Livro-Texto

Um método completo para Desenvolvimento Orientado a Objetos com UML: da análise à implementação em Java

Rosana T. Vaccare Braga
Paulo Cesar Masiero

ICMC – Universidade de São Paulo
São Carlos – 2007

Um método completo para Desenvolvimento Orientado a Objetos com UML: da análise à implementação em Java

Sumário

Capítulo 1 – Introdução

1.1 – Contexto e motivação para modelagem orientada a objetos

O que é análise Orientada a Objetos? Para responder a esta pergunta, convém discutir um pouco o processo de desenvolvimento de sistemas. Em geral, antes que um sistema possa ser desenvolvido, é necessário saber seus requisitos e passar por uma fase de análise, com o objetivo de criar modelos que representem o sistema em formatos mais facilmente implementáveis. Esse esforço é necessário e vale a pena, pois para garantir a qualidade do sistema resultante é necessário seguir um processo de desenvolvimento de software.

Iniciar a programação diretamente, sem passar pela fase de análise, como se faz freqüentemente quando se desenvolve um sistema pequeno e simples, pode ser desastroso quando o sistema alvo é grande e/ou complexo. Pode-se ocasionalmente obter um sistema satisfatório e que consiga sobreviver ao desafio da manutenção, mas isso certamente seria um caso isolado e dependente da sorte e de competências individuais.

Quando se segue um processo de desenvolvimento, seja ele qual for, uma fase importante refere-se à análise do sistema. Essa fase é tratada em uma parte do Processo Unificado denominada "Disciplina de Requisitos". O Processo Unificado é o processo adotado neste livro (ver o Capítulo 2). Os modelos produzidos durante essa disciplina são depois transformados em modelos de projeto e, finalmente, em código fonte. Os modelos de análise são mais próximos do que ocorre no mundo real, enquanto os modelos de projeto vão um pouco além e refletem mais concretamente as soluções computacionais para tornar a implementação possível.

A análise orientada a objetos oferece diversas opções de modelos para que as informações sobre o sistema a ser desenvolvido sejam representadas de forma o mais semelhante possível aos objetos presentes no mundo real. Assim, a análise e projeto de software orientados a objetos conduzem a modelos contendo vários objetos, muitos dos quais correspondem diretamente a elementos do sistema do mundo real. Por exemplo, para representar uma biblioteca por meio de um sistema orientado a objetos, o modelo teria objetos como Livro, Leitor, Atendente, Bibliotecária, etc.

A análise orientada a objetos fornece mecanismos para representação de modelos estáticos e dinâmicos. Modelos estáticos podem mostrar, por exemplo, os objetos do sistema e seus relacionamentos. Já os modelos dinâmicos mostram a comunicação entre os objetos para alcançar os propósitos do sistema. Neste caso, é importante identificar a ordem em que as atividades são desempenhadas para cumprir um certo objetivo.

Por exemplo, um modelo estático de uma biblioteca mostraria o relacionamento entre seus diversos componentes. Biblioteca possui Estantes que armazenam Livros, que são emprestados pelo Atendente ao Leitor. Um modelo dinâmico poderia mostrar o processo de empréstimo de um livro a um leitor. O Leitor dirige-se ao balcão e entrega ao atendente o

livro a ser emprestado. O Atendente solicita os dados do leitor, calcula a data de devolução e registra o empréstimo.

1.2 – Histórico da Orientação a objetos

A Orientação a Objetos (OO) é uma tecnologia para a produção de modelos que especifiquem o domínio do problema de um sistema. Quando construídos corretamente, sistemas orientados a objetos são flexíveis a mudanças, possuem estruturas bem conhecidas e oferecem a oportunidade de criar e implementar componentes reutilizáveis. Modelos orientados a objetos são implementados convenientemente utilizando uma linguagem de programação orientada a objetos.

A OO surgiu no fim da década de 60, quando dois cientistas noruegueses, Ole-Johan Dahl e Kristen Nygaard, criaram a linguagem Simula (*Simulation Language*). SIMULA I (1962-65) e Simula 67 (1967), as primeiras linguagens OO, eram usadas para simulações do comportamento de partículas de gases. Os conceitos de objetos, classes e herança já estavam presentes, embora não necessariamente com estes nomes. A herança apareceu apenas em Simula 67 e se falava apenas em criação de subclasses. Simula passou a ser usada com freqüência na década de 70, sendo inclusive a linguagem utilizada como base para desenvolver as primeiras versões de Smalltalk, logo no início da década de 70, quando surgiu o termo “Programação Orientada a Objeto” (POO).

Os principais conceitos de OO, descritos mais adiante, como por exemplo classes, objetos, encapsulamento, herança e polimorfismo, passaram a ser incorporadas nas linguagens de programação existentes, como por exemplo em Ada, BASIC, Lisp e Pascal. Mas adicionar características OO a linguagens que não haviam sido projetadas para esse fim levou a problemas de compatibilidade e manutenibilidade de código. Por outro lado, linguagens de programação puramente OO não possuíam certas características que os programadores passaram a exigir. Isso levou à criação de novas linguagens baseadas em métodos OO, mas permitindo algumas características procedimentais em locais seguros. Eiffel é um exemplo de linguagem relativamente bem sucedida criada com esses objetivos. Mais recentemente surgiram várias outras linguagens desse tipo, como por exemplo Python e Ruby. Além de Java, algumas das linguagens orientadas a objetos atuais com maior importância comercial são VB.NET e C#, projetada para a plataforma .NET da Microsoft.

Na metade da década de 80, quando as linguagens orientadas a objetos começaram a fazer sucesso, com grande influência de C++, que é uma extensão de C, surgiu a necessidade de processos para dar suporte ao desenvolvimento de software orientado a objetos. Na verdade, os processos de desenvolvimento utilizados na época eram os relacionados ao paradigma da análise estruturada. Realizar a análise estruturada [Yourdon, 1990] de um sistema e depois programá-lo usando uma linguagem orientada a objetos era muito trabalhoso, pois os modelos resultantes da análise estruturada precisavam de muitas adaptações até que pudessem ser implementados usando OO. Assim, o surgimento da orientação a objetos exigiu a criação de processos que integrassem o processo de desenvolvimento e a linguagem de modelagem, por meio de técnicas e ferramentas adequadas.

Abordagens como OMT [Rumbaugh, 1990], Booch [Booch, 1995] e Fusion [Coleman et al, 1994] foram as primeiras a surgir para dar suporte à análise e projeto OO. Elas estabelecem processos por meio dos quais é possível partir de um problema do mundo real, obter diversos modelos de análise, derivar os modelos de projeto e chegar às classes a serem implementadas.

Depois de quase uma década do surgimento das linguagens OO, estabeleceu-se uma gama enorme de processos de desenvolvimento OO, o que passou a dificultar a comunicação entre analistas e projetistas de software. Em geral, cada processo propunha sua própria notação para OO, com símbolos diferentes para denotar herança, agregação, associações etc. A UML (*Unified Modeling Language*) [Rational, 2000] – Linguagem de Modelagem Unificada – surgiu com o intuito de criar uma notação completa e padronizada, que todos pudessem usar para documentar o desenvolvimento de software OO. A Figura 1.1 mostra os modelos de cada um dos processos de análise existentes anteriormente e que serviram como base para a criação da UML (em itálico mostram-se os nomes dos diagramas correspondentes na UML).

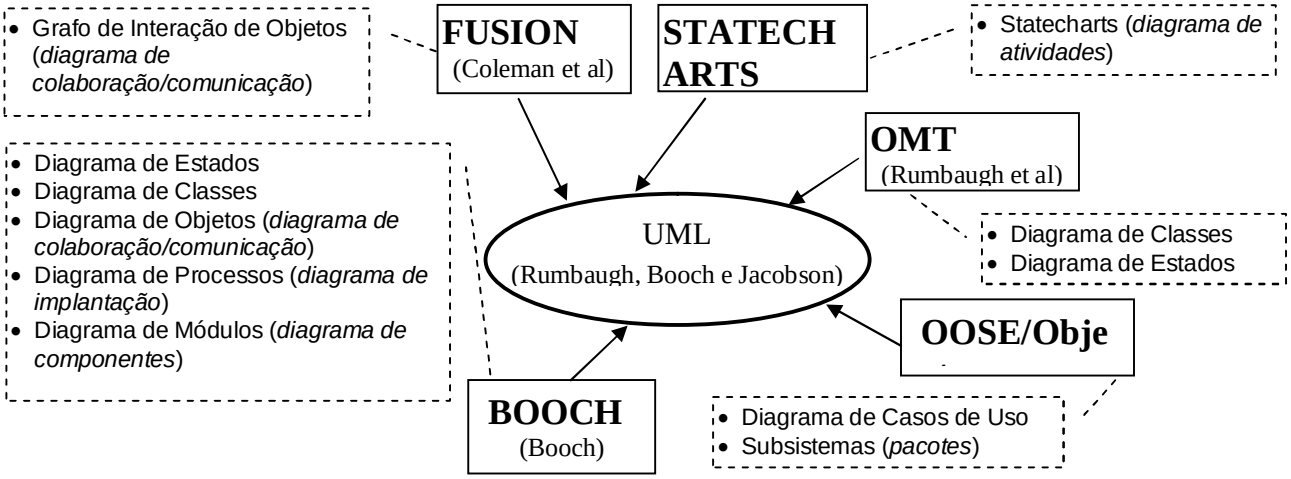


Figura 1.1 – Derivação da UML

No entanto, a UML não apresenta um processo, mas apenas a notação. Por isso, alguns anos depois de sua criação, passaram a surgir propostas de processos de desenvolvimento com base na UML. Exemplos são o PU (Processo Unificado) [Larman, 2004] e sua especialização pela Rational, o RUP (*Rational Unified Process*) [Krutchen, 2000]. Neste livro adotaremos o PU como sendo o processo pelo qual partimos dos requisitos de um software e chegamos ao código OO correspondente. O PU será explicado detalhadamente no Capítulo 2.

1.3 – Conceitos básicos de OO

Alguns conceitos básicos sobre o paradigma OO são introduzidos nesta seção e serão refinados ao longo do texto, na medida em que forem necessários. São conceitos

considerados fundamentais para o entendimento da abordagem OO, tais como classes, objetos, abstração, encapsulamento e herança.

1.3.1 – Classes e Objetos

O princípio mais básico da análise OO é a organização de **Objetos** do sistema em **Classes**. **Objetos** podem ser definidos como entidades individuais que tenham características e comportamento em comum. Por exemplo, o Leitor de nossa biblioteca possui atributos tais como nome, número-USP, data de nascimento, curso etc. Possui também comportamento, como registrar-se na biblioteca, emprestar livro, devolver livro etc. Quando se deseja falar de um conjunto de objetos semelhantes sem ter que falar de cada um individualmente, fala-se da **Classe** a que esses objetos pertencem. Por exemplo, podemos falar da classe Leitor, que contém todos os leitores da biblioteca do ICMC-USP. Cada leitor em particular, como por exemplo João da Silva, é dito ser um Objeto da Classe Leitor (ver Figura 1.2). Um outro exemplo: para representar a cozinha de uma casa por meio de um sistema orientado a objetos, o modelo teria objetos como Mesa, Cadeira, Geladeira, Fogão, Forno de Microondas, Armário, Balcão, Cozinheiro, Pessoa, Despensa, Utensílios, Alimento, etc. O Forno de Microondas possui atributos como capacidade, potência, status, hora, etc. Possui também comportamento, como ligar, programar, mudar potência, descongelar, desligar, etc. Em OO, o comportamento é implementado por meio de métodos de cada classe, o que será tratado no Capítulo 5.

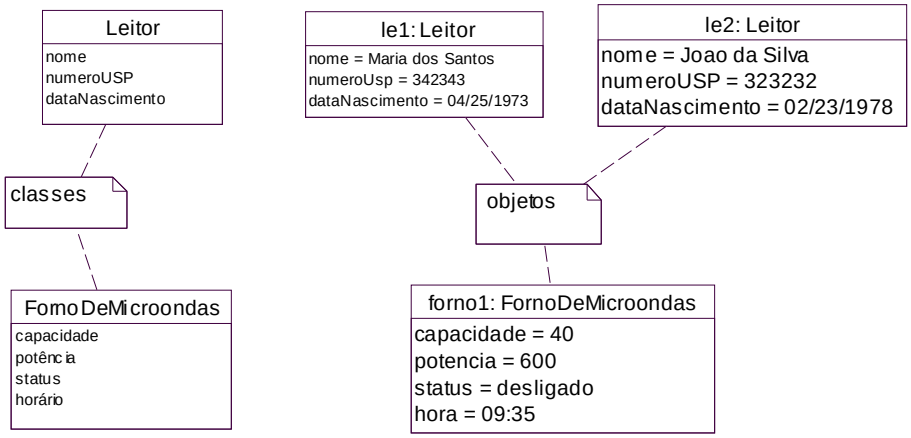


Figura 1.2 – Classes X Objetos

1.3.2 – Abstração

Abstração é o processo pelo qual conceitos gerais são formulados a partir de conceitos específicos. Em outras palavras, um conceito geral é formado pela extração de características comuns de exemplos específicos. Assim, em uma abstração, detalhes são ignorados, para nos concentrarmos nas características essenciais dos objetos de uma coleção. O conceito de abstração é vital na análise OO, pois permite a criação de classes por meio das características e comportamentos comuns a diversos objetos.

A abstração utiliza um estratégia de simplificação de detalhes. Como detalhes concretos podem permanecer ambíguos, vagos ou indefinidos, para que a comunicação em um nível abstrato tenha sucesso é necessário que haja uma experiência comum ou intuitiva entre os interlocutores. Por exemplo, é diferente uma conversa entre os arquitetos que irão projetar uma cozinha, preocupados com dimensões, usabilidade da cozinha etc., e entre o cozinheiro e o proprietário da casa, que estão mais preocupados com as receitas a serem elaboradas durante a semana.

A abstração nos leva a representar os objetos de acordo com o ponto de vista e interesse de quem os representa. Por exemplo, no caso de nossa cozinha, descrevemos um Forno por meio de sua capacidade, potência, status, hora, etc. Essas são as características que nos interessam para que o forno seja considerado útil na preparação de alimentos. Ao fazermos a identificação de um forno a partir dessas características, estamos fazendo uma abstração, já que não estamos considerando muitos outros detalhes necessários para descrever totalmente um Forno. Para um vendedor de Fornos, outras características seriam necessárias, como por exemplo, fabricante, preço, dimensões, voltagem, etc. Portanto, devemos utilizar apropriadamente o conceito de abstração na análise de sistemas, representando nos sistemas que vamos criar apenas aquelas características que nos interessam dos objetos reais, isto é, as características relevantes para o sistema a ser projetado e implementado.

1.3.3 – Encapsulamento

Outro conceito importante na orientação a objetos é o encapsulamento. Ele permite que certas características ou propriedades dos objetos de uma classe não possam ser vistas ou modificadas externamente, ou seja, ocultam-se as características internas do objeto. Por exemplo, no caso da Cozinha, há diversas propriedades do Forno que fazem com que ele cozinhe o alimento, mas elas ficam ocultas para os demais objetos, não importando como ocorrem, mas apenas que o resultado final seja obtido, que é o cozimento do alimento. Se for substituído o forno, todos os demais objetos continuam intactos – apenas muda a forma de cozimento internamente à classe Forno. No contexto de uma linguagem OO, o encapsulamento protege os dados que estão dentro dos objetos, evitando que eles sejam alterados incorretamente. A única forma de alterar esses dados é por meio de funções dos próprios objetos, que são chamadas de operações ou métodos.

1.3.4 – Herança

Herança é um mecanismo que permite que características comuns a diversas classes sejam colocadas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas (as subclasses). Cada subclasse apresenta as características (estrutura e métodos) da superclasse e acrescenta a elas novas características ou comportamento. Dizemos que uma subclasse **herda** todas as propriedades da superclasse e acrescenta suas características próprias e exclusivas. As propriedades da superclasse não precisam ser repetidas em cada subclasse.

Por exemplo, quando pensamos em um forno, logo nos vem à mente sua utilidade geral, que é cozinhar alimentos. Várias subclasses de Forno adicionam características próprias aos

diversos tipos de forno, como microondas, à gás, elétrico e à lenha, conforme ilustrado na Figura 1.3. Podemos criar diversos níveis de hierarquia de herança. Por exemplo, um forno pode ter vários sub-tipos, tais como microondas, à gás, elétrico e à lenha.

