl.

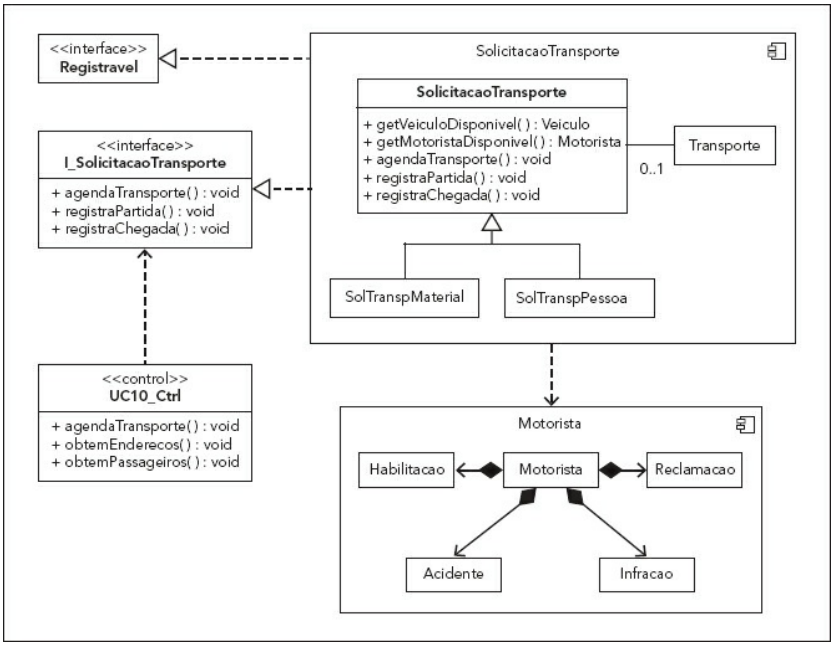


Figura 70 – Diagrama de componentes.

Fonte: elaborada pelo autor.

Observa-se que a notação para componente é uma caixa com um símbolo indicativo no canto superior direito. Dentro desta caixa podem ser representadas as classes que o compõem. Os relacionamentos de dependência são como os já utilizados em outros diagramas. Uma interface é representada por uma classe com o estereótipo «interface». O compartimento das operações pode ser exibido evidenciando a especificação da interface. O relacionamento de realização de uma interface por um componente é denotado por uma linha tracejada com um triângulo não preenchido tocando a interface. Alternativamente, pode-se usar a notação com ícones, como mostra a Figura 71.

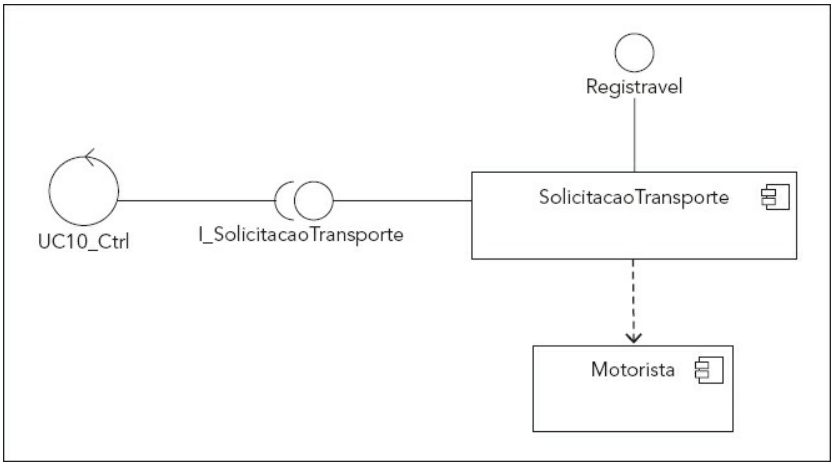


Figura 71 – Diagrama de componentes utilizando a notação de ícones.
Fonte: elaborada pelo autor.

Embora se esteja tratando de componentes como componentes de software, é importante salientar que a UML define três tipos de componentes:

- de execução – que existem em tempo de execução, tais coo, processos e threads;
- de instalação – que constituem a base dos componentes de execução (dll, Active-X);

- de trabalho – a partir dos quais os componentes de instalação são criados (documentos, arquivos-fonte, arquivos-objeto).

Os componentes podem ter diversos estereótipos, dentre eles «executavel», «documento» e «tabela».

8.6.2 Diagrama de implantação

O diagrama de implantação da UML é utilizado para representar a arquitetura física de um sistema. Ele representa a topologia física do sistema e, opcionalmente, pode representar os componentes que são executados nessa topologia. Os elementos deste diagrama são os nós e as conexões.

Um nó é uma unidade física que representa um recurso computacional com capacidade de processamento como processadores, servidores, dispositivos, sensores, roteadores. É representdo por um cubo, que pode conter o nome e o tipo do nó. Conexões ligam os nós e podem mostrar os mecanismos de comunicação entre os mesmos, como meio físico ou protocolos de comunicação. Uma conexão é representada por uma linha ligando dois nós. A Figura 72 exibe um diagrama de implantação.