Outro recurso disponibilizado por algumas abordagens OO é a herança múltipla. Nesse caso, uma classe pode herdar de mais de uma classe, ou seja, um subclasse possui duas ou mais super-classes, das quais herda todos os atributos e comportamento. Na Figura 1.4 apresenta-se o caso em que um Leitor da Biblioteca possui como superclasses Estudante e Funcionário. Isso significa que ele herda os atributos de ambas as classes, além de ter um atributo próprio. Problemas podem ocorrer ao projetar sistemas OO usando herança, principalmente se existir comportamento com o mesmo nome em ambas as superclasses da classe com herança múltipla. Por exemplo, se tanto Estudante quanto Funcionário possuírem uma funcionalidade para “calcularDataDeDevolução”, cada qual retornando um valor diferente (por exemplo, dataDeHoje + 7 e dataDeHoje+15), e se a classe Leitor não definir novamente essa funcionalidade, ficará a cargo do compilador saber qual dos dois comportamentos será executado (provavelmente será escolhida uma das alternativas, para o caso do programador esquecer de informar qual delas quer usar).

Outras características importantes da orientação a objetos, como polimorfismo, agregação, etc., serão exploradas nos capítulos seguintes, na medida em que forem necessários para entendimento dos conceitos aos quais se relacionam.

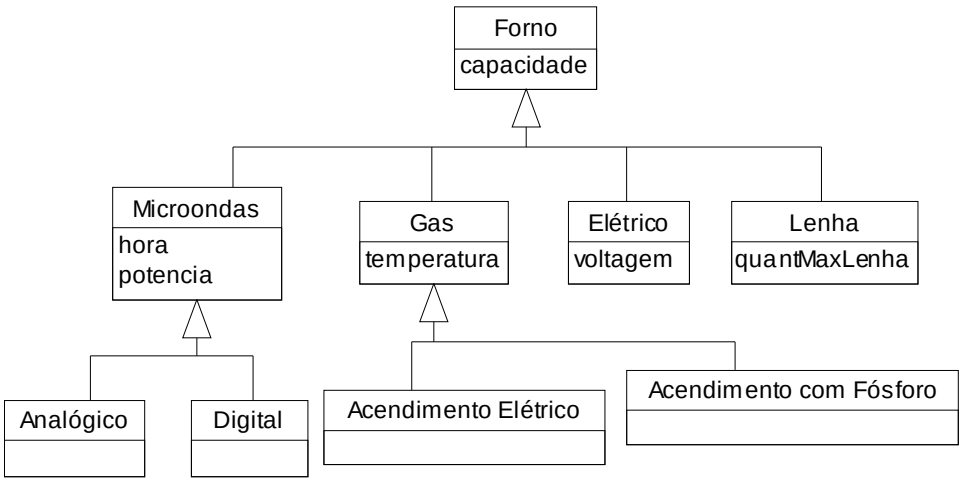


Figura 1.3 – Exemplo de Herança

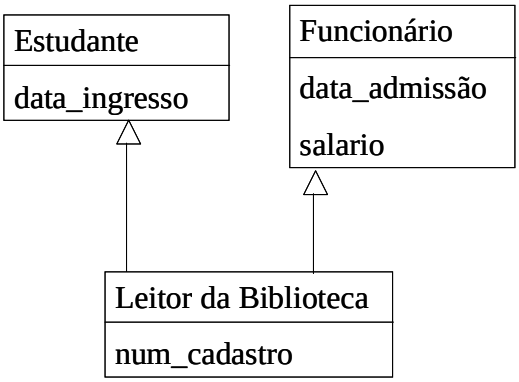


Figura 1.4 – Exemplo de Herança Múltipla

1.4 – Organização do Livro

O livro está organizado em duas partes. Na Parte I são apresentados todos os conceitos sobre análise e projeto OO usando UML, no contexto do Processo Unificado. Na Parte II apresenta-se um exemplo prático de aplicação dos conceitos, para um Sistema de Controle de Rodovias, mostrando-se os artefatos obtidos em cada fase de aplicação do Processo Unificado. Os apêndices contêm material complementar necessário para aprofundar-se nos detalhes sobre os exemplos fornecidos.

1.5 – Exercícios Complementares

1.5.1. Discuta porque é importante para uma empresa utilizar modelos de análise durante o desenvolvimento de software.
1.5.2. Comente a evolução do software que culminou com a abordagem orientada a objetos, citando suas principais vantagens em relação à abordagem estruturada.
1.5.3. Discuta a diferença entre classes e objetos, exemplificando.
1.5.4. Cite um exemplo em que a herança poderia ser utilizada na modelagem de um sistema de folha de pagamento.
1.5.5. Qual é o conceito na abordagem estruturada que deu origem ao conceito de encapsulamento? Descreva-o.
1.5.6. Cite qual é a abstração para cada um dos casos abaixo:
  1.5.6.1 – Brinquedo, Medicamento, Eletrodoméstico
  1.5.6.2 – Mão-de-obra de um mecânico, Consulta Médica, Massagem
  1.5.6.3 – Empréstimo de Livro, Locação de Fita e Hospedagem em Hotel
  1.5.6.4 – Reservar Sala, Agendar Horário Salão Beleza, Escolher Membro para Banca de Concurso

Capítulo 2 – Modelos de Processo de Desenvolvimento de Software e o Processo Unificado

2.1 – Modelos de Processo de Software

Desenvolver software é geralmente uma tarefa complexa e sujeita a erros. O processo que inicia quando surge a necessidade de desenvolver um software para resolver um determinado problema e termina quando o software é implantado, pode tomar diversos rumos, dependendo de inúmeros fatores que ocorrem durante todo o processo. Por exemplo, o processo pode ser mais ou menos eficiente de acordo com a forma como as tarefas são distribuídas aos responsáveis pelo desenvolvimento, os artefatos que devem ser produzidos ao final e as decisões tomadas ao longo do desenvolvimento, entre outros.

Ao longo dos anos, empresas desenvolvedoras de software têm documentado seus processos, dando origem a diversos modelos de processo de software, também chamados de paradigmas de desenvolvimento de software. Cada modelo de processo de software fornece uma representação abstrata de um processo em particular, que pode ser seguido pelo desenvolvedor der software. Na verdade, ele representa uma tentativa de colocar ordem em uma atividade inerentemente caótica, que é a de desenvolvimento de software. Como exemplos de modelos pode-se citar: Modelo em Cascata, Modelo de Prototipagem, Modelo Evolucionário, Desenvolvimento Baseado em Componentes e Modelo de Métodos formais.

Muitos desses modelos surgiram para aprimorar outro modelo existente, adaptando-o a novas situações e ambientes de desenvolvimento. Por exemplo, o modelo em cascata estabelece que o desenvolvimento inicia com a coleta de requisitos do sistema e prossegue com a análise, projeto, codificação e finalmente teste do sistema, conforme ilustrado na Figura 2.1. Esse modelo apresenta diversos problemas em decorrência de sua seqüência totalmente linear de atividades, o que não corresponde ao que ocorre na realidade de um projeto.

O problema mais grave ocorre quando mudanças nos requisitos são notadas em fases mais avançadas do processo. O retrabalho exigido para encaixar essas mudanças pode comprometer custos e cronogramas. O segundo problema está relacionado ao fato de que muitas vezes o usuário não consegue estabelecer os requisitos explicitamente no início do processo. Isso torna o modelo em cascata ineficiente, pois ele depende dos requisitos para prosseguir para as demais fases. Outro problema é que uma versão executável do sistema só será obtida no fim do processo e, portanto, o usuário não terá como implantar o sistema de forma incremental.

Por outro lado, o modelo em Cascata trouxe contribuições importantes para o processo de desenvolvimento de software: imposição de disciplina, planejamento e gerenciamento, além de fazer com que a implementação do produto seja postergada até que os objetivos tenham sido completamente entendidos.

O modelo de prototipagem surgiu para amenizar o segundo problema do modelo em Cascata pois, com a criação de protótipos durante o processo, o cliente tem mais chance de antever os resultados e melhor definir seus requisitos. A Figura 2.2 ilustra o modelo de prototipagem. Embora resolva um problema do modelo em cascata, o modelo de prototipagem introduz outros problemas: o cliente não sabe que o software que ele obtém em cada ciclo não considerou, durante o desenvolvimento, a qualidade global e a manutenibilidade a longo prazo. É bastante comum que o desenvolvedor faça uma implementação de baixa qualidade (incompleta e com baixo desempenho), com o objetivo de produzir rapidamente um protótipo. Assim, o cliente precisa estar consciente de que o protótipo servirá apenas como um mecanismo de definição dos requisitos, podendo ser descartado no final do ciclo, quando então o produto final será desenvolvido de acordo com todos os requisitos de qualidade exigidos.

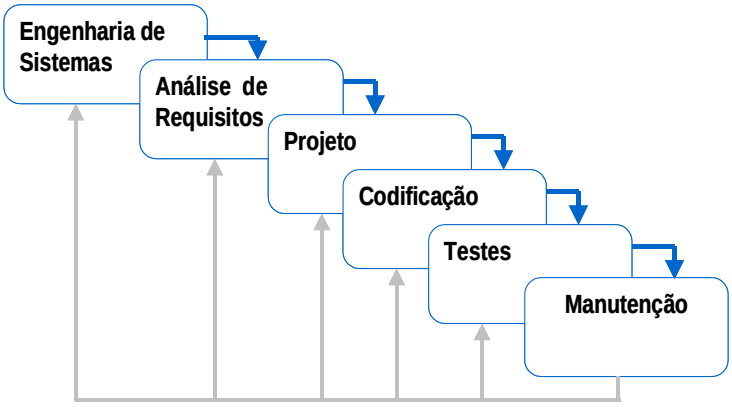


Figura 2.1 – Modelo em Cascata

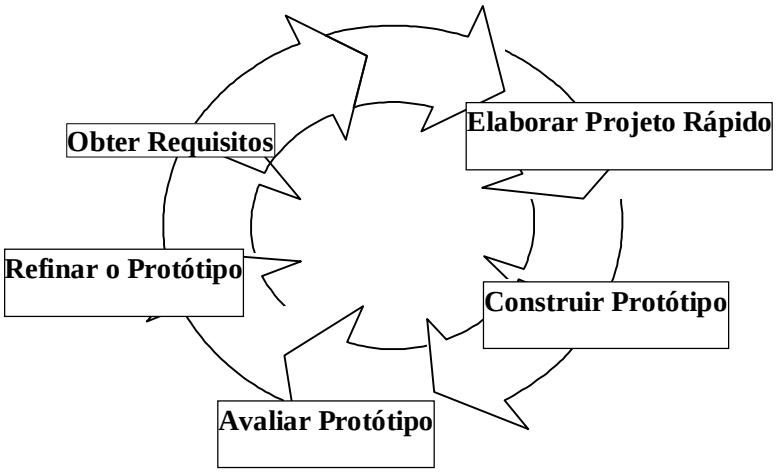


Figura 2.2 – Modelo de Prototipagem

2.2 – Modelo Incremental

O modelo incremental possui características compostas a partir dos modelos de processo de software mencionados na seção 2.1: o Modelo em Cascata e o Modelo de Prototipagem. A idéia é conduzir pequenos ciclos de desenvolvimento para descobrir os requisitos do usuário, cada qual produzindo incrementos ao sistema, de formapara que o sistema final seja composto desses incrementos. A Figura 2.3 ilustra o Modelo Incremental. Em geral, a versão inicial é o núcleo do produto (a parte mais importante). A evolução acontece quando novas características são adicionadas à medida que são sugeridas pelo usuário.

Este modelo é importante quando é difícil estabelecer a priori uma especificação detalhada dos requisitos, mas difere do modelo de prototipagem porque produtos operacionais são produzidos a cada iteração, ou seja, cada incremento resulta em produto que possui qualidade, embora não atenda a todas as funções desejadas. As novas versões podem ser planejadas para que os riscos técnicos possam ser administrados (por exemplo, de acordo com a disponibilidade de determinado hardware). Com o modelo incremental, tenta-se sanar os problemas dos modelos em cascata e de prototipagem. O desenvolvimento incremental resolve tanto o problema de não se conhecer todos os requisitos a princípio, pois pode-se identificá-los aos poucos durante os incrementos, quanto o problema do usuário ter que esperar até o fim do processo para ter o software executável, pois as diversas partes produzidas podem entrar em operação imediatamente, ou pelo menos servir para treinamento dos usuários e teste de sistema. Ao mesmo tempo, resolve o problema da qualidade dos protótipos produzidos, pois cada incremento é feito seguindo normas de qualidade.

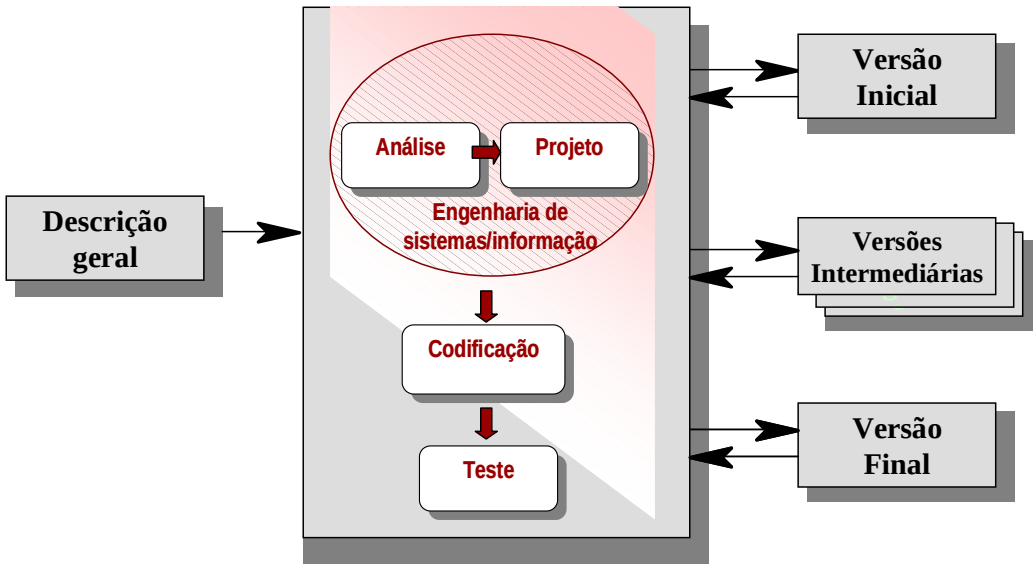


Figura 2.3 – Modelo Incremental

2.3 – O Processo Unificado (PU)

2.3.1 – Visão Geral

O Processo Unificado (PU) é um modelo de processo de software baseado no modelo incremental (seção 2.2), visando a construção de software orientado a objetos. Ele usa como notação de apoio a UML (*Unified Modeling Language*) [Rational, 2000]. O RUP (*Rational Unified Process*) é um refinamento detalhado do PU, proposta pela empresa IBM Rational [Krutchen, 2000]. A idéia mais importante do PU é o desenvolvimento iterativo. O desenvolvimento de um software é dividido em vários ciclos de iteração, cada qual produzindo um sistema testado, integrado e executável. Em cada ciclo ocorrem as atividades de análise de requisitos, projeto, implementação e teste, bem como a integração dos artefatos produzidos com os artefatos já existentes, conforme ilustrado na Figura 2.4.

No início do desenvolvimento, deve-se planejar quantos ciclos de desenvolvimento serão necessários para alcançar os objetivos do sistema, para que as partes mais importantes devem ser priorizadas e alocadas nos primeiros ciclos. Portanto, é de vital importância no PU que a primeira iteração estabeleça os principais riscos e o escopo inicial do projeto, de acordo com a funcionalidade principal do sistema. Deve-se evitar correr o risco de descobrir, em iterações posteriores, que o projeto é inviável. Para isso, as partes mais complexas do sistema devem ser atacadas já no primeiro ciclo, pois são elas que apresentam maior risco de inviabilizar o projeto.

O tamanho de cada ciclo pode variar de uma empresa para outra e conforme o tamanho do sistema. Por exemplo, uma empresa pode desejar ciclos de 4 semanas, outra pode preferir 3 meses. Produtos entregues em um ciclo podem ser colocados imediatamente em operação, mas podem vir a ser substituídos por outros produtos mais completos em ciclos posteriores. É claro que isso deve ser muito bem planejado pois, caso contrário, o esforço necessário para substituir um produto por outro pode vir a comprometer o sucesso do desenvolvimento incremental.

Um caso de uso (abordado no Capítulo 3) é uma seqüência de ações executadas de forma colaborativa entre o sistema e um ou mais usuários, para produzir um ou mais resultados valiosos para os interessados no sistema. O PU é dirigido por casos de uso, pois os utiliza para dirigir todo o trabalho de desenvolvimento, desde a captação inicial e negociação dos requisitos até a aceitação do código (testes). Os casos de uso são centrais ao PU e outros métodos iterativos, pois:

- Os requisitos funcionais são registrados preferencialmente por meio deles
- Eles ajudam a planejar as iterações
- Eles podem conduzir o projeto
- O teste é baseado neles

No PU, a arquitetura do sistema em construção é o alicerce fundamental sobre o qual ele se erguerá. Entende-se por arquitetura a organização fundamental de todo o sistema, o que inclui elementos estáticos, dinâmicos, o modo como trabalham juntos e o estilo arquitetônico total que guia a organização do sistema.

A arquitetura também se refere a questões como desempenho, escalabilidade, reuso e restrições econômicas e tecnológicas. Assim, a arquitetura deve ser uma das preocupações da equipe de projeto.

A arquitetura, juntamente com os casos de uso, deve orientar a exploração de todos os aspectos do sistema. Os motivos que fazem com que a arquitetura seja importante no PU são:

- Melhor entendimento da visão global do sistema. Essa visão é importante e não pode ser deixada de lado em nenhum momento, para que o projeto não se desvie do foco principal para o qual foi concebido.
- Melhoria da organização do esforço de desenvolvimento: por exemplo, a modularização pode diminuir a complexidade do sistema por meio de módulos mais gerenciáveis por equipes paralelas de desenvolvimento.
- Aumento da possibilidade de reuso – camadas podem ser projetadas para que possam ser reutilizadas em outros projetos
- Facilidade de evolução do sistema – um projeto com uma boa arquitetura pode ser mais flexível e previsível em relação a futuras mudanças
- Guia para seleção e exploração dos casos de uso – uma boa arquitetura facilita a alocação dos casos de uso nos diversos ciclos de desenvolvimento.

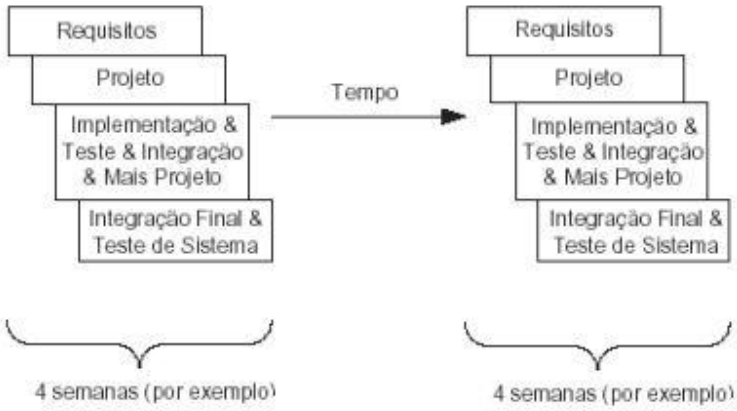


Figura 2.4 – Desenvolvimento Iterativo e Incremental [Larman, 2004]

2.3.2 – As Fases do PU

Cada um dos ciclos de desenvolvimento do PU é dividido em quatro fases: concepção, elaboração, construção e transição, conforme ilustrado na Figura 2.4. **Marcos referenciais** podem ser estabelecidos em um ou mais pontos, nos quais são tomadas decisões que permitem o prosseguimento normal do desenvolvimento ou podem exigir reajustes de cronograma e custos.

Na fase de concepção estabelece-se a viabilidade de implantação do sistema. Após definir o escopo do sistema, identificando as funcionalidades que pertencerão ou não a ele, pode-se fazer estimativas de custos e cronograma, além de identificar os potenciais riscos que

devem ser gerenciados ao longo do projeto. Faz-se ainda o esboço da arquitetura do sistema, que servirá como alicerce para a sua construção.

Na fase de elaboração cria-se uma visão refinada do sistema, com a definição dos requisitos funcionais, detalhamento da arquitetura criada na fase anterior e gerenciamento contínuo dos riscos envolvidos. Estimativas realistas feitas nesta fase permitem preparar um plano para orientar a construção do sistema.

Na fase de construção o sistema é efetivamente desenvolvido e, em geral, tem condições de ser operado, mesmo que em ambiente de teste, pelos clientes. É nesta fase que o desenvolvimento iterativo e incremental é realizado.

Na fase de transição o sistema é entregue ao cliente para uso em produção. Testes são realizados e um ou mais **incrementos** do sistema são implantados. Um incremento é a diferença entre entregas feitas em iterações uma seguida à outra. Defeitos são corrigidos, se necessário.

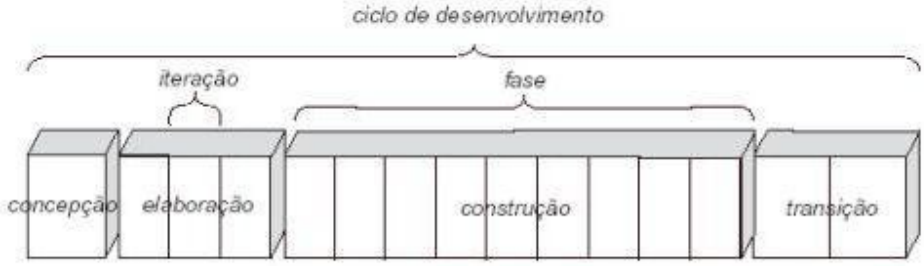


Figura 2.5 – Fases de um ciclo de desenvolvimento no PU [Larman, 2004]

2.3.3 – As Disciplinas do PU

Se analisarmos as fases do PU, podemos ter a impressão de que cada ciclo de iteração comporta-se como o modelo em Cascata, visto na seção 2.1. Mas isso não é verdade: paralelamente às fases do PU, atividades de trabalho, denominadas **disciplinas do PU**, são realizadas a qualquer momento durante o ciclo de desenvolvimento. Conforme ilustrado na Figura 2.6, as disciplinas entrecortam ortogonalmente todas as fases do PU, podendo ter maior ênfase durante certas fases e menor ênfase em outras, mas podendo ocorrer em qualquer uma delas. Considerando a disciplina "Projeto", nota-se no exemplo da Figura 2.6 que ela têm maior ênfase no fim da fase de elaboração e no início da fase de construção.

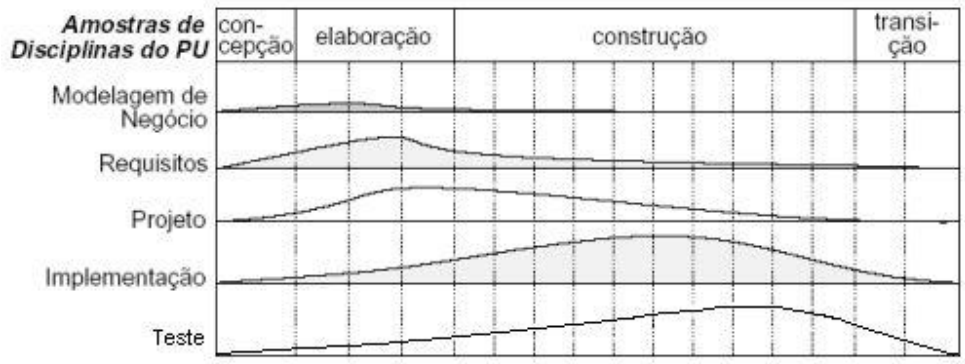


Figura 2.6 – Disciplinas e Fases do PU [Larman, 2004]

Como exemplo de disciplinas pode-se mencionar como as mais importantes: Modelagem de Negócio, Requisitos, Projeto, Implementação, Teste, Implantação, Gestão de Configuração e Mudanças, Gestão de Projeto e Ambiente. A ênfase ou carga de uma disciplina em cada fase pode variar de projeto para projeto. Algumas disciplinas podem ter grande parte de sua carga associada a uma das fases como, por exemplo, a disciplina "Implementação" tem grande ênfase na fase de Construção, enquanto que "Requisitos" é uma disciplina com pouca ênfase nessa fase. Outras disciplinas são importantes em várias fases, portanto, a ênfase é distribuída ao longo de todas elas, como por exemplo a disciplina Teste, cuja distribuição é praticamente igual nas fases de elaboração, construção e transição.

2.3.4 – Os Artefatos do PU

Cada uma das disciplinas do PU pode gerar um ou mais artefatos, que devem ser controlados e administrados corretamente durante o desenvolvimento do sistema. Artefatos podem ser quaisquer dos documentos produzidos durante o desenvolvimento, tais como modelos, diagramas, documento de especificação de requisitos, código fonte, código executável, planos de teste etc. Muitos dos artefatos são opcionais, sendo produzidos de acordo com as necessidades específicas de cada projeto.

A Tabela 1, baseada na sugestão de Larman [2004], mostra exemplos de alguns dos artefatos correspondentes a cada uma das disciplinas do PU, bem como as fases nas quais eles são produzidos inicialmente ou refinados. Há documentos produzidos que não se tornam artefatos e são descartados. Geralmente são documentos intermediários que apóiam alguma atividade ou a geração de outros documentos, como por exemplo formulários utilizados para entrevistas e planilhas para cálculos e estimativas.

Tabela 1 – Exemplos de Artefatos do PU (p – produzir, r – refinar)

Disciplina	Artefato *Iteração* →	Concepção C_1	Elaboração $E_1...E_n$	Construção $C_1...C_n$	Transição $T_1...T_n$
Modelagem de Negócio	Modelo Conceitual		p		
Requisitos	Diagrama de Casos de Uso	p	r		
	Casos de Uso Textuais	p	r		
	Diagrama de Seqüência do Sistema	p	r		
	Contratos para operações	p	r		
	Glossário	p	r		
Projeto	Diagrama de Classes		p	r	
	Diagrama de Colaboração		p	r	
	Diagrama de Pacotes		p	r	
	Documento de Arquitetura do Software		p		
Implementação	Código fonte			p	r

2.4 – Exercícios propostos complementares

2.4.1. Discuta a necessidade das empresas adotarem um processo de desenvolvimento de software.
2.4.2. Quais as características do PU que o fazem melhor que o Modelo em Cascata? E que o Modelo de Prototipagem?
2.4.3. Descreva os princípios básicos do PU e suas fases.
2.4.4. O que são disciplinas do PU? Como elas se combinam com as fases do PU?
2.4.5. Qual é a importância dos casos de uso no PU?

Capítulo 3 – Disciplina de Requisitos – Casos de Uso e Diagrama de Casos de Uso

3.1 – Dos requisitos para os casos de uso

Para que a análise orientada a objetos tenha sucesso, produzindo modelos que representem o mundo real o mais fielmente possível, é preciso conhecer detalhadamente o comportamento do sistema alvo. O documento de requisitos, em geral, é escrito de maneira informal e considerando os requisitos funcionais e não funcionais que o sistema deve satisfazer. No entanto, muitas vezes ele não existe ou existe mas não deixa claro o comportamento do sistema diante de cada evento que ocorre, nem tampouco a interação do sistema com o mundo exterior. Neste livro consideraremos que existe um documento de requisitos ou que ele pode ser produzido por meio de uma técnica de requisitos qualquer (não coberta aqui por não fazer parte do objetivo deste livro). Portanto, pressupomos que um documento de requisitos é dado como entrada para nosso processo.

Uma forma bastante intuitiva de documentar interação entre o sistema e o mundo exterior é por meio dos casos de uso. Assim, o primeiro passo para o desenvolvimento de um sistema é obter seu modelo de Casos de Uso, tomando como base o documento de requisitos elaborado anteriormente. Deve-se ressaltar que nem sempre somente o documento de requisitos é suficiente para desenvolver os casos de uso. Ele pode ser complementado com outras atividades, como entrevistas com os interessados, análise de documentos, análise de sistemas semelhantes etc. Conforme discutido no capítulo 2, o PU possui uma disciplina chamada “Requisitos”, para cuidar da especificação dos requisitos do sistema, e essa disciplina pode ser realizada durante as quatro fases (concepção, elaboração, construção e transição), embora com menor ênfase em algumas delas. Portanto, embora os casos de uso tomem como base o documento de requisitos, pode-se continuar a especificar as necessidades do usuário durante a elaboração detalhada dos casos de uso, já que essa elaboração exige um profundo entendimento das ações a serem executadas pelo sistema diante de cada situação possível.

Um **caso de uso** representa uma possível utilização do sistema por um **ator**, que pode ser uma pessoa, dispositivo físico, mecanismo ou subsistema que interage com o sistema alvo, utilizando algum de seus serviços. Por exemplo, um ator pode ser um funcionário que opera um sistema bancário, um sensor que avisa o sistema sobre a ocorrência de algum evento no mundo real, ou um subsistema que utiliza os serviços de um outro subsistema (por exemplo um subsistema de vendas de produtos utiliza os serviços de um subsistema de autorização de crédito para poder concluir a venda por cartão de crédito). Um caso de uso narra a interação entre o sistema e os atores envolvidos, para atingir um ou mais objetivos.

Um caso de uso pode conter vários **cenários** em que o sistema e os atores interagem, com sucesso ou fracasso, para atingir um objetivo. Cada cenário é uma seqüência específica de ações e interações entre os atores e o sistema alvo, ou um caso particular desde o início de uma interação até sua conclusão [Larman, 2004]. Por exemplo, o cenário em que o caixa efetua uma retirada de dinheiro de uma conta corrente, com sucesso, faz parte do caso de

uso “Efetuar retirada”, que pode ter outros cenários em que o cliente não tem saldo suficiente, ou desiste do saque antes da conclusão.

Elaborar os casos de uso a partir do documento de requisitos nem sempre é fácil. Dependendo do formato utilizado para documentar os requisitos, pode ser pouco intuitivo identificar os atores e os casos de uso com os quais interagem. Documentos de requisitos mais focados nos aspectos técnicos do sistema podem precisar de uma revisão antes de serem utilizados para derivar os casos de uso. Outra alternativa bastante viável é elaborar os casos de uso antes de produzir o documento de requisitos, ou seja, os casos de uso podem ser utilizados como uma forma de identificar os requisitos da aplicação, ou pelo menos ajudar nessa identificação. Por fim, pode-se fazer ambas as atividades em paralelo, ou seja, à medida que os requisitos são identificados, elaboram-se tanto os casos de uso quanto o documento de requisitos.

Deve-se ressaltar que o documento de requisitos é, muitas vezes, utilizado como um contrato entre desenvolvedor e cliente. Nesse caso, se, ao elaborar os casos de uso, forem descobertos problemas com o documento de requisitos (por exemplo, redundâncias, inconsistências, omissões, etc.), deve-se tomar alguma atitude para administrar a situação, provavelmente fazendo a substituição do documento por outro atualizado. Isso envolve reavaliar o contrato, e pode acarretar riscos, tais como não cumprimento de cronogramas e estimativas de custos.

3.2 – Identificação dos atores

Vários tipos de atores podem ser identificados em um sistema, de acordo com o tipo de interação que existe entre o ator e o sistema. Por exemplo, um Atendente que realiza o empréstimo de um livro em uma Biblioteca é considerado um **ator principal**, pois toda a comunicação com o sistema computacional é feita por ele. Já o Leitor que dirige-se ao balcão de empréstimo da Biblioteca para fazer o empréstimo é um **ator secundário**, pois interage com o sistema por intermédio do Atendente, mas não diretamente com o sistema computacional.

Ao analisar cada requisito do sistema, seja por meio da análise do documento de requisitos, ou seja diretamente a partir da necessidade do usuário (por exemplo, durante entrevista com o cliente que deseja desenvolver o sistema), deve-se observar atentamente quem são os atores que supostamente serão responsáveis, direta ou indiretamente, pela interação com o sistema a fim de realizar uma certa tarefa e atingir um objetivo do negócio. Muitas vezes vários atores estão envolvidos em uma mesma atividade (ou caso de uso), devendo-se classificá-los em primários e secundários. Deve-se ter cuidado para escolher apenas os atores que interessam ao sistema, isto é, que estão dentro das fronteiras do sistema. Por exemplo, em um sistema bancário, quando o Cliente deposita um cheque, que o Caixa enviará à Câmara de Compensação, apenas Cliente e Caixa fazem parte dos limites do sistema. A Câmara de Compensação está fora desses limites, e portanto não é de interesse para o sistema representá-la. Por outro lado, se o sistema faz o controle dos cheques compensados, ou se estivermos analisando um sistema de compensação, então a Câmara de Compensação poderia ser um ator relevante.

Terminada a busca por atores, produz-se uma lista de possíveis atores, que serão atribuídos aos casos de uso na próxima etapa. Entretanto, lembre-se que podem surgir novos atores durante o detalhamento dos casos de uso, como será discutido posteriormente. Por exemplo, ao analisar o subsistema de aquisição de livros no sistema de biblioteca, surgirá o ator "Bibliotecária Chefe".

3.3 – Identificação dos casos de uso

Identificados os atores, deve-se analisar a sua interação com o sistema para atingir os resultados esperados. Cada requisito do sistema deve ser analisado em busca dos eventos que ocorrem no mundo real e que dão origem a uma interação entre um ator e o sistema – os casos de uso. Conforme dito na Seção 3.1, um caso de uso representa uma possível utilização do sistema por um ator.