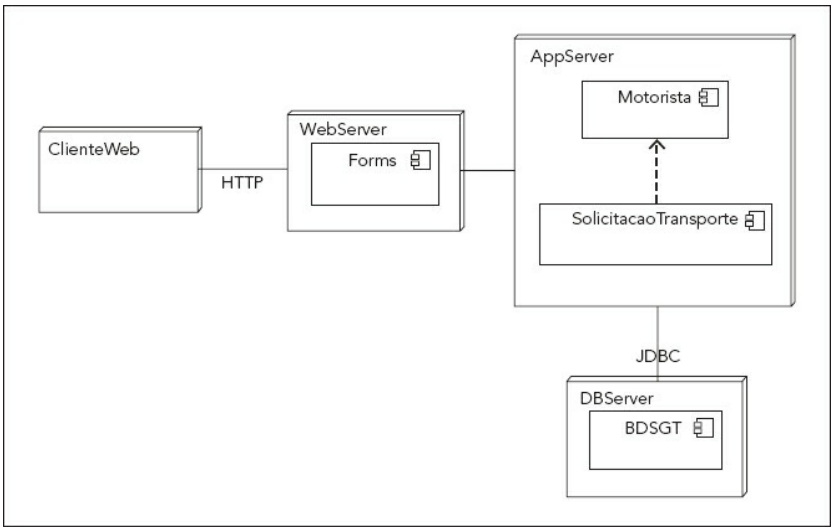


Figura 72 – Diagrama de implantação.
Fonte: elaborada pelo autor.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar o conhecimento sobre projeto de software, recomenda-se a leitura de Larman (2007) e Wazlawick (2004). Sobre padrões de projeto, sugere-se a leitura de Braude (2005), Fowler (2006) e Gamma (2000). Outras informações sobre projeto de interfaces de usuário podem ser obtidas em Galitz (2007) e Nielsen (1993).

RESUMO DO CAPÍTULO

O projeto trata de estabelecer uma solução para o sistema. Envolve a definição da arquitetura do software e o detalhamento técnico da solução. A arquitetura do software engloba a arquitetura física e a arquitetura lógica. A arquitetura lógica pode ser representada através de um Diagrama de Pacotes, e a arquitetura física utilizando-se o diagrama de implantação.

Um estilo comum para arquitetura lógica de sistemas de software é o em camadas. Geralmente têm-se três camadas principais: camada de domínio, camada de serviços e camada de interface com usuário. Cada uma deve ser detalhada tecnicamente. No detalhamento da camada de domínio duas atividades são necessárias: o projeto de colaboração de objetos, para o qual se pode usar o diagrama de componentes, e a elaboração do diagrama de classes de projeto. Para a construção de um produto de software de qualidade, deve-se minimizar o acoplamento e maximizar a coesão de seus objetos. Para minimizar o acoplamento entre objetos, deve-se evitar criar visibilidades desnecessárias entre os mesmos. Diretrizes documentadas em padrões de projetos podem ser seguidas para facilitar esse processo.

Um dos serviços mais importantes da camada de serviços é o de persistência. Como a maioria dos sistemas de software OO terá suas informações armazenadas em banco de dados relacionais, classes, atributos e relacionamentos deverão ser mapeados para o paradigma relacional em um processo chamado de mapeamento OO-ER. A camada de interface de usuário pode ser dividida em duas subcamadas, uma contendo os

elementos gráficos de interação com o usuário e outra que controla a lógica de operação da interface.

As classes de software podem ser agrupadas em componentes para os quais são definidas interfaces contendo a especificações dos serviços realizados pelo mesmo. O diagrama e componentes pode ser usado para modelar componentes.

CAPÍTULO 9

TESTES

Este capítulo aborda a disciplina de testes. Inicialmente o gerenciamento de qualidade é introduzido, pois teste de software se insere neste contexto. Na sequência, são apresentados os conceitos a respeito de teste, seus tipos e estágios. Também são comentados, brevemente, os testes de sistemas orientados a objetos e aplicações de teste. Por fim, as etapas de um processo de teste são apresentadas e exemplos de casos de teste baseados em cenários de casos de uso são demonstrados.

A disciplina de testes compreende um conjunto de atividades que pertencem a um tema mais amplo da Engenharia de Software chamado de verificação e validação (V&V). A verificação refere-se ao conjunto de atividades que garantem que o software implemente corretamente sua especificação. Já a validação refere-se a um conjunto diferente de atividades que garantem que o software construído satisfaça às exigências do cliente. A verificação e validação estão relacionadas a um tema mais amplo que é a garantia da qualidade de software.

Embora as atividades de teste de software desempenhem um papel extremamente importante na validação e verificação, outras atividades são necessárias, como, por exemplo, revisões e inspeções de software. O teste de software é o último estágio para avaliação da qualidade.

9.1 Defeito, erro e falha

Diversos termos são utilizados na Engenharia de Software no contexto da verificação e validação de software. Dentre eles, é importante destacar os seguintes:

- defeito (*fault*) – passo, processo ou definição de dados incorretos, como, por exemplo, uma instrução ou comando incorreto;
- erro (*error*) – diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro;

- falha (*failure*) – produção de uma saída incorreta com relação à especificação.

Um defeito pode ser encontrado no software em si ou na documentação e dados de configuração do mesmo. Embora um defeito seja uma variação de um atributo do produto desejado, ele não traz nenhum impacto até afetar o usuário/cliente ou o sistema operacional. Quando um defeito causa um erro na operação ou impacta negativamente um usuário/cliente tem-se uma falha. Um defeito pode tornar-se uma falha. Contudo, é possível que um defeito nunca se transforme em uma falha, assim como um único defeito pode originar uma infinidade de falhas.

9.2 Gerenciamento da qualidade de software

A Engenharia de Software objetiva a produção de software de qualidade. Pressman (2001) define qualidade como "conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido".

Segundo Sommerville (2010), o gerenciamento de qualidade de software pode ser estruturado em três atividades principais: garantia de qualidade (SQA – *Software Quality Assurance*), planejamento de qualidade e controle de qualidade.

As atividades de garantia de qualidade de software definem uma estrutura para se atingir a qualidade de software. Este processo envolve definir ou selecionar padrões que devem ser aplicados ao processo de desenvolvimento de software. Existem dois tipos de padrões que podem ser estabelecidos como parte do processo de garantia de qualidade: padrões de produto e padrões de processo. Padrões de produto são aqueles que se aplicam ao produto de software em desenvolvimento. Eles incluem padrões de documentos, como a estrutura do documento de requisitos a ser produzido; padrões de documentação, como um cabeçalho-padrão de comentário para uma definição de classe de objto;, e padrões de codificação, que definem como uma linguagem de programação deve ser utilizada. Padrões de processo são os que definem os processos a serem seguidos durante o desenvolvimento de software. Eles podem incluir definições de especificação, processos de projeto e validação, e uma descrição dos documentos que devem ser gerados no curso desses processos. A relação entre a qualidade do processo e qualidade do produto é complexa. Em princípio, quanto mais maturo um processo de software maior a probabilidade de se obter qualidade no produto.