Por exemplo, em um sistema de biblioteca, o fato do leitor dirigir-se a um balcão para efetuar a retirada de um ou mais livros representa um caso de uso, que inicia no momento em que o leitor entrega ao atendente o(s) livro(s) que deseja emprestar e termina quando o atendente lhe entrega o(s) livro(s) devidamente autorizado(s) para retirada ou o avisa que não é possível emprestar. Várias ações intermediárias ocorrem durante esse caso de uso, como por exemplo, o leitor fornece sua identificação, o atendente verifica se o livro pode ser emprestado e se o leitor está apto a emprestar livros, o livro é desmagnetizado e a data de devolução é calculada de acordo com o tipo de leitor.

Cada requisito pode corresponder a um ou mais casos de uso e um caso de uso pode referir-se a um ou mais requisitos. Considere como exemplo os requisitos da Tabela 3.1. O caso de uso "Emprestar livro" corresponde aos requisitos R1, R2 e R3, simultaneamente. R3 poderá ter correspondência com outros casos de uso, como por exemplo "Devolver livro", porque quando o leitor devolve um livro deve-se verificar se sua situação precisa ser regularizada (caso ele esteja "não apto" em razão do atraso do livro que está sendo devolvido no momento).

Para garantir que todos os requisitos foram cobertos, sugere-se a construção de uma tabela que cruze as informações do documento de requisitos com os casos de uso, como ilustrado na Tabela 3.2. Isso também torna mais fácil manter a consistência entre o documento de requisitos e os casos de uso, sempre que alguma alteração ocorre em um dos dois, melhorando a **rastreabilidade** dos requisitos. Assim, se for mudado um requisito, sabe-se facilmente quais casos de uso devem ser revisados e vice-versa.

Tabela 3.1 – Parte dos requisitos para um Sistema de Biblioteca

R1. Para usar os serviços de uma biblioteca, os leitores deverão estar registrados e possuir um cartão com número de identificação e foto.

R2. O sistema deve permitir que um *leitor apto* empreste um ou mais livros, por um período de tempo que varia de 1 semana a 6 meses, dependendo do tipo de leitor (1 semana para estudantes de graduação, 15 dias para estudantes de pós-graduação e 6 meses para docentes).

R3. O leitor está apto a emprestar livros se não possuir em seu poder livros com data de devolução vencida (menor do que a data atual) e desde que o número de livros emprestados não ultrapasse o número máximo permitido, que depende do tipo de leitor (6 livros para estudantes de graduação, 10 livros para estudantes de pós-graduação e 15 livros para docentes).

Tabela 3.2 – Requisitos X Casos de Uso

Requisitos	Casos de Uso
R1, R2, R3	Emprestar livro Um leitor empresta um ou mais livros da biblioteca, por um período de tempo que depende do tipo de leitor
R1, R3, R4	Devolver Livro Um leitor devolve um livro que estava em seu poder, tornando-o novamente disponível para empréstimo
. . .	

Desenvolvedores inexperientes devem estar atentos para não confundir casos de uso com eventos ou operações do sistema (operações serão tratadas no capítulo 5). Por exemplo, “informar o código do leitor” ou “informar os livros a serem emprestados” não são casos de uso, mas operações executadas em resposta a eventos que ocorrem no sistema. Esses eventos fazem parte de um único caso de uso, “Emprestar Livros”, neste caso.

Outro ponto importante a discutir é se devemos considerar como casos de uso as diversas consultas e cadastros do sistema [Waslawick, 2004]. Muitos requisitos do sistema podem referir-se a simples operações de inclusão, alteração ou modificação de dados cadastrais, como por exemplo, os livros da biblioteca. Seria necessário considerar então três casos de uso: Incluir Livro, Alterar dados do livro e Excluir Livro? E quanto às consultas, por exemplo, Consultar Livro por Autor, Consultar Livro por Título, etc., seriam casos de uso? Não há uma resposta única a esta pergunta. Alguns autores preferem representar essas funções como casos de uso, pois ocorrem no sistema com freqüência, enquanto outros autores as consideram simples demais para serem representadas como casos de uso. De fato, consultas, inserções, alterações e exclusões possuem lógica simples e bem conhecida de todos, de forma que não precisam de um detalhamento maior na fase de análise do sistema. Ao mesmo tempo, são importantes para ter uma noção geral do escopo e tamanho do sistema.

Portanto, neste livro enumeraremos consultas, inserções, alterações e exclusões como casos de uso do sistema, representando-as nos diagramas de casos de uso (seção 3.4), mas não faremos o detalhamento desses casos de uso (seção 3.5) durante a análise do sistema. Com isso, teremos um diagrama de casos de uso que reflete as funcionalidades do sistema sem, no entanto, nos preocuparmos com a lógica interna de casos de uso triviais. Essa lógica poderá ser necessária durante o projeto do sistema, portanto prorrogaremos sua definição até lá.

3.4 – Diagramas de Caso de Uso em UML

Os Diagramas de Caso de Uso da UML têm por objetivo representar os atores e sua interação com o sistema em cada caso de uso. Um ator é representado por um "homem-palito" com o nome do ator ou por um retângulo com o estereótipo «ator» e o nome do ator, conforme ilustrado na Figura 3.1. Um **estereótipo** é um mecanismo que possibilita estender componentes da UML. Por meio de um estereótipo, pode-se dar maior destaque a um componente que tem semelhança com outros, mas que possui alguma(s) característica(s) que o distingue de outros elementos do mesmo tipo. Existem vários estereótipos pré-definidos em UML, como é o caso de «ator», mas o usuário pode definir seus próprios estereótipos.

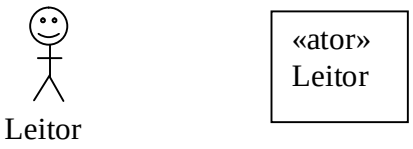


Figura 3.1 – Representações para o ator na UML

O homem palito é mais freqüentemente utilizado quando o ator é uma pessoa, mas nada impede utilizá-lo quando o ator é um dispositivo físico ou um subsistema (nestes casos é mais comum utilizar a segunda alternativa).

Os casos de uso são denotados por uma elipse com o nome do caso de uso dentro ou fora dela, conforme a Figura 3.2. A segunda alternativa é mais comum entre as ferramentas CASE, pois fica mais fácil redimensionar apenas o texto quando ele se encontra fora da elipse do que ter que redimensionar a elipse toda.

Figura 3.2 – Representações para o caso de uso na UML

Um Diagrama de Casos de Uso mostra os atores do sistema e os respectivos casos de uso dos quais participa, seja como ator principal ou secundário. A Figura 3.3 ilustra um Diagrama de Casos de Uso (parcial) para um sistema de biblioteca, no qual participam três atores: o Leitor, o Atendente e a Bibliotecária.

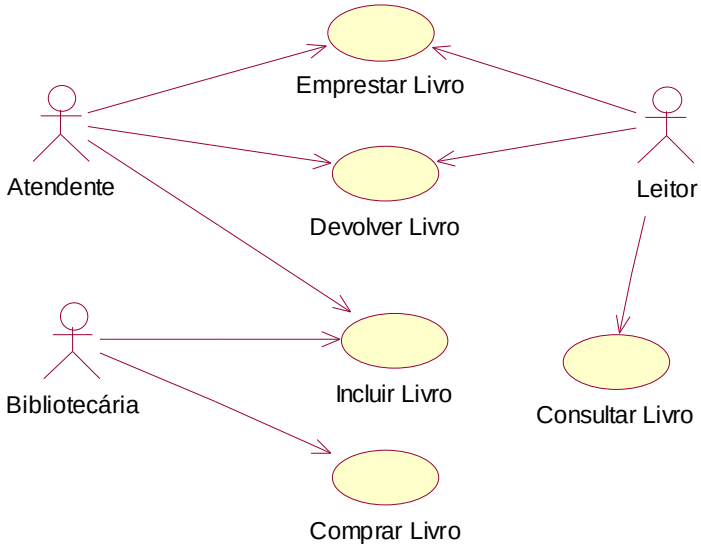


Figura 3.3 – Diagrama de Casos de Uso para sistema de biblioteca (parcial)

3.5 – Descrição dos Casos de uso

Enquanto os diagramas de caso de uso dão uma visão geral dos atores e de sua interação com o sistema, os casos de uso são descrições textuais dessa interação. O trabalho maior para elaboração dos casos de uso está justamente na escrita dos casos de uso, ao invés da construção dos diagramas em si. A descrição do caso de uso deve conter a seqüência de eventos que ocorrem tipicamente durante sua utilização, bem como as possíveis seqüências que ocorrem alternativamente, ou seja, se algumas ações podem dar errado e não ocorrer, ou acarretar em outras ações diferenciadas, tudo isso deve estar documentado no caso de uso. O exemplo da Figura 3.4 mostra uma sugestão para documentar um caso de uso, proposta por Larman [2004].

Na seção “Cenário de Sucesso Principal”, deve-se descrever o caso típico em que tudo corre bem, isto é, trata-se do caso mais otimista, em que tudo dá certo e o objetivo do caso de uso é alcançado. Na seção “Fluxos Alternativos” analisa-se cada linha do cenário de sucesso principal, vendo o que pode dar errado e qual seria o comportamento do sistema nesses casos. Por exemplo, na biblioteca, se tudo correr bem, o leitor está apto a emprestar e o livro está disponível e sem reserva. É justamente isso que deve estar refletido na seção “Cenário de Sucesso Principal”. Depois, cada possível exceção é descrita como Fluxo alternativo, por exemplo, o leitor tem livros em atraso e por isso não pode emprestar enquanto não regularizar sua situação, ou o livro está reservado para algum outro leitor.

Caso de Uso: Emprestar Livro

Ator Principal: Atendente

Interessados e Interesses:

- Atendente: deseja registrar que um ou mais livros estão em posse de um leitor, para controlar se a devolução será feita no tempo determinado.
- Leitor: deseja emprestar um ou mais livros, de forma rápida e segura.
- Bibliotecário: deseja controlar o uso dos livros, para que não se percam e para que sempre se saiba com que leitor estão no momento.

Pré-Condições: O Atendente é identificado e autenticado.

Garantia de Sucesso (Pós-Condições): Os dados do novo empréstimo estão armazenados no Sistema. Os livros emprestados possuem status "emprestado"

Cenário de Sucesso Principal:

1. O Leitor chega ao balcão de atendimento da biblioteca e diz ao atendente que deseja emprestar um ou mais livros da biblioteca.
2. O Atendente seleciona a opção para realizar um novo empréstimo.
3. O Atendente solicita ao leitor sua carteira de identificação, seja de estudante ou professor.
4. O Atendente informa ao sistema a identificação do leitor.
5. O Sistema exibe o nome do leitor e sua situação.
6. O Atendente solicita os livros a serem emprestados.
7. Para cada um deles, informa ao sistema o código de identificação do livro.
8. O Sistema informa a data de devolução de cada livro.
9. Se necessário, o Atendente desbloqueia os livros para que possam sair da biblioteca.
10. O Leitor sai com os livros.

Fluxos Alternativos:

(1-8). A qualquer momento o Leitor informa ao Atendente que desistiu do empréstimo.

3. O Leitor informa ao Atendente que esqueceu a carteira de identificação.
    1. O Atendente faz uma busca pelo cadastro do Leitor e pede a ele alguma informação pessoal para garantir que ele é mesmo quem diz ser.

4. O Leitor está impedido de fazer empréstimo, por ter não estar apto.
    1.Cancelar a operação.

7a. O Livro não pode ser emprestado, pois está reservado para outro leitor.
    1. O Atendente informa ao Leitor que não poderá emprestar o livro e pergunta se deseja reservá-lo.
    2. Cancelar a operação (se for o único livro)

7b. O Livro não pode ser emprestado, pois é um livro reservado somente para consulta.
    1. Cancelar a operação (se for o único livro)

Figura 3.4 - Caso de Uso Completo para "Emprestar Livro"

Quando algum passo do cenário de sucesso principal precisa ser repetido várias vezes, utiliza-se a expressão “Para cada”. Por exemplo, na Figura 3.4, o passo 7 é repetido para cada livro a ser emprestado pelo leitor. Pode-se também dar a noção de repetição de uma série de passos, por exemplo, um passo poderia ser redigido como “Repita os passos 7 a 9 para cada livro a ser emprestado pelo leitor”.

3.6 – Formatos de casos de uso: resumido X completo

Pode-se descrever um caso de uso com maior ou menor nível de detalhes, dependendo de sua complexidade e importância de seu detalhamento para a fase corrente do PU. Por exemplo, numa fase inicial de levantamento das funcionalidades do sistema, com objetivo de fazer estimativa de tempo e custo do projeto, pode-se utilizar o formato **resumido**, que consiste basicamente do nome do caso de uso e um resumo sucinto, de um parágrafo, descrevendo o caso otimista (cenário de sucesso principal), como ilustrado na Figura 3.5. Por outro lado, na fase de elaboração, o caso de uso precisa ser descrito no formato **completo**, para que os modelos produzidos com base neles reflitam o mais fielmente possível as funcionalidades desejadas.

Caso de uso: “Emprestar Livro”
Visão Geral: O Atendente da biblioteca realiza o empréstimo de um ou mais livros a um leitor apto a emprestar livros. O empréstimo é válido por um determinado período de tempo, de acordo com o tipo de leitor. Os livros são levados pelo leitor, depois de devidamente desmagnetizados, e marcados como “emprestados”.

Figura 3.5 - Caso de Uso Resumido para “Emprestar Livro”

Pode-se diferenciar os casos de uso completos em dois casos distintos: abstrato e concreto. Um caso de uso no formato **abstrato completo** é descrito de forma a retratar todas as funcionalidades desejadas, mas sem envolver na descrição aspectos técnicos dependentes de implementação. Por exemplo, o caso de uso da Figura 3.4 é abstrato, pois nada é dito a respeito da tecnologia envolvida na identificação do leitor ou do livro.

Por outro lado, um caso de uso no formato **concreto completo** descreve detalhes de implementação e técnicas de software/hardware utilizadas. No exemplo do empréstimo de livro, poderia ser fornecido um projeto da interface gráfica com o usuário (GUI), com detalhes de sobre como a leitura da identificação do leitor é feita (por exemplo, por código de barras), além da indicação de como os dados são informados em cada um dos campos da GUI. A Figura 3.6 ilustra o mesmo caso de uso, Emprestar Livro, na versão concreta.
Observe que o formato concreto completo possui detalhes que só são conhecidos mais à frente no processo de desenvolvimento, pois requer detalhes e decisões de projeto que são tomadas nas fases mais avançadas de elaboração e construção do PU. Já o formato abstrato completo pode ser escrito bem antes, pois pode-se manter a descrição o mais geral possível, possibilitando que as decisões que envolvem detalhes desconhecidos sejam tomadas posteriormente.

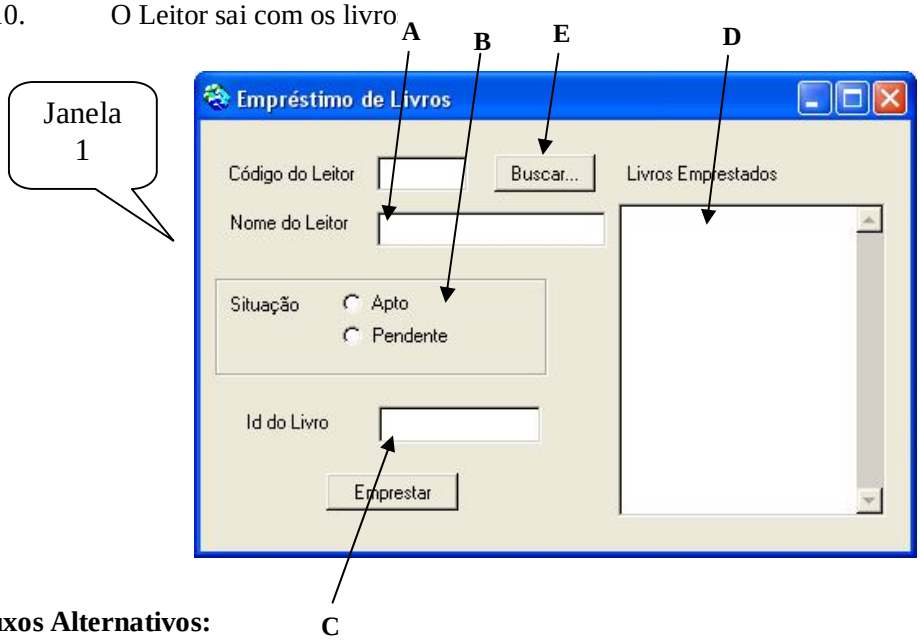
Caso de Uso: Emprestar Livro

Ator Principal: Atendente

. . .

Cenário de Sucesso Principal:

1. O Leitor chega ao balcão de atendimento da biblioteca e diz ao atendente que deseja emprestar um ou mais livros da biblioteca.
2. O Atendente seleciona a opção “Realizar um empréstimo” no menu principal do sistema de biblioteca.
3. O Atendente solicita ao leitor sua carteira de identificação, seja de estudante ou professor.
4. O Atendente passa a caneta leitora de código de barras na carteira de identificação.
5. O Sistema exibe nos campos A e B da Janela 1 o nome do leitor e sua situação.
6. O Atendente solicita os livros a serem emprestados.
7. Para cada um deles, o atendente lê o ISBN do livro por meio de seu código de barras, e o sistema exibe o ISBN no campo C da Janela 1. o Atendente clica no botão Emprestar para concretizar o empréstimo.
8. O Sistema exibe no campo D da Janela 1 o nome do livro e sua data de devolução.
9. O Atendente passa os livros pelo desmagnetizador para que possam sair da biblioteca.
10. O Leitor sai com os livro



Fluxos Alternativos:

(1-8). A qualquer momento o Leitor informa ao Atendente que desistiu do empréstimo.

3. O Leitor informa ao Atendente que esqueceu a carteira de identificação.
    1. O Atendente usa o botão E da Janela 1 para fazer uma busca pelo cadastro do Leitor. Pede ao leitor seu CPF para garantir que ele é mesmo quem diz ser. Caso não coincida o CPF, cancela a operação.

    . . .

Figura 3.6 - Caso de Uso Completo para “Emprestar Livro” (versão concreta)

3.7 – Relacionamentos entre casos de uso

Os casos de uso podem ter diversos tipos de relacionamentos uns com os outros. Por exemplo, pode ser que a execução de um caso de uso implique na execução de um outro. Ou pode ser que um caso de uso possua uma parte que se repete em outros casos de uso. Para evitar redundância de texto, pode-se isolar essas partes em casos de uso separados, e relacioná-los uns aos outros.

3.7.1 – O Relacionamento de Inclusão

Quando um caso de uso possui um comportamento parcial comum a vários outros casos de uso, pode-se criar um caso de uso separado para a parte comum e utilizar o relacionamento "incluir" entre eles. Esse relacionamento indica obrigatoriedade, ou seja, quando diz-se que um primeiro caso de uso inclui um segundo, necessariamente o segundo é executado quando o primeiro o é.

No diagrama de casos de uso em UML, o relacionamento de inclusão é feito por meio de uma seta tracejada entre os casos de uso, direcionada para o caso de uso incluído, e com o estereótipo «include» sobre a linha traçada, conforme mostrado na Figura 3.7. Neste exemplo, tanto as operações de saque quanto de depósito em uma conta bancária incluem o registro da movimentação da conta.

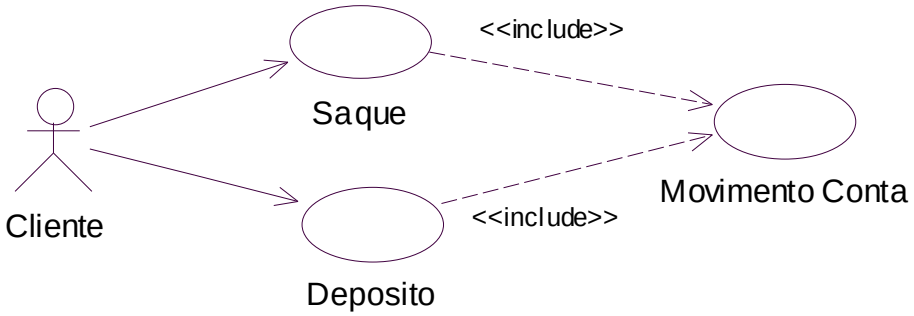


Figura 3.7 – Relacionamento "include"

Na descrição do caso de uso, o relacionamento de inclusão pode ser mostrado por meio de uma referência ao caso de uso incluído, seja no cenário de sucesso principal ou nos fluxos alternativos. Por exemplo, para o caso de uso Saque da Figura 3.7, que inclui o caso de uso Movimento Conta, a descrição poderia ser a da Figura 3.8.

Caso de Uso: Saque

. . .

Cenário de Sucesso Principal:

1. O Cliente chega ao balcão do Caixa e solicita uma quantia em dinheiro.
2. O Cliente entrega ao Caixa seu cartão magnético.
3. O Caixa escolhe a opção de saque e passa o cartão magnético pela leitora.
4. O Sistema exibe o nome do cliente e uma mensagem informando se seu saldo é suficiente para sacar a quantia solicitada.
5. Caso haja saldo suficiente, o Caixa entrega o dinheiro ao cliente.
6. Incluir o caso de uso: **Movimento Conta**.
7. O Cliente sai com o dinheiro.

. . .

Figura 3.8 - Caso de Uso com relacionamento Incluir

3.7.2 – O Relacionamento de Extensão

Muitas vezes é indesejável modificar um caso de uso para atender situações novas que podem surgir durante a análise do sistema. Pode-se desejar acrescentar algum comportamento ao caso de uso sem, no entanto, modificar o texto original. Nesses casos o relacionamento de extensão pode ser útil. Um caso de uso estende outro se ele adiciona comportamento ao caso de uso base. Quando um fluxo alternativo é complexo e merece maior detalhamento, pode-se escrevê-lo na forma de uma extensão ao caso de uso base.

Por exemplo, se o Leitor da Biblioteca esqueceu sua carteira de identificação e portanto precisa ser identificado de outra forma, pode-se fazer um caso de uso para tratar essa ocorrência e ligar esse caso de uso ao caso de uso Emprestar Livro por meio de um relacionamento de extensão, usando o estereótipo «extend», conforme ilustrado na Figura 3.9.

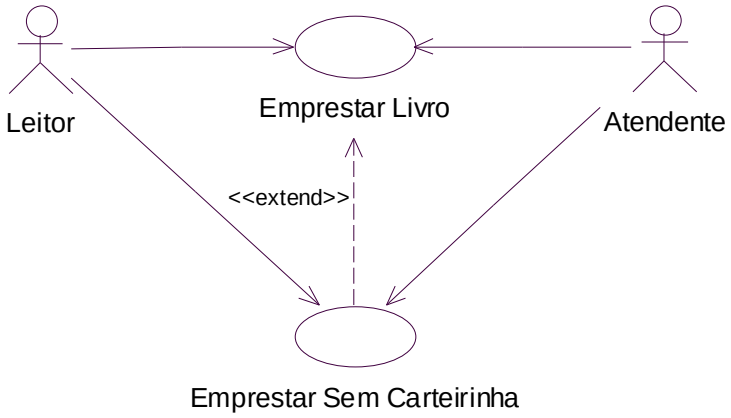


Figura 3.9 - Relacionamento Extend

3.8 – Ferramentas CASE para apoiar os casos de uso

Diversas ferramentas CASE (*Computer-Aided Software Engineering* – Engenharia de Software Auxiliada por Computador) estão disponíveis para dar suporte à UML. Além da funcionalidade de desenhar os diagramas, imprimi-los, exportá-los para outros editores, etc., uma ferramenta CASE oferece validações e consistência entre modelos e geração automática de código a partir de certos diagramas. Por exemplo, a partir do diagrama de classes, pode-se gerar o código fonte em Java ou C++ de cada classe e de seus métodos canônicos. Métodos canônicos são os métodos básicos que, idealmente, toda classe deve possuir, como por exemplo, um método para criar objetos da classe, para destruir um objeto, para atribuir e recuperar valor de cada um de seus atributos, entre outros.

Dentre as ferramentas CASE mais utilizadas atualmente, podemos citar a Rational Rose, a Visual Paradigm, a ArgoUML e a Poseidon. A ArgoUML é livre, podendo ser modificada para atender necessidades específicas. Porém, possui algumas limitações de desempenho. As demais ferramentas citadas devem ser licenciadas, mas existem versões demonstrativas que podem ser obtidas pela Internet. Neste livro será utilizada a ferramenta Rational Rose, de propriedade da IBM, que disponibiliza um programa acadêmico para convênio com Universidades e versões de avaliação pela Internet.

Para os casos de uso, a ferramenta Rose oferece um editor de diagramas de caso de uso, mas não oferece um gabarito para a sua descrição. Pode-se documentar essa descrição usando o campo de documentação, que está disponível para qualquer um dos elementos dos modelos, porém esse campo é um editor ASCII sem maiores recursos. O campo de documentação está disponível para quaisquer elementos gráficos, bastando clicar duplamente no elemento gráfico. Deve-se também registrar os requisitos que deram origem aos casos de uso, fazendo uma referência cruzada, conforme explicado na seção 3.3 (Tabela 3.2).

3.9 – Exercícios propostos

3.9.1. Dados os requisitos do sistema Passe-Livre (apêndice A), elaborar o Diagrama de Casos de Uso para o sistema.

3.9.2. Descreva no formato resumido pelo menos 6 casos de uso (os considerados mais importantes).

3.9.3. Monte uma tabela relacionando cada caso de uso aos requisitos a que ele se refere.

3.9.4. Alocar os casos de uso do diagrama elaborado no exercício 3.9.1 em 3 ciclos de desenvolvimento, considerando os conceitos do Processo Unificado.

3.9.5. Considerando que no exercício 3.9.1. um dos casos de uso definidos foi “Entrar na Autopista” elaborar sua descrição textual no formato completo abstrato.

3.9.6. Considerando que no exercício 3.9.1. um dos casos de uso definidos foi "Comprar Gizmo" elaborar sua descrição textual no formato completo concreto.

3.10 – Exercícios complementares

3.10.1. Observe o funcionamento de uma loja que aluga fitas de vídeos e DVDs para clientes cadastrados. Escreva um documento de especificação dos requisitos para um sistema que automatize lojas desse tipo. Dê ênfase aos requisitos funcionais.

3.10.2. Escreva um documento de requisitos de um sistema para automatização de uma biblioteca universitária. Use a biblioteca da sua Instituição como base e o formato similar ao utilizado no Apêndice A.

3.10.3. Faça o Diagrama de Casos de Uso do sistema de biblioteca especificado no item anterior. Para cada Caso de Uso identificado, descreva-o no formato resumido.

3.10.4. Considerando o sistema de bibliotecas especificado no exercício 3.10.3, elabore a especificação dos Casos de Uso no formato abstrato completo para Emprestar Livro e Devolver Livro.

3.10.5. Depois de ter feito a especificação indicada no exercício 3.10.3, vá até o sítio http://gnuteca.codigolivre.org.br e baixe o sistema Gnuteca, para automação de bibliotecas, distribuído na forma de software com código aberto. Instale esse software em seu computador ou no laboratório, execute-o e verifique quais das funcionalidades especificadas por você são atendidas por esse software.

3.10.6. Entre no sítio virtual da empresa Amazon.com e simule o processo de compra de um livro. Depois, escreva um caso de uso no formato concreto completo que especifica essa transação comprar livro. Você pode iniciar a partir do ponto em que os livros estão selecionados e incluídos na lista de compras. O nome do caso de uso será Comprar Livros. Em seguida, a partir do caso de uso concreto completo, escreva um caso de uso abstrato completo. Opcionalmente, tente escrever o caso de uso abstrato completo diretamente da análise (engenharia reversa) da interface do sistema da Amazon.

3.10.7. Repita o exercício 3.10.6 para uma livraria online brasileira (por exemplo: Saraiva ou Cultura). Compare os casos de uso resultantes.

3.10.8. Elabore o Diagrama de Casos de uso e especifique os casos de uso principais do sistema de loja de videolocadora, a partir da especificação elaborada na questão 1.

3.10.9. Considere a descrição abaixo, de um sistema de suporte ao departamento de obras da prefeitura de uma cidade (exemplo adaptado a partir de um exercício extraído do livro de Pressman [2002]):

"Os cidadãos podem obter acesso a um sítio da Web e relatar a localização e gravidade dos buracos que estão nas ruas da cidade. À medida que os buracos são relatados eles são registrados num "sistema de reparo do departamento de obras públicas" e lhes é atribuído um número de identificação, armazenado por endereço da rua, tamanho (numa escala de 1 a 10), localização (no meio da rua, na calçada, etc.), distrito (determinado pelo endereço da rua) e prioridade de reparo (determinada pelo tamanho do buraco). Para cada buraco é aberta uma ordem de serviço, que será atribuída a uma equipe responsável por consertá-lo (é importante informar o número de pessoas na equipe) e será decidido qual equipamento é necessário para o reparo. Depois de consertado, são informados o número de horas aplicadas no reparo, o estado do buraco (trabalho em andamento, reparado, reparo temporário, não reparado), quantidade de material de enchimento usado e custo do reparo (calculado a partir de horas aplicadas, quantidade de pessoas, material e equipamento usados). Finalmente, um arquivo de danos é criado para conter informação sobre danos relatados devido ao buraco e incluem nome do cidadão, endereço, número do telefone, tipo de dano e quantia em reais de prejuízo causado pelo dano. O SARB é um sistema on-line; todas as consultas devem ser feitas interativamente."

Transforme essa descrição em documento de especificação de requisitos, em que os requisitos estão todos escritos da forma "O sistema deve.fazer...xxxx", conforme modelo do Apêndice A.

3.10.10. Identifique os casos de uso do sistema de reparos de buracos e elabore o seu diagrama de casos de uso. Escolha a função principal do sistema e especifique seu caso de uso no formato abstrato completo.

3.10.11. Especifique o caso de uso no formato abstrato completo para "Sacar dinheiro de uma ATM". ATM (*Automatic Teller Machine*) é o nome dado aos caixas eletrônicos. Usar uma ATM qualquer como base.

3.10.12. Elabore o documento de requisitos para um sistema do tipo "disque-denúncia". Considere, por exemplo, o Disque-trote de uma Universidade, em que um aluno ou cidadão denuncia a atitude violenta dos veteranos em relação aos calouros. Deve ser feito o registro do fato e deve ser aberto um processo para averiguar a veracidade da informação e punir os responsáveis. Esse processo deve passar por várias situações até que seja encerrado, por exemplo: novo, em averiguação, em tratamento e encerrado. Utilize como modelo o Apêndice A.

3.10.13. Com base no documento de requisitos do sistema Disque-Trote, elabore o seu diagrama de casos de uso e escolha algumas funções importantes do sistema para especificar o caso de uso no formato abstrato completo.

Capítulo 4 – Disciplina de Requisitos – Modelo Conceitual

4.1 – Conceitos e atributos

Conforme visto no Capítulo 1, o objetivo da disciplina de requisitos é produzir modelos que representem o sistema a ser desenvolvido. O Modelo Conceitual têm por objetivo mostrar todos os conceitos importantes no domínio do sistema, bem como as associações entre esses conceitos. A idéia é fazer com que o usuário que tem acesso a esse modelo entenda os principais elementos do domínio que estão envolvidos no sistema a ser desenvolvido.

O modelo conceitual fornece um panorama geral do sistema, podendo servir como um ponto inicial para começar a entender o escopo do sistema e seu funcionamento global. Assim, alguém que não conhece o domínio do sistema pode utilizar o modelo conceitual para ter uma idéia dos principais conceitos e seus relacionamentos.

4.1.1 – Como identificar conceitos

A identificação de conceitos do domínio do sistema não é uma tarefa trivial, principalmente para novatos em análise orientada a objetos. Alguns conceitos são bastante óbvios quando se descreve o sistema, como por exemplo os conceitos Leitor e Livro quando se trata de um sistema de Biblioteca. Entretanto, muitos conceitos estão geralmente implícitos e requerem uma maior maturidade do analista para descobri-los. Por exemplo, o conceito de Empréstimo, no mesmo sistema de Biblioteca, poderia não ser modelado como um conceito, mas sim como uma associação entre Leitor e Livro (associações são tratadas na Seção 4.2). Isso não é totalmente errado, mas as informações sobre o empréstimo em si ficam difíceis de representar por meio de uma simples associação.

Por isso, existem algumas sugestões básicas para a correta identificação dos conceitos do sistema.

> **Passo 1**: Comece isolando, no documento de requisitos ou na descrição dos casos de uso elaboradas na fase anterior, todos os substantivos presentes no texto. Eles são candidatos a conceitos. Por exemplo, retorne à Figura 3.4. Analisando o cenário de sucesso principal, pode-se isolar diversos substantivos, conforme mostrado na Figura 4.1.

1. O Leitor chega ao balcão de atendimento da biblioteca e diz ao atendente que deseja emprestar um ou mais livros da biblioteca.
2. O Atendente seleciona a opção para adicionar um novo empréstimo.
3. O Atendente solicita ao leitor sua carteirinha, seja de estudante ou professor.
4. O Atendente informa ao sistema a identificação do leitor.
5. O Sistema exibe o nome do leitor e sua situação.
6. O Atendente solicita os livros a serem emprestados.
7. Para cada um deles, informa ao sistema o código de identificação do livro.
8. O Sistema informa a data de devolução de cada livro.
9. O Atendente desbloqueia os livros para que possam sair da biblioteca.
10. O Leitor sai com os livros.