O planejamento da qualidade deve selecionar os padrões organizacionais que forem apropriados a um determinado produto e processo de desenvolvimento e adaptá-los a um projeto específico de software.

O controle de qualidade envolve supervisionar o processo de desenvolvimento de software a fim de assegurar que os procedimentos e os padrões de garantia de qualidade sejam seguidos. Uma das abordagens para o controle de qualidade é a das revisões de qualidade, nas quais o software, sua documentação e os processos utilizados para produzi-lo são revisados por um grupo de pessoas. Elas são responsáveis por conferir se os padrões de projeto foram seguidos e se o software e os documentos estão em conformidade com esses padrões. Os desvios em relação aos padrões são anotados e submetidos à atenção da gerência do projeto.

9.2.1 Revisões de software

Revisões são métodos utilizados para a validação da qualidade de um processo ou produto. São aplicadas em vários pontos durante o desenvolvimento de software e servem para descobrir defeitos que possam ser eliminados. Segundo Pressman (2001), o benefício das revisões técnicas é a descoberta precoce dos defeitos de software, de forma que cada defeito possa ser corrigido antes do passo seguinte do processo de engenharia de software. Isto reduz o custo do software. O custo de reparação de um defeito descoberto na fase de projeto é, em média, de 60 a 100 vezes menor que o mesmo defeito descoberto com o software em operação. As revisões podem ser de três categorias:

- revisões de software – viam à detecção de erros nos requisitos, projetos ou código;
- revisões de progresso – visam fornecer informações sobre o progresso do projeto como custo, prazo, esco;
- revisões de qualidade – buscam uma análise técnica dos componentes ou da documentação do produto, a fim de encontrar inconsistências entre a especificação e o projeto, código ou documentação dos componentes e garantir que os padrões de qualidade definidos sejam seguidos.

As revisões podem ser realizadas de várias maneiras, com diversos graus de formalismo, indo desde um encontro informal, onde se discutem aspectos técnicos, até as revisões técnicas formais.

9.2.1.1 Revisões técnicas formais

As revisões técnicas formais são uma classe de revisão que inclui diversas técnicas, tais como, *walkthroughs*, inspeções e revisões *round-robin*. São realizadas em pequenos grupos e têm por objetivo:

- descobrir erros de função, lógica ou implementação em qualquer representação do software, incluindo modelos e código-fonte;
- verificar se o software que se encontra sob revisão atende a seus requisis;
- garantir que o software tenha sido representado de acordo com padrões predefinidos.

As revisões são realizadas em reuniões devidamente planejadas. Devem ter entre três e cinco participantes e duração inferior a duas horas. Uma reunião de revisão deve focar em uma parte específica do software, de forma a aumentar a probabilidade de encontrar defeitos. Na técnica de *walkthrough*, o produtor do artefato sendo reviado, "caminha através" do produto, explicando o material, enquanto os revisores levantam questões baseadas na preparação antecipada que fizeram e realizam os apontamentos que devem ser posteriormente tratados.

9.3 Teste de software

Teste de software é o processo de executar software com a intenção de descobrir erros. A realização de um teste se dá pela execução de casos de teste. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto e um teste bem-sucedido é aquele que revela um erro ainda não descoberto. A atividade de teste não pode mostrar a ausência de defeitos; ela só pode mostrar se defeitos de software estão presentes. Tem-se que a probabilidade de existência de erros é proporcional aos erros encontrados.

Após a realização de um teste, os resultados são avaliados, comparandose os resultados do teste com os resultados esperados. Quando os resultados obtidos não são os esperados, infere-se a presença de erro e inicia-se o processo depuração. As atividades de teste e de depuração são atividades diferentes, mas complementares.

9.3.1 Depuração

A depuração ocorre em consequência de testes bem-sucedidos, ou seja, quando um caso de teste revela um erro e visa à sua remoção. Embora a depuração deva ser um processo disciplinado, ela ainda tem muito de arte. Um engenheiro de software, ao avaliar os resultados de um teste, frequentemente se defronta com um indício sintomático de uma falha de software. Ou seja, a manifestação externa do erro (falha) e a causa interna do mesmo (defeito) não têm nenhuma relação óbvia entre si. O processo mental que liga um sintoma a uma causa é a depuração. A seguir estão listadas algumas características dos erros que tornam o processo de depuração complexo e difícil:

- o sintoma e a causa podem estar distantes (em termos de código-fonte);
- o sintoma pode ser intermitente;
- o sintoma pode ser causado por erro humano, o que pode não ser rastreável;
- o sintoma pode ser devido a causas distribuídas;
- o sintoma pode ser resultado de problemas de temporalidade e não de processamento;
- pode ser difícil reproduzir com precisão as condições da falha.

Em geral, três categorias de abordagens à depuração podem ser propostas: força bruta, *backtracking* e eliminação da causa. A abordagem da força bruta refere-se ao uso de computador para registrar e analisar grandes volumes de informação a respeito da falha, como, por exemplo, *dumps* de memória e análise de *logs*. *Backtracking* é uma abordagem bastante comum, utilizada em trechos de código não muito grandes. Inicia-se no local em que o sintoma foi descoberto e rastreia-se o código-fonte para trás manualmente até que o local da causa seja encontrado. A técnica de eliminação da causa introduz o conceito de partição binária. Os dados relacionados à ocorrência de erros são organizados para isolar causas potenciais. Uma hipótese de causa é lançada e os dados de uma partição são usados para provar ou refutar a hipótese. O processo repete-se até que a causa seja encontrada.