Figura 4.1 – Trecho de um caso de uso com os substantivos sublinhados

Passo 2: Considere cada um dos substantivos isoladamente e verifique se são relacionados a assuntos importantes no domínio do sistema. Muitos deles podem ser descartados simplesmente porque fogem do escopo do sistema, ou porque são similares a outros conceitos já identificados, ou porque são meramente propriedades (atributos, conforme será visto na Seção 4.1.2) de outros conceitos. Por exemplo, na Figura 4.1, podemos descartar *balcão e opção* pelo primeiro motivo e *identificação do leitor, nome do leitor, situação*, *código de identificação do livro* e *data de devolução* pelo terceiro motivo. *Carteirinha* pode ser descartada por ser um mero instrumento para identificar o leitor, portanto o interesse na carteirinha se deve ao código que ela possui e que identifica o leitor.

Passo 3: Isole agora os verbos que poderiam ser transformados em substantivos (possivelmente com a ajuda de outras palavras). Concentre-se nos verbos que representam ações de interesse para o sistema, ou seja, aqueles relacionados a eventos e transações que possuem informações importantes e que devem ser lembrados pelo sistema. Por exemplo, na Figura 4.2 há vários verbos sublinhados, mas somente um deles, "emprestar", é candidato a conceito no sistema de Biblioteca. Como a palavra empréstimo já havia sido sublinhada no passo 1, a adição desse verbo não representa nenhum novo conceito nesse exemplo.

Passo 4: Para cada candidato a conceito, verifique se ele é composto de outras partes que sejam de interesse para o sistema, mesmo que elas não apareçam explicitamente no texto. Por exemplo, um empréstimo normalmente refere-se a vários livros emprestados em uma mesma ocasião para um mesmo leitor. Assim, pode-se dizer que o empréstimo é dividido em vários itens de empréstimo, um para cada livro emprestado. Portanto, ItemDoEmpréstimo é um outro conceito. Outro exemplo: Um conserto de carro engloba tanto os serviços de mão-de-obra executados quanto a venda das peças substituídas para realizar o conserto. Portanto ServiçoExecutado e PeçaVendida também são conceitos.

1. O Leitor chega ao balcão de atendimento da biblioteca e diz ao atendente que deseja emprestar um ou mais livros da biblioteca.
2. O Atendente seleciona a opção para adicionar um novo empréstimo.
3. O Atendente solicita ao leitor sua carteirinha, seja de estudante ou professor.
4. O Atendente informa ao sistema a identificação do leitor.
5. O Sistema exibe o nome do leitor e sua situação.
6. O Atendente solicita os livros a serem emprestados.
7. Para cada um deles, informa ao sistema o código de identificação do livro.
8. O Sistema informa a data de devolução de cada livro.
9. O Atendente desbloqueia os livros para que possam sair da biblioteca.
10. O Leitor sai com os livros.

Figura 4.2 – Trecho de um caso de uso com os verbos sublinhados

Mesmo seguindo os passos acima, alguns conceitos poderão vir a ser descobertos só mais à frente, na etapa de projeto. Isso ocorre porque muitos conceitos são difíceis de serem descobertos usando os documentos existentes até então, mas que são mais facilmente identificados durante o projeto da colaboração entre os objetos para cumprir as responsabilidades necessárias ao funcionamento do sistema.

4.1.2 – Como identificar Atributos

Conforme mencionado no passo 1 (seção 4.1.1), vários dos substantivos sublinhados a partir do texto do caso de uso podem ser candidatos a atributos dos conceitos já identificados. É necessária uma certa cautela ao incluir os atributos, para não tornar o modelo conceitual muito complexo desnecessariamente. Portanto, limite-se a adicionar os atributos importantes para compreender o real sentido de cada conceito, ou atributos que serão importantes para o futuro projeto do sistema, por se referirem à propriedades do conceito que tem papel definido no cumprimento das funcionalidades.

Por exemplo, os atributos *identificação do leitor*, *nome do leitor*, *situação*, *código de identificação do livro* e *data de devolução* devem ser adicionados aos respectivos conceitos, pois serão utilizados de alguma forma no projeto da parte lógica do sistema.

4.1.3 – Representação na UML

Conceitos são também chamados de classes conceituais do sistema e, na UML, utiliza-se a mesma notação do diagrama de classes, que será visto no Capítulo 8. A Figura 4.3 ilustra a notação para Conceitos em UML, referente aos exemplos mencionados nas sub-seções anteriores: um retângulo com uma divisória, na qual a parte superior contém o nome do conceito e a parte inferior os atributos do conceito.

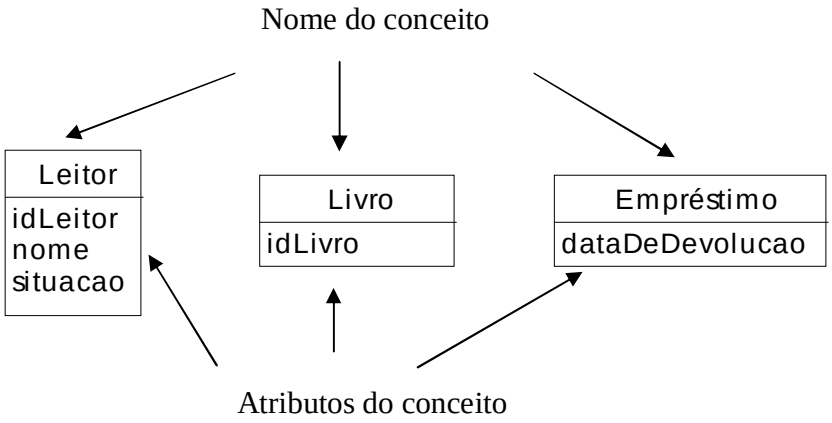


Figura 4.3 – Conceitos em UML

4.2 – Associações

Conceitos são importantes para entender o domínio do sistema. Analisar os relacionamentos entre conceitos do sistema é de grande ajuda para melhor compreender as conexões existentes entre as partes que compõem o sistema. Uma associação pode ser definida como um relacionamento entre conceitos, que precisa ser lembrado pelo sistema durante seu funcionamento. Por exemplo, existe uma associação entre Empréstimo e Leitor, que precisa ser lembrada pelo sistema, pois quando o Empréstimo termina o Leitor e o Livro devem ter sua situação regularizada.

4.2.1 – Como identificar associações

Assim como os conceitos, existem associações que são bastante simples de serem identificadas, bastando aplicar certas regras básicas. Outras associações são implícitas e podem requerer mais experiência do analista para identificá-las logo na disciplina de requisitos, ou podem surgir mais adiante, na disciplina de projeto.

Da mesma forma que introduzir muitos atributos pode ser prejudicial para o entendimento do modelo conceitual, incluir associações em demasia também causa um efeito indesejado, levando a um modelo confuso e conseqüentemente com pouca legibilidade. Portanto, uma regra básica é evitar incluir associações redundantes ou que podem ser derivadas a partir de outras.

Três regras que podem ajudar na identificação de associações são:

Regra 1: Um conceito que, fisicamente ou logicamente, faz parte de outro. Por exemplo, um livro que está fisicamente armazenado em uma estante ou um ItemDeEmpréstimo que logicamente faz parte do Empréstimo.

Regra 2: Um conceito que serve para descrever ou qualificar outro conceito. Por exemplo, um Livro pode ser classificado em diversas Categorias ou por Autor; um Item de Estoque que é descrito por uma Especificação de Produto.

Regra 3: Um conceito que é responsável por registrar ou manter informações sobre outro. Por exemplo, o Atendente é quem registra e atende o Leitor; a Bibliotecária é responsável pelos Livros.

Regra 4: Os verbos sublinhados nos casos de uso (Passo 3 da Seção 4.1.1) podem indicar associações entre conceitos (por exemplo, o Atendente é quem empresta o Livro ao Leitor).

4.2.2 – Representação na UML

A Figura 4.4 ilustra a notação UML para associação entre dois conceitos. Uma linha contínua liga os dois conceitos. Uma etiqueta com o nome da associação é colocada sobre ou sob a linha (de preferência de maneira centralizada). Nos extremos de cada lado da associação é colocada a multiplicidade da associação, conforme explicado na seção 4.2.3. A direção de leitura de uma associação é da esquerda para a direita e de cima para baixo. Por exemplo, na Figura 4.4 lê-se: Atendente registra Leitor. Caso o nome da associação tenha sido escrito pensando em um sentido diferente do tradicional, deve-se colocar um pequeno triângulo para denotar que a leitura é num sentido não convencional, conforme mostrado na Figura 4.5, onde lê-se: Leitor é registrado por Atendente.

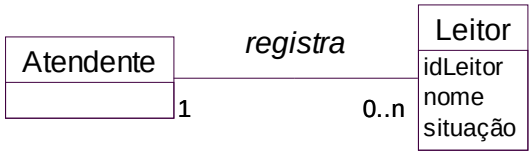


Figura 4.4 – Associação em UML

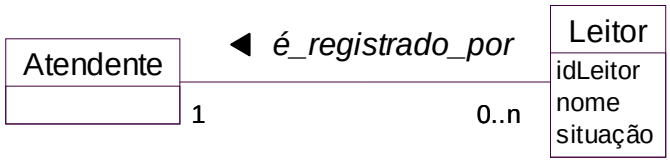


Figura 4.5 – Associação em UML (leitura no sentido contrário)

4.2.3 – Multiplicidade

Ao definir uma associação, é importante saber quantos objetos de uma classe C1 estão (ou podem estar) associados a quantos objetos de uma classe C2. Esse conceito é conhecido como **multiplicidade** da associação, e é denotado por meio de uma anotação em cada um dos lados da associação.

Por exemplo, nas Figuras 4.4. e 4.5, próximo ao conceito Atendente encontra-se o valor 1 e próximo ao conceito Leitor encontra-se o valor 0..n. O significado desses valores pode ser encontrado na Figura 4.6. A leitura da multiplicidade é feita sempre considerando um (1) objeto do lado oposto à multiplicidade analisada e contrapondo esse objeto ao número de objetos que podem existir relacionados a ele em um dado momento. Por exemplo, na Figura 4.4 lendo-se a multiplicidade da esquerda para a direita tem-se "Um Atendente registra 0 (zero) ou mais leitores" e lendo-se da direita para a esquerda tem-se "um Leitor é registrado por um Atendente". Pode-se também fazer a leitura do ponto de vista de objetos, ou seja, um objeto Atendente registra vários objetos Leitor e um objeto Leitor é registrado por um objeto Atendente. Com isso, consegue-se restringir os tipos de associações que podem ser estabelecidas entre os objetos de cada conceito. É importante ressaltar que algumas ferramentas CASE, como é o caso da versão mais antiga da IBM Rational Rose, não utilizam o n para denotar muitos objetos, mas o asterisco ("*"). AUML é flexível neste ponto, deixando para o usuário as duas alternativas.

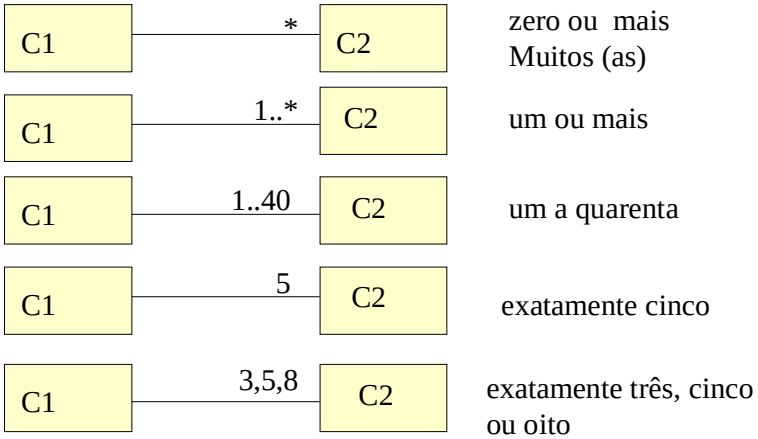


Figura 4.6 – Multiplicidades em UML

4.2.4 – Associação Reflexiva

É comum precisarmos denotar a associação entre objetos de uma mesma classe, o que é chamado de associação reflexiva. Isso pode ser feito com uma linha ligando o objeto a ele mesmo, como mostrado na Figura 4.7, que mostra que um objeto da classe Pessoa, pode ter uma associação de paternidade com zero ou mais pessoas dessa mesma classe. Uma anotação especial é colocada juntamente com a multiplicidade, chamada de **papel**, que denota o papel desempenhado pelo objeto de cada lado da associação. No exemplo da Figura 4.7, um objeto da classe Pessoa, com o papel de "pai", é pai de zero ou mais objetos dessa mesma classe, que desempenham o papel de "filhos".

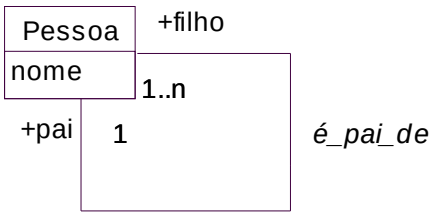


Figura 4.7 – Associação Reflexiva

Uma última observação é que o conceito de papéis pode ser utilizado normalmente em associações não reflexivas, se for importante estabelecer o papel de cada objeto na classe origem e destino da associação, como no exemplo da Figura 4.8.

4.3 – Tipo Associativo

Um tipo associativo é uma associação que também possui propriedades de classe (ou uma classe que tem propriedades de uma associação). É mostrada como uma classe, ligada por uma linha tracejada a uma associação, conforme mostrado na Figura 4.8, na qual o emprego que uma pessoa possui em uma dada empresa é modelado como um tipo associativo. O emprego poderia ter sido modelado simplesmente pela associação entre empresa e Pessoa, que é do tipo "muitos para muitos", já que uma empresa emprega várias pessoas e uma pessoa pode ter empregos em diversas empresas.

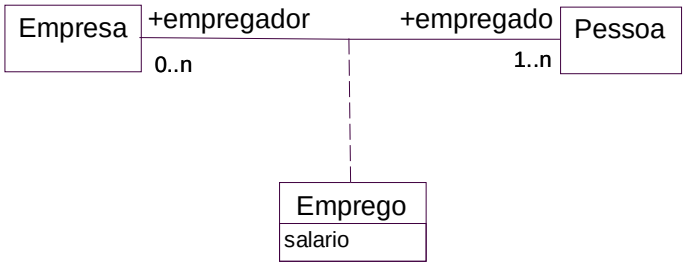


Figura 4.8 – Tipo Associativo

Alguns indícios de que um tipo associativo pode ser útil em um modelo conceitual são:

- um atributo está relacionado com uma associação, ou seja, o atributo não é inerente a nenhum dos conceitos nas extremidades da associação, mas parece pertencer à associação em si. Por exemplo, se não houvesse o tipo associativo Emprego na Figura 4.8, seria difícil decidir onde colocar o atributo "salário", pois como uma pessoa tem vários empregos, o salário é diferente em cada um deles. Além disso, também não se pode colocá-lo na empresa, pois o salário é diferente para cada funcionário.
- as instâncias do tipo associativo têm um tempo de vida dependente do tempo de vida da associação. Por exemplo, na Figura 4.8, um emprego só faz sentido entre Empresa e Pessoa associadas.

- existe uma associação muitos-para-muitos entre dois conceitos, bem como informações relacionadas à associação propriamente dita, que é o caso do exemplo da Figura 4.8. Ao invés de usar o tipo associativo, poderia ter sido criado o conceito Emprego, removendo-se a associação entre Empresa e Pessoa e criando duas associações um-para-muitos, uma entre Empresa e Emprego e outra entre Pessoa e Emprego, conforme mostrado na Figura 4.9.

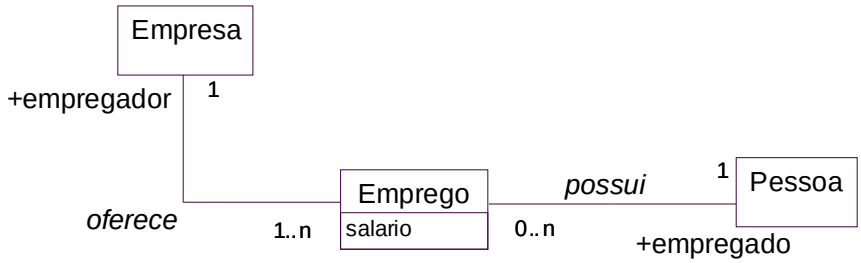


Figura 4.9 – Alternativa ao Tipo Associativo

4.4 – Herança

Herança é um mecanismo que permite que características comuns a diversas classes sejam colocadas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas (as subclasses). Cada subclasse apresenta as características (estrutura e métodos) da superclasse e acrescenta a elas novas características ou comportamento. Dizemos que uma subclasse **herda** todas as propriedades da superclasse e acrescenta suas características próprias e exclusivas. As propriedades da superclasse não precisam ser repetidas em cada subclasse.