9.3.2 Casos de teste

Segundo Pressman (2001), qualquer produto desenvolvido por engenharia pode ser testado de duas maneiras. Conhecendo-se a função específica que um produto

projetado deve executar, testes podem ser realizados para demonstrar que cada função é totalmente operacional. Conhecendo-se o funcionamento interno de um produto, testes podem ser realizados para garantir que a operação interna do produto tenha um desempenho de acordo com as especificações e que os componentes internos foram postos à prova.

A primeira abordagem de teste é chamada de funcional (ou teste de caixa preta), e a segunda, teste estrutural (ou teste de caixa branca). Em software, um teste funcional examina alguns aspectos de um sistema sem se preocupar com a estrutura lógica do software. Testes funcionais atuam sobre as interfaces do software, fornecendo e obtendo dados e informações e comparando os resultados obtidos com os resultados esperados.

O teste estrutural do software baseia-se em um minucioso exame dos detalhes procedimentais. Os caminhos lógicos através do software são testados, fornecendo-se casos de teste que põem à prova conjuntos específicos de código. Infelizmente, testes estruturais que ponham à prova todos os caminhos possíveis de um software são praticamente impossíveis, pois mesmo pequenos trechos de código podem gerar milhões de caminhos distintos.

É importante ressaltar que as técnicas de teste devem ser vistas como complementares e a questão está em como utilizá-las de forma que as vantagens de cada uma sejam melhor exploradas em uma estratégia de teste que leve a atividades de teste que sejam eficazes e de baixo custo.

Além das abordagens funcional e de estrutura, existe a abordagem de teste baseada em erros.

9.4 Tipos de teste

Conforme mencionado anteriormente, para se conduzir as atividades de teste dispõe-se de técnicas de teste funcional, estrutural e baseada em erros.

9.4.1 Teste estrutural

O teste estrutural baseia-se no conhecimento da estrutura interna da implementação. O teste estrutural é um método de projeto de casos de teste que usa a estrutura de controle para derivar casos de teste. A seguir são descritas duas técnicas para testes estruturais: teste de caminho básico e teste de estruturas de controle.

Teste de caminho básico é uma técnica que visa gerar casos de teste para

exercitarem cada instrução do software pelo menos uma vez durante a atividade de testes. A complexidade ciclomática é uma métrica de software baseada na teoria dos grafos, que proporciona uma medida quantitativa da complexidade lógica de um programa. Quando usado no contexto do método de teste do caminho básico, o valor computado da complexidade ciclomática define o número de caminhos independentes do conjunto básico de um programa e oferece um limite máximo para o número de testes que deve ser realizado para garantir que todas as instruções sejam executadas pelo menos uma vez. Por exemplo, dado o código fonte de uma operação de uma classe, pode-se computar a complexidade ciclomática desta operação. Este valor indica a quantidade de fluxos linearmente independentes na operação. Estes fluxos devem ser identificados e, para cada um, deve ser projetado um caso de teste que contenha valores e gere situações de tal forma que cada passo do fluxo independente seja executado.

Teste de estruturas de controle são métodos de projeto de caso de teste que põem à prova as estruturas de controle de fluxo de um código. Por exemplo, pode-se projetar um caso de teste para que o resultado de uma decisão no fluxo em análise seja verdadeiro e outro para que seja falso. De forma similar, podese projetar casos de teste para validar as condições de término de execução de iterações (*loops*) ou o disparo de exceções.

9.4.2 Teste funcional

O teste funcional também é conhecido como teste caixa preta pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída. Na técnica de teste funcional são verificadas as funções do sistema sem preocupação com os detalhes de implementação.

O teste funcional envolve dois passos principais: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software. As funções que o software deve conter são identificadas a partir de sua especificação. Assim, uma especificação bem elaborada e de acordo com os requisitos do usuário é essencial para esse tipo de teste. A seguir, são descritas três técnicas para testes funcionais: particionamento em classes de equivalência, análise do valor limite e grafo de causa-efeito.

9.4.2.1 Particionamento em classes de equivalência

É um método de teste de caixa preta que divide o domínio de entrada de um programa em classes de dados a partir das quais os casos de teste podem ser derivados. O projeto de casos de teste para o particionamento de equivalência baseia-se na avaliação de classes de equivalência para uma condição de entrada. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Por exemplo, se uma condição de entrada for um intervalo, deve-se definir uma classe válida, contendo valores dentro do intervalo, e duas classes inválidas, uma para valores superiores ao intervalo e outra para os valores inferiores ao intervalo. Devem ser projetados casos de testes que explorem todas as entradas do objeto de teste com instâncias significativas de todas as combinações de classes válidas e inválidas.

9.4.2.2 Análise do valor limite

É uma técnica de projeto de casos de teste que complementa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, devem ser selecionados os elementos limites das classes. Isto porque é bastante comum a presença de erros no tratamento de valores limítrofes de domínios.

9.4.2.3 Técnicas de grafo de causa-efeito

É uma técnica de projeto de casos de teste para validar requisitos derivados de expressões complexas, geralmente escritas em linguagem natural. Esta técnica sugere transcrição de tais expressões em grafos de causa-efeito. Esses grafos podem ser convertidos em árvores ou tabelas de decisão. As regras identificadas nas árvores ou tabelas podem sugerir valores e situações para o projeto dos casos de teste.

9.4.3 Teste baseado em erros

São técnicas que consistem em incluir propositalmente algum erro no programa e observar seu comportamento, comparando-o com o comportamento original. Fornecem indicadores para gerenciar o processo de teste, como porcentagem de erros remanescentes e qualidade dos casos de teste. Existem dois tipos de testes baseados em erros: semeadura de erros e análise de mutantes.

9.4.3.1 Semeadura de erros

Na técnica de semeadura de erros uma quantidade definida de erros artificiais e típicos é introduzida em um programa que já foi objeto de teste e cujos erros são conhecidos. Os casos de teste já realizados são repetidos para o programa e verifica-se a quantidade de novos erros que foram identificados. A proporção entre o total de novos erros identificados e o total de novos erros introduzidos permite estimar a quantidade de erros remanescentes para o conjunto de casos de teste definido para o programa. Se os casos de teste foram capazes de identificar 40% dos novos erros, pode-se supor que isto vai ocorrer para o conjunto total de erros do programa original. Desta forma, pode-se comparar a estimativa de erros remanescentes com a confiabilidade esperada (especificada) para o software.