Por exemplo, quando pensamos em um forno, logo nos vêm à mente sua utilidade geral, que é cozinhar alimentos. Várias subclasses de Forno adicionam características próprias aos diversos tipos de forno, como microondas, à gás, elétrico e à lenha, conforme ilustrado na Figura 4.10, já utilizando a notação da UML, que é de um triângulo não preenchido ligando a superclasse às suas subclasses. Podemos criar diversos níveis de hierarquia de herança. Por exemplo, fornos à gás podem ser de diversos tipos, como com acendimento elétrico, com fósforo e para camping.

Em termos de Modelo Conceitual, chamamos as superclasses de **tipos** e as subclasses de **subtipos**. É importante identificar a herança durante a análise do sistema, embora, durante o projeto, seja possível identificar outros casos que podem ter passado despercebidos durante a análise.

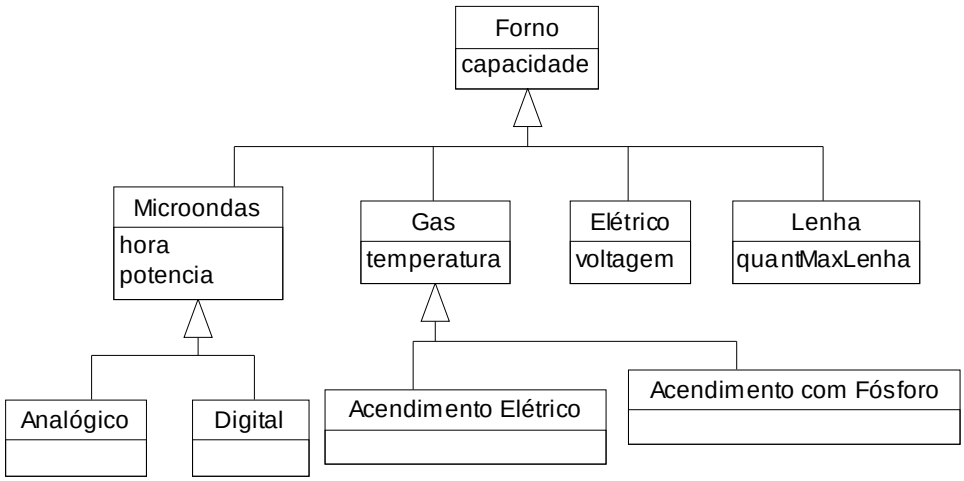


Figura 4.10 – Exemplo de Herança

Para ter certeza de que um conceito ou classe pode ser representado usando herança, deve-se ter em mente duas regras importantes: a regra "é-um" e a regra dos 100%.

A regra "é-um" estabelece que todos os membros do conjunto de um subtipo devem ser membros do conjunto do supertipo, ou seja, o subtipo é um supertipo. Por exemplo, podemos dizer que um microondas digital é um forno, e por isso possui todos os atributos e comportamentos esperados de um forno.

A regra dos 100% estabelece que 100% da definição do supertipo dever ser aplicado ao subtipo, ou seja, o subtipo deve estar conforme com 100% dos elementos do supertipo, que são seus atributos e associações. Assim, se a superclasse possui um dado atributo, é necessário garantir que todas as subclasses também o possuam, ou seja, ele deve fazer sentido para elas e deve ser importante conhecer seu conteúdo. O mesmo vale para as associações da superclasse, que devem ser aplicáveis às subclasses. Por exemplo, se incluirmos a classe Cozinheiro no modelo da Figura 4.10 e associarmos essa classe à classe forno, significando que o Cozinheiro utiliza o Forno, isso deve ser verdadeiro para todas as subclasses, ou seja, todos os tipos de forno devem poder ser utilizáveis por um cozinheiro.

4.5 – Agregação

Agregação é um mecanismo pelo qual um objeto inclui atributos e comportamento de outros objetos a ele agregados, indicando a existência de um *todo*, composto por *partes*. Um exemplo de agregação é um carro: consiste em 4 rodas, um motor, chassis, caixa de câmbio, e assim por diante.Quando uma agregação é usada, ela geralmente descreve vários níveis de abstração. As perguntas básicas para identificar a agregação seriam: o objeto "é composto de" outros objetos? O objeto "consiste em" outros objetos? O objeto "contém" outros objetos? O objeto "é parte de" outro objeto? A Figura 4.11 ilustra a agregação no caso da cozinha: o fogão é o todo, e poderia ser composto de partes como os queimadores

(de um a seis), o forno e a estufa. A representação em UML é de um losango ligando a classe que representa o todo às classes que representam as partes.

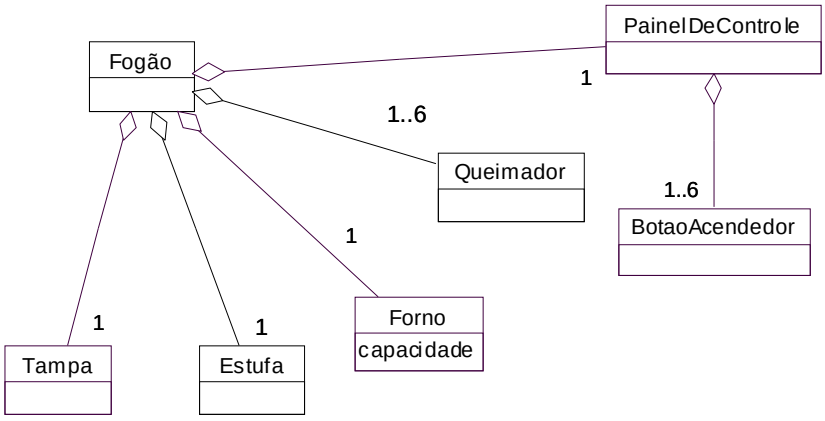


Figura 4.11 – Exemplo de Agregação

Há dois tipos possíveis de agregação: a agregação composta (composição) e a agregação compartilhada. A **agregação composta** ou composição ocorre quando a multiplicidade na extremidade do composto pode ser no máximo 1. A notação para isso em UML é um losango negro. Por exemplo, na Figura 4.12 um automóvel é composto de quatro rodas. O losango negro significa que a roda pertence a no máximo um automóvel.

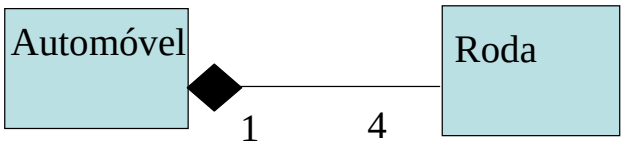


Figura 4.12 – Exemplo de Agregação composta

Já a agregação compartilhada, denotada em UML por um losango vazio, denota que a multiplicidade na extremidade do composto pode ser maior do que um. Por exemplo, a Figura 4.13 ilustra o caso de pacotes na UML, que são estruturas que agregam vários outros elementos da UML. No entanto, um dado elemento que pertence a um pacote específico pode simultaneamente pertencer a outros pacotes diferentes. Este é um exemplo de agregação lógica. Já na Figura 4.14 mostra-se um exemplo de agregação física, entre CD e Música.

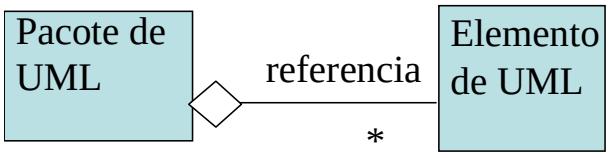


Figura 4.13 – Exemplo de Agregação compartilhada

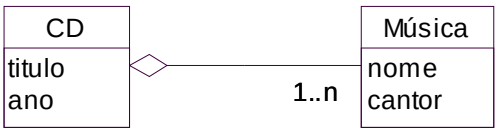


Figura 4.14 – Exemplo de Agregação compartilhada

4.6 – Um exemplo

Para melhor ilustrar os modelos conceituais, a Figura 4.15 contém um Modelo Conceitual de uma biblioteca (versão simplificada).

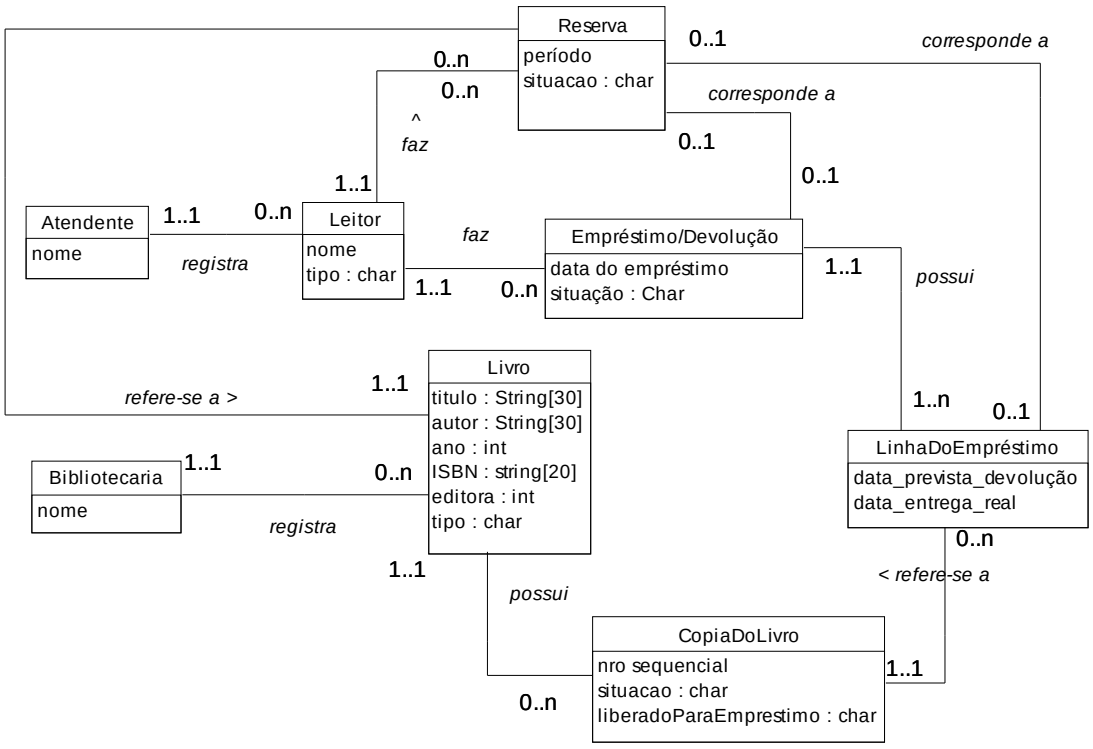


Figura 4.15 – Modelo Conceitual para Biblioteca

4.7 – Exercícios propostos

4.7.1. Com base nos requisitos do sistema Passe-Livre, fornecidos no Apêndice A, e nos casos de uso completo abstratos resultantes dos exercícios 3.9.5 e 3.9.6, elaborar o Modelo Conceitual para o sistema, seguindo as diretrizes fornecidas nas seções anteriores deste Capítulo.

4.7.2. Considerando o resultado exercício 4.7.1, mostre como chegou à lista de candidatos a conceitos e porque descartou ou aceitou cada um deles.

4.7.3. Considerando o resultado do exercício 4.7.1, explique pelo menos 3 associações e suas multiplicidades.

4.7.4. Refine o modelo do exercício 4.7.1, incluindo herança e agregação. Justifique.

4.8 – Exercícios complementares

4.8.1. Considere uma loja que aluga fitas de vídeos e DVDs para clientes cadastrados. Elabore o modelo conceitual para essa loja, considerando o documento de requisitos produzido no exercício 3.10.1 da seção 3.10.

4.8.2. Com base no documento de requisitos de um sistema para automatização de uma biblioteca universitária, produzido no exercício 3.10.2 da seção 3.10, elabore o modelo conceitual para esse sistema.

4.8.3. Refine o modelo conceitual elaborado no exercício 4.8.1, considerando associações de herança e agregação.

4.8.4. Estenda o modelo conceitual do exercício 4.8.3 para incluir venda de produtos na locadora (por exemplo, revistas, DVDs, CDs, etc.)

4.8.5. Refine o modelo conceitual elaborado no exercício 4.8.2, considerando associações de herança e agregação.

4.8.6. Com base no modelo conceitual elaborado no exercício 4.8.4, reutilize esse modelo para produzir outro modelo, agora para um sistema de locação de carros, no qual podem ser alugados carros, ao mesmo tempo em que se adquirem itens como seguro, gasolina, etc.

4.8.7. Considere a descrição do exercício 3.10.9, da seção 3.10, de um sistema de suporte ao reparo de buracos de uma cidade. Elabore o modelo conceitual para tal sistema.

4.8.8. Reutilize parte do modelo produzido no exercício 4.8.7 para produzir um modelo conceitual para um sistema de conserto de carros. Sempre que possível, utilize conceitos de herança e agregação, representando-os em UML.

Capítulo 5 – Disciplina de Requisitos – Cenários, Contratos de Operações e Diagramas de Estados

5.1 – Cenários ou Diagramas de Seqüência do Sistema

5.1.1 – Visão Geral

Para finalizar a disciplina de requisitos do PU, é recomendável que se tenha uma noção mais concreta do comportamento esperado do sistema diante dos vários eventos que ocorrem em cada caso de uso. Uma forma prática de conseguir isso é construindo cenários ou diagramas de seqüência do sistema (DSS). Um DSS mostra, em alto nível, os principais eventos que fazem parte de um caso de uso. Cada um desses eventos dispara uma operação do sistema para tratá-lo. A idéia é identificar quais são esses eventos e quem é o ator responsável por iniciá-lo.

Os DSS são também conhecidos por Cenários, por mostrarem um cenário global do funcionamento do sistema. Por meio deles é possível saber quais são os grandes **eventos** que ocorrem, ou ações principais de um ator, sendo que o sistema é considerado como uma caixa-preta, que recebe o evento de um ator, processa-o de alguma forma que não nos interessa detalhar no momento e, opcionalmente, retorna uma resposta ao ator.

Por exemplo, um DSS para o caso de uso “Emprestar Livro”, visto no Capítulo 3, mostraria basicamente três eventos: iniciar o empréstimo, quando o atendente informa ao sistema a identificação do leitor para checar se é um leitor apto a emprestar; emprestar livro(s), em que são informados ao sistema os dados sobre os livros a emprestar; e encerrar o empréstimo, em que registra-se efetivamente o empréstimo para o leitor, mudando a situação dos livros para “em circulação”. Dizemos que “iniciar o empréstimo” é um evento de entrada que ocorre e dispara a execução de uma **operação** para tratá-lo. Tanto o evento quanto a operação correspondente devem ter o mesmo nome, para facilitar a abordagem.

O evento “iniciar o empréstimo” é disparado pelo atendente, que fornece informações necessárias para o correto tratamento, similarmente a parâmetros passados para uma função. Nesse caso, o código de identificação do leitor é suficiente para que o sistema faça a verificação da situação do leitor no sistema, isto é, verifique se ele está apto a emprestar (não possui livros em atraso e não atingiu o máximo de livros permitidos). O evento “emprestar livro”, também disparado pelo atendente, deve fornecer a identificação do livro, para que seja possível para o sistema verificar a disponibilidade do livro (por exemplo, deve-se verificar se não é um livro retido somente para consulta, se o livro não está reservado para outro leitor, etc.). O evento “encerrar o empréstimo” não precisa de informações adicionais, pois terá como responsabilidade criar o novo empréstimo, associá-lo ao leitor e atualizar a situação de todos os livros emprestados, o que não requer parâmetros.

Note que o cenário apresentado é apenas uma possível solução para o caso de uso “Emprestar Livro”, ou seja, poderia haver outras formas de organizar os eventos. No

entanto, a seqüência sugerida está fortemente baseada na seqüência típica de eventos apresentada na Figura 3.4. Um bom exercício é analisar tal figura e sugerir outras seqüências de eventos possíveis, comentando qual seria o comportamento esperado de cada evento nessas outras sugestões. Por exemplo, se a última operação sugerida acima, "encerrar o empréstimo", fosse eliminada, os livros poderiam ter sua situação alterada no evento "emprestar livro(s)" o novo empréstimo poderia ser criado e associado ao leitor no evento "iniciar empréstimo".

Há uma grande influência da seqüência típica de eventos do caso de uso em relação aos eventos definidos no DSS. Por isso, é importante que a seqüência típica reflita fielmente o que ocorre no sistema real. Por exemplo, se no caso de uso "Emprestar Livro" a seqüência fosse outra, digamos, se os livros fossem primeiramente checados um a um, para depois pedir a identificação do leitor, mudariam as responsabilidades atribuídas a cada um dos eventos. Uma dessas mudanças seria em relação a verificar se o livro está reservado ou não. Sem saber a identificação do leitor a priori, essa verificação teria que ser deixada para depois.

O Processo Unificado recomenda que se construa um DSS para cada caso de uso relevante do sistema. A classificação de um caso de uso como relevante ou não é subjetiva, mas pode-se dizer que os casos de uso alocados aos primeiros ciclos de desenvolvimento são mais relevantes e, portanto, devem ter um DSS correspondente, que define quais são as operações a serem implementadas no sistema.