9.4.3.2 Análise de mutantes

Esta técnica é empregada para avaliar o quanto um conjunto de casos de teste é adequado para testar um dado programa. Um conjunto de casos de teste T é gerado para um programa P. A aplicação de T revela a existência do conjunto de erros E no programa P. Gera-se, então, um programa mutante P' que possua pequenas mudanças em relação ao original (defeitos comuns em programação). A aplicação de T revela a existência do conjunto de erros E' no programa P'. Se E <> E', o mutante é considerado "morto". Se E = E', o mutante continua "vivo". Os mutantes vivos são então analisados. Ou o mutante P' é equivalente ao programa original P, ou o caso de teste T é insuficiente para detectar a modificação introduzida, o que significa que outros erros podem estar escondidos no programa P e novos casos de teste devem ser definidos.

O escore de mutação é o índice dado pela divisão do número de mutantes mortos pelo número de mutantes gerados. Este valor varia no intervalo entre 0 e 1. Quanto maior o escore de mutação, mais adequado é o conjunto de teste para o programa que está sendo testado.

9.5 Estágios de teste

As atividades de teste de software se desenvolvem ao longo do processo de desenvolvimento de software, ocorrendo em três estágios ou níveis de teste: de unidade, de integração e de sistema.

9.5.1 Teste de unidade

O teste de unidades concentra-se no esforço de verificação da menor unidade de projeto de software – um componente, uma classe ou mesmo uma operação de classe. O teste de unidade baseia-se sempre na caixa branca e pode ser realizado em paralelo para múltiplos componentes. A atividade de testes de unidades geralmente é considerada um adjunto da codificação. Depois que o código-fonte foi desenvolvido, revisado e verificado, inicia-se o projeto do caso de teste da unidade. Cada caso de teste deve estar associado a um conjunto de resultados esperados.

Visto que uma unidade pode não representar um todo íntegro, um software *driver* e/ou *stub* deve ser desenvolvido para cada unidade de teste. *Drivers* e *stubs* são pequenos programas ou trechos de código que não pertencem ao escopo do produto de software em desenvolvimento, mas que precisam ser criados para viabilizar o processo de teste. Os *drivers* emulam o componente que demanda uma ou mais interfaces, enquanto os *stubs* fazem o papel das interfaces requeridas. Alternativamente, softwares específicos para teste unitário podem ser utilizados, eliminando a necessidade de criação de *drivers* e *stubs*.

9.5.2 Teste de integração

Refere-se ao teste que deve ser aplicado a componentes que foram integrados. Frequentemente existe uma tendência para integração não incremental, do tipo *big-bang*. Esta abordagem aumenta em muito o esforço de teste, visto que o processo de depuração se torna muito mais complexo. A integração incremental, na qual os componentes são integrados e testados um a um ou em pequenas quantidades, torna o processo de depuração mais simples, além de aumentar a probabilidade de as interfaces serem testadas completamente. Nesse caso, durante o processo de integração, *drivers* e *stubs* terão de ser criados de forma a viabilizar a integração, até que todos os componentes tenham sido integrados.

Testes de regressão, isto é, a realização de todos ou de alguns dos testes anteriores, devem ser realizados a fim de garantir que novos erros não tenham sido introduzidos com a integração de um novo componente. A integração pode ser de baixo para cima (*bottom-up*) ou de cima para baixo (*top-down*). Neste caso, ainda pode ser do tipo *depth-first* (primeiro em profundidade) ou *breadth-first* (primeiro em largura).

Diferentemente dos testes de unidade, o teste integrado deve ser planejado e

realizado por completo. Durante o teste integração, uma versão estável dos componentes deve estar disponível e para uso exclusivo do teste.

9.5.3 Teste de sistema

O software é um dos componentes de um sistema baseado em computador. O teste de sistema é, na verdade, uma série de diferentes testes, cujo objetivo é pôr um sistema baseado em computador completamente à prova. Existem vários tipos de teste de sistema, cada um com um propósito específico. Dentre outros, destacam-se:

- teste de recuperação – é um teste de sistema que força o software a falhar de diversas maneiras e verificar se a recuperação é adequadamente executada;
- teste de segurança – visa verificar se todos os mecanismos de proteção embutidos são eficientes;
- teste de estresse ou capacidade – executa o sistema de maneira a exigir recursos em quantidade, frequência ou volume anormais para testar os limites máximos do mesmo;
- teste de desempenho – é idealizado para testar o desempenho *run-time* do software. O teste de desempenho exige instrumentação de hardware e software para monitorar e registrar o desempenho do sistema em intervalos constantes de forma que se possa descobrir situações que levam à degradação e possível falha do sistema;
- teste de regressão – teste total do sistema após uma mudança significativa com a finalidade de identificar "efeitos colaterais" da mudança;
- teste de validação, aceitação ou homologação – visa, por meio de uma série de testes caixa preta, validar se o software está em conformidade com os requisitos do software e se atende às expectativas dos usurário/cliente.

Quando um software customizado é construído para um cliente, uma série de testes de aceitação é realizada para capacitar o cliente a validar todos os requisitos. Se o software for projetado para ser utilizado por muitos clientes, torna-se impraticável o uso de testes de aceitação. A maioria dos construtores de softwares comerciais usa um processo denominado teste alfa e teste beta. O teste alfa é realizado por um cliente nas instalações do desenvolvedor. O teste beta é realizado em uma ou mais instalações

do cliente pelo usuário final do software.

9.6 Estratégia de teste

Uma estratégia de teste se caracteriza pela definição da abordagem geral a ser aplicada nos testes, descrevendo como o software será testado, identificando os estágios de teste que serão aplicados e os métodos, técnicas e ferramentas a serem usadas. Fornece o guia e direção para as fases e etapas restantes do processo de testes e será a base para a formulação dos planos de testes.

O processo de desenvolvimento de software inicia-se com a definição dos requisitos, que são posteriormente analisados, projetados e finalmente implementados. A cada etapa, o nível de abstração diminui. Uma estratégia de teste pode ser imaginada ao contrário. Inicia-se com o teste de unidade, segue-se com o teste integrado e conclui-se com testes de sistema. A cada etapa, o nível de abstração aumenta.

O teste de unidade foca nas menores unidades componentes do software que podem ser testadas, tais como, componentes ou classes. O teste de unidade faz muito uso de técnicas de caixa branca. Em seguida, os componentes têm de ser integrados para formarem um pacote de software, e um teste de integração precisa ser feito. As técnicas de caixa preta são mais adequadas para o teste integrado, embora não se exclua a possibilidade/necessidade de se utilizar técnicas de caixa branca. Depois que o software foi integrado, um conjunto de testes de alto nível é realizado. Os testes de sistema oferecem a garantia final de que o software atende a todos os requisitos funcionais e não-funcionais. As técnicas de caixa preta são usadas exclusivamente durante a validação.

9.7 Teste orientado a objetos

Há duas atividades fundamentais no processo de testes: o teste de componentes, em que os componentes de sistema são testados individualmente, e os testes de integração, em que coleções de componentes são integradas em subsistemas e no sistema final para testes. Essas atividades são igualmente aplicáveis a sistemas orientados a objetos. Contudo, há diferenças importantes entre sistemas OO que afetam o procedimento de teste. Dentre estas diferenças cabe mencionar:

- objetos, vistos como componentes individuais, são maiores que funções isoladas;
- quando integrados não há um nível superior óbvio;
- quando objetos são reutilizados, pode-se não se ter acesso ao código fonte do componente em análise.

Os três estágios de teste podem ser aplicados a testes de sistemas OO. O teste unitário vai englobar o teste das operações individuais associadas com os objetos (caixa branca ou caixa preta) e o teste de classes de objetos. No primeiro, podem ser usadas técnicas de caixa branca e de caixa preta. No segundo, técnicas de caixa preta. Os testes estruturais das classes podem ser feitos a partir das técnicas de caminhos básicos e de fluxos de controle. O teste de classes de objeto inclui:

- o teste isolado de todas as operações associadas com o objeto;
- a atribuição e consulta de todos os atributos associados com o objeto;
- o teste do objeto em todos os estados níveis possíveis.

Para o teste integrado, envolvendo o agrupamento de objetos, a integração *top-down* ou *botton-up* é inadequada para criar grupos de objetos relacionados, pois em sistemas OO não há um nível superior óbvio. Outras abordagens, como testes com base em cenários, devem ser utilizadas. Existem três abordagens possíveis para os testes de integração: teste de threads, teste de interação de objetos e o teste de caso de uso ou testes baseados em cenários. Os casos de uso descrevem as funções do sistema e a realização de cada cenário de cada caso de uso implica a interação de um conjunto de objetos em determinado estado. Os casos de teste são projetados para avaliar estes cenários.

No teste de sistema de software orientado a objetos a verificação e a validação são realizadas da mesma maneira que para outros tipos de sistemas.

9.8 Aplicações de teste

Como os testes constituem uma fase dispendiosa e trabalhosa de processo de software, ferramentas de teste foram desenvolvidas. Atualmente oferecem uma gama de recursos, reduzindo significativamente o custo do processo de testes. Estas ferramentas podem incluir as seguintes funcionalidades:

- gerenciamento de teste (execução dos testes e análise dos resultados);
- geração de dados de teste;
- comparação de arquivos (essenciais em testes de regressão);
- geração de relatórios;
- simulação de ambientes de execução e de operação do sistema por usuários (neste último caso, roteiros simulando o usuário operando o software são gravados e depois executados, simulando o uso concorrente ou massivo do sistema).

Para citar um exemplo de ferramenta, JUnit é um framework de testes de regressão desenvolvido por Erich Gamma e Kent Beck, podendo ser usado para a realização de testes unitários em Java. JUnit estrutura os testes e provê mecanismos para executá-los automaticamente, além de prover ferramentas para asserções, executar testes, agregar testes e mostrar resultados.

9.9 Processo de teste

O teste de produtos de software envolve basicamente quatro etapas: planejamento, projeto dos testes, execução e avaliação dos resultados.

9.9.1 Planejamento

O planejamento trata da elaboração da estratégia de teste e do plano de testes. Abrange ainda a preparação do ambiente de teste, incluindo equipamentos, ferramentas e recursos necessários para especificar, executar e avaliar os testes. Os ambientes de desenvolvimento e produção costumam ser distintos entre si. Por isso, a necessidade da criação de um ambiente de teste capaz de reproduzir todas as características do ambiente de produção para o qual o software está destinado.

9.9.2 Projeto dos testes

O projeto dos testes trata da elaboração dos casos de teste, dos *scripts* (no caso de uso de ferramentas de automação) e dos roteiros de teste. Os casos de teste devem ser concebidos de forma a verificar e validar tanto os requisitos funcionais quanto os

não-funcionais do software.

Para validar requisitos funcionais em processos dirigidos a casos de uso ou cenários, como é o caso do PU, os casos de teste devem ser projetados de forma a testar os cenários representativos dos casos de uso. Os casos de teste devem ser identificados e especificados em um documento próprio. As especificações devem conter exemplos de dados concretos para a realização dos testes e uma massa de dados válida deve ser obtida. A sequência com que os testes serão realizados pode ser definida no roteiro de teste. No caso da utilização de ferramentas de automação de testes, os *scripts* de testes funcionais devem ser elaborados nessas ferramentas. O Quadro 11 apresenta uma lista de cenários para testes referentes ao caso de uso UC10 – Solicitar Transporte. O Quadro 12 mostra a especificação de um caso de teste funcional baseado em cenários. O cenário em questão é o fluxo principal do caso de uso UC10 e é identificado por CT010_01.

Quadro 11 – Cenários para teste de software

UC10 – Solicitar Transporte	
Caso de Tese	**Cenário**
CT010_01	Solicitação de transporte compassageiro ➜ O sistema agendou uma solicitação, reservando um veículo e um motorista para o transporte. ➜ O sistema emitiu uma ordem de transporte para o motorista alocado ao transporte. ➜ Os passageiros foram notificados por e-mail.
CT010_02	Solicitação de transporte compassageiro, mas passageiro não existe ➜ UC020 – Manter Passageiro é invocado. ➜ O sistema agendou uma solicitação, reservando um veículo e um motorista para o transporte. ➜ O sistema emitiu uma ordem de transporte para o motorista alocado ao transporte. ➜ Os passageiros foram notificados por e-mail.
	Solicitação de transporte compassageiro, mas local de partida não existe ➜ UC030 – Manter Local é invocado.

CT010_03	→ O sistema agendou uma solicitação, reservando um veículo e um motorista para o transporte. → O sistema emitiu uma ordem de transporte para o motorista alocado ao transporte. → Os passageiros foram notificados por e-mail.
CT010_04	Solicitação de transporte com passageiro, mas local de chegada não existe → UC030 – Manter Local é invocado. → O sistema agendou uma solicitação, reservando um veículo e um motorista para o transporte. → O sistema emitiu uma ordem de transporte para o motorista alocado ao transporte. → Os passageiros foram notificados por e-mail.
CT010_05	Solicitação de transporte com passageiro, mas sistema não pôde agendar transporte → O sistema notificou o Gerente de Transporte sobre a impossibilidade de agendar uma solicitação
CT010_06	Solicitação de transporte de materiais → O sistema agendou uma solicitação, reservando um veículo e um motorista para o transporte. → O sistema emitiu uma ordem de transporte para o motorista alocado ao transporte.
CT010_07	Solicitação de transporte de materiais, mas sistema não pôde agendar transporte → O sistema notificou o Gerente de Transporte sobre a impossibilidade de agendar uma solicitação

Fonte: elaborado pelo autor.

Quadro 12 – Caso de teste para um cenário específico

Caso de Teste	**CT010-01 – Solicitação de transporte com passageiro**	**OK**
Precondição	→ O solicitante Solicitante1 está autenticado e autorizado	☐
	→ Motorista Motorista1 está livre das 16h às 18h	☐

	→ O veículo ABC-1234 está livre das 16h às 18h	☐
	→ Fulano (e-mail = fulano@mail.com)	☐
	→ Beltrano (e-mail = beltrano@mail.com)	☐
Procedimentos	1. O caso de uso se inicia quando o solicitante requisita um transporte.	☐
	2. O sistema apresenta uma lista de locais de destino previamente cadastrados. (Aeroporto XYZ, Hotel ABC, Empresa)	☐
	3. O solicitante escolhe um local de destino. (Aeroporto XYZ)	☐
	4. O solicitante informa a hora desejada para chegada. (17h)	☐
	5. O sistema apresenta uma lista de locais de partida previamente cadastrados. (Aeroporto XYZ, Hotel ABC, Empresa)	☐
	6. O solicitante escolhe um local de partida. (Empresa)	☐
	7. O solicitante informa a hora da saída. (16h)	☐
	8. O sistema apresenta uma lista de passageiros previamente cadastrados. (Fulano, Beltrano, Cicrano)	☐
	9. O solicitante seleciona os passageiros. (Fulano e Beltrano)	☐
	10. O solicitante confirma a solicitação. (OK)	☐
	11. O sistema agenda a solicitação, emitindo uma ordem de transporte para o motorista alocado e, no caso de transporte de passageiros, um e-mail para cada um deles.	☐
	12. O caso de uso se encerra.	☐
Resultados esperados	→ O sistema agendou uma solicitação para os passageiros Fulano e Beltrano com local de partida = " Empresa", local de chegada = " Aeroporto", data de partida = " 16h", data de chegada = " 17h", reservando o veículo (ABC-1234) e o motorista Motorista1.	☐
	→ O sistema emitiu uma ordem de transporte para o motorista Motorista1 e enviou e-mail com dados do transporte para fulano@mail.com e beltrano@mail.com	☐
Observações		

Fonte: elaborado pelo autor.

9.9.3 Execução dos testes e avaliação dos resultados

A execução refere-se à execução dos testes planejados conforme os casos de teste, dos *scripts* (no caso de uso de ferramentas de automação) e dos roteiros de teste

e do registro dos erros obtidos. Os erros encontrados devem ser relatados e encaminhados para depuração.

RECOMENDAÇÕES PARA COMPLEMENTAÇÃO DE ESTUDOS

Para aprofundar o conhecimento sobre teste de software recomenda-se a leitura de Perry (1999) e Myers (1979).

RESUMO DO CAPÍTULO

O teste de software, em conjunto com revisões de software, é uma atividade para verificação e validação que está estritamente ligada com a garantia da qualidade de software. Existem três tipos de testes: teste estrutural ou caixa branca, teste funcional ou caixa preta e teste baseado em erros. Para cada um desses tipos de teste, diversas técnicas podem ser aplicadas, tais como, teste do caminho básico, particionamento de equivalência e análise de mutantes. Os testes podem ser aplicados em diversos estágios, indo desde o teste unitário de um componente de software, passando pelo teste integrado, no qual são testados um grupo de componentes integrados, até se chegar ao teste de sistema, onde todo o sistema é posto à prova. Diversos testes de sistema, com diversas finalidades, podem ser executados. Dentre eles, pode-se citar o teste de desempenho e o teste de aceitação. Uma estratégia de teste define quais tipos de testes serão aplicados em quais estágios e em quais momentos do processo de desenvolvimento de software, sendo insumo para o planejamento de teste. Um processo de teste envolve o planejamento do testes, o projeto do testes, que demanda a criação de casos de teste, *scripts* e roteiros, a execução dos testes e a avaliação dos resultados. Erros devem ser submetidos à depuração para correção do defeito. A execução de testes torna-se menos dispendiosa se assistida por ferramentas de teste.

REFERÊNCIAS

BEZERRA, Eduardo. *Princípios de Análise e Projeto de Sistemas com UML*. Rio de Janeiro: Elsevier, 2002. 286 p.

BOEHM, B. & PAPACCIO, P. *Understanding and Controlling Software Costs*. IEEE, 1988.

BOOCH, Grady. *Object-oriented analysis and design: with applications*. 2. ed. Reading: Addison-Wesley, 1994. 589 p.

BRAUDE, Eric. Projeto de Software. *Da programação à arquitetura: uma abordagem baseada em Java*. Porto Alegre: Bookman, 2005. 619 p.

COCKBURN, Alistair. *Escrevendo casos de uso eficazes*. Um guia prático para desenvolvedores de software. Bookman. Porto Alegre, 2005. 254 p.

COHN, Mike. *Desenvolvimento de Software com Scrum*. Porto Alegre: Bookman, 2011.

FOWLER, Martin. *Padrões de arquitetura de aplicações corporativas*. Porto Alegre: Bookman, 2006. 493 p.

GALITZ, Wilbert. *The Essencial Guide to User Interface Design: an introduction to GUI design principles and techniques*. Indianapolis: Wiley Publishing Inc, 2007.

GAMMA, Erich. *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Porto Alegre: Bookman, 2000. 364 p

GUEDES, Gilleanes T. A. *UML 2. Uma abordagem prática*. São Paulo: Novatec, 2009. 485 p.

IEEE – Institute of Electrical and Electronics Engineers. IEEE Std. 610 12-1990 –, Standard glossary off software engineering terminology, IEEE, Piscataway, NJ, 1997.

JACOBSON, Ivar. *Object-oriented software engineering: a use case driven approach*. Harlow: Addison-Wesley, 1994. 528 p.

JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James. *The unified software development process*. Reading: Addison-Wesley, 1999. 463 p.

KOTONYA, G.,; SOMMERVILLE, I. *Requirements Engineering: processes and techniques*. Chichester: John Wiley & Sons, 1998. 282 p.

KRUCHTEN, Philippe. *The Rational Unified Process – An Introduction*. 2 ed. Addison-Wesley, 2001. 298 p.

LARMAN, Craig. *Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento interativo*. 3. ed. Porto Alegre: Bookman,

2007. 695 p.

MYERS, Glenford J. *The art of software testing*. New York: John Wiley & Sons, 1979. 177 p.

NIELSEN, Jakob. *Usability Engineering*. San Diego: Morgan Kaufmann, 1993.

OMG, Object Management Group, 2010. *UML 2.3 Infrastructure Specification*. www.omg.org

OMG, Object Management Group, 2010. *UML 2.3 Superstructure Specification*. www.omg.org

PERRY, William E. *Effective methods for software testing*. 2. ed. New York: John Wiley & Sons, 1999. 812 p.

PMI, PROJECT MANAGEMENT INSTITUTE. *Um Guia do conhecimento em gerenciamento de projetos (Guia PMBOK)*. 4. ed. Philadelphia: Project Management Institute, 2008. 459 p.

PRESSMAN, Roger S. *Software engineering: a practitioner's approach*. 5. ed. Boston: Mc-Graw-Hill, 2001. 860 p.

SOMMERVILLE, Ian. *Engenharia de software*. 8. ed. São Paulo: Pearson, 2010. xiv, 552 p.

WAZLAWICK, Raul Sidnei. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. Rio de Janeiro: Elsevier, 2004. Campus 298 p.

1 Diagramas de casos de uso serão apresentados no Capítulo 5.
2 Há uma discussão sobre se nessas situações tem-se um ou quatro casos de uso. A prática, contudo, tem demonstrado que, exceto para casos muito complexos, o ideal é tratar todos esses cenários com um único caso de uso.
3 Cabe ressaltar que os primórdios do pensamento orientado a objetos remontam à década de 1970, com o desenvolvimento das primeiras linguagens orientadas a objeto: Smalltalk e SIMULA.

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

EDITORA UNISINOS

Editora da Universidade do Vale do Rio dos Sinos
EDITORA UNISINOS
Av. Unisinos,
950 93022-000 São Leopoldo RS Brasil

Telef.: 51.3590 8239
Fax: 51.3590 8238
editora@unisinos.br

Knewitz, Marcos André.

Introdução ao desenvolvimento de software com UML / Marcos André Knewitz. – São Leopoldo, RS : Ed. UNISINOS, 2011.
162 p. – (Coleção EAD)

K68i ISBN 978-85-7431-467-9

1. UML (Computação). 2. Métodos orientados a objetos (Computação). 3. Software – Desenvolvimento. 4. Engenharia de software. I. Título. II. Série.

CDD 005.117
CDU 004.414.23

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecário: Flávio Nunes – CRB 10/1298)
Esta obra segue as normas do Acordo Ortográfico da Língua Portuguesa vigente desde 2009.

Editor
Carlos Alberto Gianotti

Acompanhamento editorial
Mateus Colombo Mendes

Revisão	*Editoração*	*Capa*
Márcia Hendrischky Santos	Décio Remigius Ely	Isabel Carballo

Impressão, inverno de 2011.

Edição digital: dezembro 2013

Arquivo ePub produzido pela **Simplíssimo Livros**

Sobre o autor

MARCOS ANDRÉ KNEWITZ é Mestre em Computação Aplicada e bacharel em

Informática pela Universidade do Vale do Rio dos Sinos (Unisinos). Com mais de 20 anos de experiência profissional, atuou em projetos de desenvolvimento e implantação de sistemas de software nos segmentos financeiro, industrial e educacional. Atualmente, é professor universitário e gestor de Tecnologia de Informação.