5.1.2 – Notação UML

A Figura 5.1 mostra a notação UML para os DSS. Os atores são, em geral, colocados na parte superior, mais à esquerda, enquanto o Sistema é posicionado na parte superior direita. Uma linha vertical representando o tempo é desenhada para cada ator e para o sistema (considera-se que o tempo corre de cima para baixo). Os eventos são representados por setas direcionadas partindo dos atores e indo para o sistema, ou de atores para atores. No caso de ser interessante representar a resposta do sistema, pode-se incluir também setas tracejadas partindo do sistema e indo para o ator desejado. Uma notação especial para representar repetição do mesmo evento é disponibilizada: um caixa envolvendo os eventos que se repetem, na qual pode-se colocar um texto especificando o critério de repetição ou de parada. No caso da Figura 5.1, o evento iniciarEmpréstimo ocorre uma vez, o evento emprestarLivro ocorre tantas vezes quantos forem os livros a serem emprestados e o evento encerrarEmpréstimo ocorre uma única vez.

Eventos envolvendo dois atores, como por exemplo o evento entregarCarteiraIdentificação da Figura 5.1, são em geral desconsiderados durante as fases posteriores de projeto, pois ficam fora dos limites do sistema. Eles são importantes para entendimento do funcionamento global do caso de uso, mas não se tornarão operações efetivamente implementadas no software. Assim, os eventos numerados como 1 e 3 na Figura 5.1 não precisam ser modelados, ou seja, o DSS poderia ser simplesmente como o da Figura 5.2.

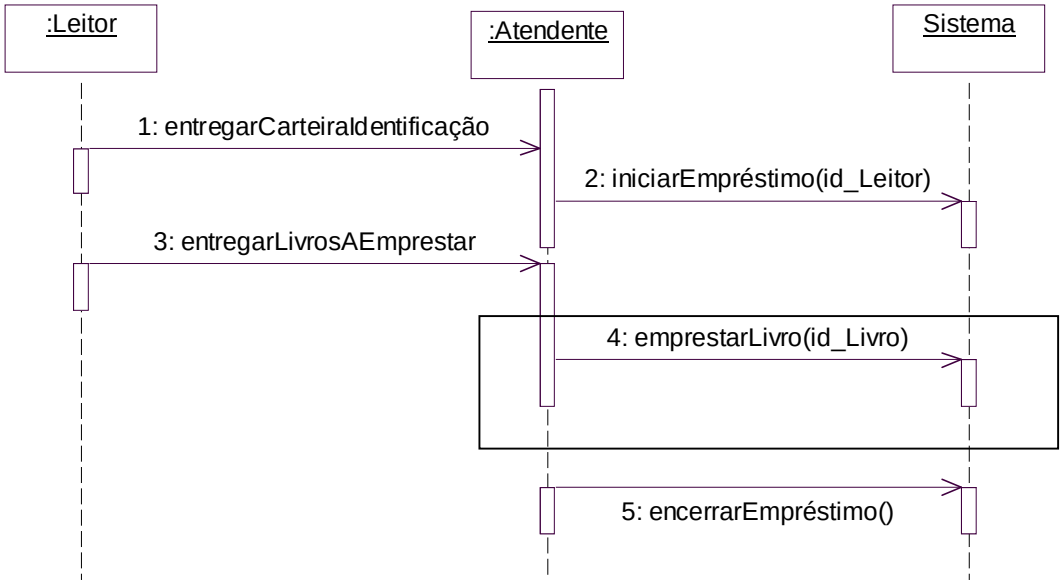


Figura 5.1 – Diagrama de Seqüência do Sistema para Emprestar Livro

Note-se que esta é uma decisão que implica que o Atendente é quem interage com o sistema. Esta é, na realidade, uma decisão arbitrária e o contrário também seria válido, mas optamos por essa solução por ser mais realística, neste caso.

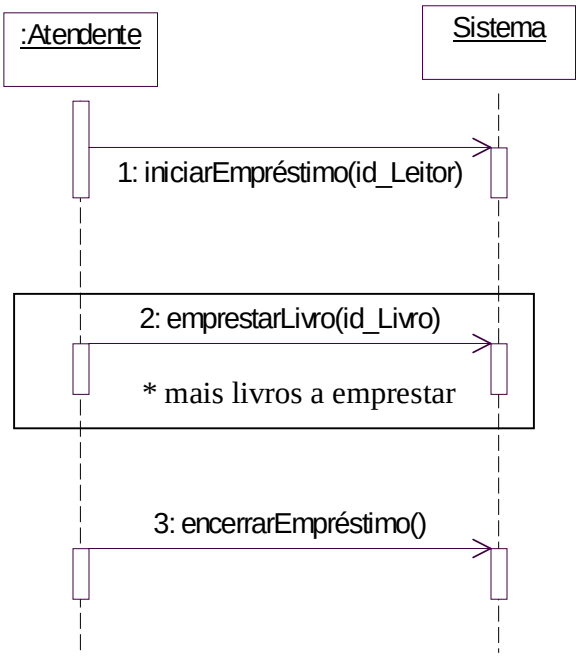


Figura 5.2 – Diagrama de Seqüência do Sistema para Emprestar Livros, versão 2

Um evento pode ser de dois tipos: de entrada ou de saída. Um **evento de entrada** é aquele no qual o ator dispara uma certa operação do sistema. Um **evento de saída** é simplesmente uma resposta do sistema à uma operação (executada após a ocorrência do evento de entrada). Na UML, os eventos de saída são denotados por meio de uma seta tracejada, partindo do Sistema em direção a um ator, como na Figura 5.3, em que o sistema informa ao atendente o nome do leitor e sua situação, em resposta à operação iniciarEmpréstimo, e a data de devolução do livro, em resposta à operação emprestarLivro. Os eventos de saída são opcionais e devem ser usados com cautela, para não tornar o DSS complexo, sem necessidade.

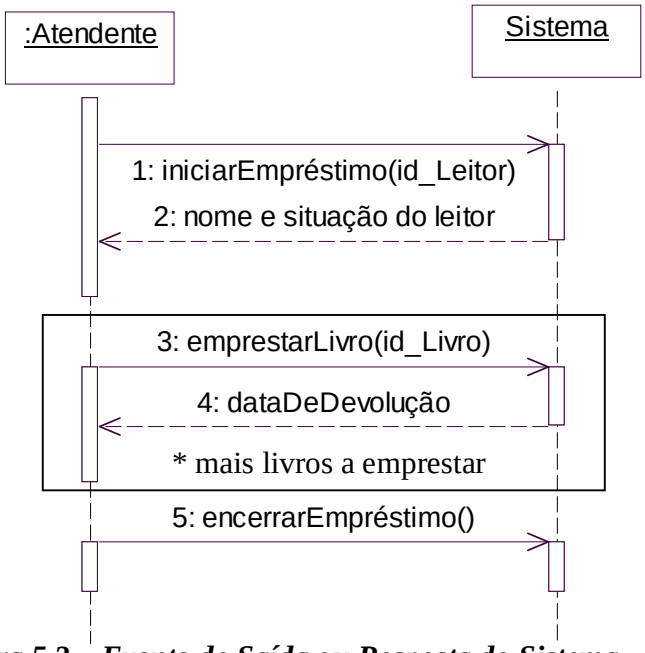


Figura 5.3 – Evento de Saída ou Resposta do Sistema

5.2 – Contrato da Operação

5.2.1 – Definição

Na Seção 5.1 vimos que um caso de uso pode ter um DSS a ele associado, o qual contém os eventos relacionados à interação entre os atores e o sistema. Vimos também que cada evento possui uma **operação** para tratá-lo e que devemos estabelecer o que cada operação deve fazer. É importante que as tarefas atribuídas às operações sejam bem documentadas, para evitar redundâncias e inconsistências. Para isso surgem os **contratos** das operações.

Um contrato especifica o comportamento esperado para cada operação correspondente a um evento do sistema. Dessa forma, não se corre o risco de desentendimentos na disciplina de projeto do sistema, quando as operações serão projetadas detalhadamente e posteriormente implementadas. No entanto, não se espera que elas forneçam um algoritmo detalhado do que deve ocorrer, mas sim apenas suas obrigações em termos do que muda no sistema após sua invocação.

Características típicas de um contrato incluem: nome da operação, parâmetros de entrada, referências cruzadas, pré-condições e pós-condições. As referências cruzadas são úteis para saber a quais casos de uso a operação se refere. As pré-condições devem fornecer as condições necessárias para que a operação seja executada, ou o estado do sistema antes da invocação da operação. As condições que estiverem listadas como pré-condições não serão verificadas pela operação, ou seja, assume-se que elas são verdadeiras ao invocar a operação. Assim, caso a operação seja invocada sem possuir as pré-condições, não se garante que o comportamento será o esperado. Por exemplo, se for estabelecido como pré-condição que o código do leitor existe no sistema, e a operação buscar(codigoLeitor) for invocada com um número de leitor inválido, o sistema poderá comportar-se de forma errônea, por exemplo, retornar um objeto nulo sem emitir mensagem de erro alguma, visto que foi assumido que o código de leitor fornecido seria válido.

As pós-condições procuram refletir o estado do sistema após a invocação da operação, mostrando o que mudou como conseqüência da sua execução. É importante refletir em termos dos conceitos identificados no Modelo Conceitual do Sistema (Capítulo 4), descrevendo, para cada possível objeto do sistema, o que muda quando a operação é invocada.

Os tipos de mudança que devem ser captados pelas pós-condições são:

- AV - Alteração no valor de atributos de um objeto: a operação atribui valores a atributos do objeto
- CO - Criação de novos objetos: durante a operação um ou mais objetos são criados no sistema. Deve-se observar que a simples recuperação de objetos já existentes, de uma base de dados por exemplo, não consiste em criação de novos objetos.
- CA - Criação de associação: a operação estabelece uma associação de um objeto com outro. Essa associação não existia antes da operação ter sido invocada.
- RA - Remoção de associação: a operação faz com que deixe de existir uma associação que existia antes de sua invocação.
- RO - Remoção de objeto: durante a operação o objeto é removido, juntamente com suas associações para outros objetos.

Além de observar o modelo conceitual, deve-se observar a seqüência de operações do DSS, para ter uma melhor idéia do contexto em que a operação está inserida (que operações já foram invocadas anteriormente) e o contexto resultante (o que se espera deixar para as operações seguintes). Com isso, o contrato pode ajudar a antecipar os objetos que aparecerão na etapa de projeto do sistema.

Por exemplo, considere a operação “encerrarEmpréstimo”. Analisando o DSS do caso de uso “Emprestar Livros” percebe-se que no momento da invocação de “encerrarEmpréstimo” já terá sido executada a operação “iniciarEmpréstimo”, a qual identificou e validou o leitor e, além disso, já terá sido executada a operação “emprestarLivro”, que se encarregou de verificar se cada um dos livros estava apto a ser emprestado. Portanto, conclui-se que a operação “encerrarEmpréstimo” deve acrescentar

um novo empréstimo, relacioná-lo com o leitor identificado e mudar a situação dos livros correspondentes. O modelo conceitual do sistema de biblioteca (Figura 4.15) é utilizado para identificar os conceitos que terão seu estado modificado. Portanto, as pós-condições da operação "encerrarEmpréstimo" são: acréscimo de um novo empréstimo, associação do novo empréstimo ao leitor identificado e mudança da situação dos livros para "emprestado". As pré-condições são: um leitor apto a emprestar livros já foi identificado e pelo menos um livro já foi identificado e está disponível para ser emprestado.

5.2.2 – Notação

A UML não fornece uma notação específica para os contratos. Sugere-se aqui um gabarito, baseado nos esquemas propostos por Coleman et al. [1994] e adaptados por Larman [2004], conforme a Figura 5.4. Outros exemplos de contratos podem ser encontrados no Capítulo 11.

Recomenda-se que um contrato seja feito para cada operação considerada relevante para o sistema. Operações sem pré e pós requisitos ou com pré e pós requisitos muito simples podem não ser tão relevantes e não precisam do contrato.

Uma operação pode aparecer em vários DSSs, já que muitos casos de uso possuem eventos similares. Por exemplo, tanto ao emprestar quanto ao devolver um livro, em algum momento o Atendente informará o ISBN do livro e o sistema deverá fazer uma busca. Portanto uma operação "buscarLivro(isbn)" seria comum aos DSSs dos dois casos de uso, tendo somente um contrato para descrevê-la, mas com referências cruzadas aos dois casos de uso.

Operação: encerrarEmpréstimo() ***Referências Cruzadas***: Caso de uso: "Emprestar Livro" ***Pré-Condições***: Um leitor apto a emprestar livros já foi identificado; pelo menos um livro já foi identificado e está disponível para ser emprestado. ***Pós-Condições***: um novo empréstimo foi registrado (CO); o novo empréstimo foi relacionado ao leitor já identificado na operação "iniciar o empréstimo" (CA); a situação dos livros emprestados foi alterada para "emprestado" (AV).

Figura 5.4 – Contrato para a operação Encerrar o empréstimo

5.3 – Diagramas de Estados

Nesta seção introduz-se um diagrama bastante útil para entender a dinâmica de um objeto: o diagrama de estados. Ele mostra todos os estados em que um determinado elemento pode estar e os eventos que podem causar a mudança de um estado para outro.

Os conceitos básicos do diagrama de estados são: eventos, estados, e transições. Usaremos a notação da UML para diagrama de estados, exemplificada na Figura 5.5, para denotar os possíveis estados de um objeto Telefone.

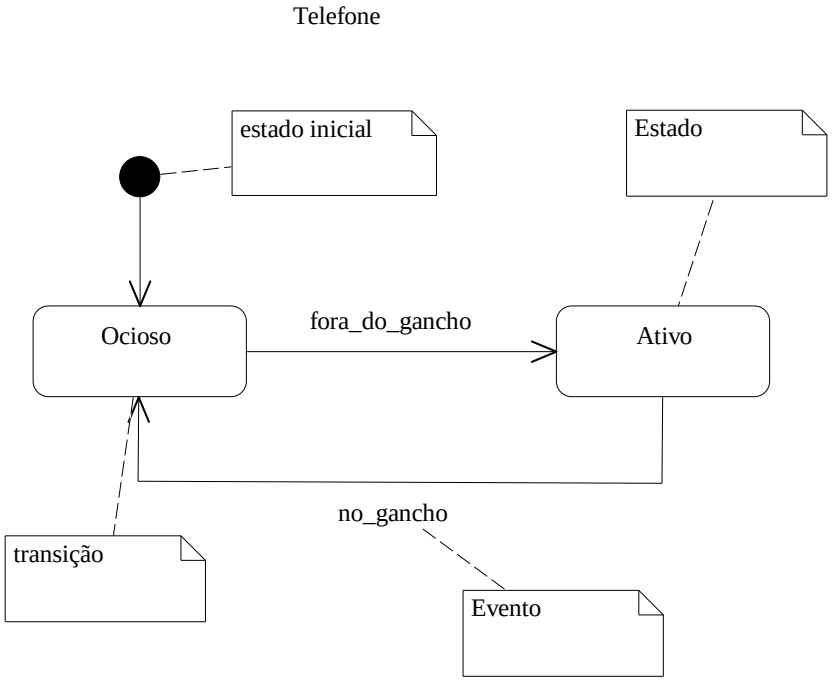


Figura 5.5 – Diagrama de Estados para um Telefone

Um **evento** é uma ocorrência significativa ou digna de nota, por exemplo, um telefone é tirado do gancho. O evento normalmente é atômico, isto é, não consome tempo.

Um **estado** é a condição de um objeto em um certo momento no tempo, o tempo entre dois eventos. Por exemplo, o telefone fica no estado “ocioso” depois que foi colocado no gancho e até ser retirado do gancho novamente.

Uma **transição** é um relacionamento entre dois estados, indicando que quando um evento ocorre um objeto passa do estado anterior para o subseqüente. Por exemplo, quando ocorre o evento “fora do gancho” o telefone passa do estado “ocioso” para o estado “ativo”, portanto há uma transição de estados. Não é necessário mostrar todos os eventos possíveis. Se ocorrer um evento não mostrado, ele é ignorado. Isso permite criar diagramas com diferentes níveis de abstração.

Se um objeto responde a um evento sempre da mesma forma, então ele é considerado **independente de estado** (ou não modal) com relação àquele evento. Se para todos os eventos de interesse um tipo sempre reage da mesma maneira, então ele é um tipo independente de estado. Ao contrário, tipos **dependentes de estado** reagem de maneira diferente a eventos, dependendo de seu estado. Tipos e classes comumente dependentes de

estado são: casos de uso, sistemas, janelas (por exemplo, copiar-colar só é uma operação válida se existir algo na área de transferência), transações e dispositivos físicos, entre outros.

Existem alguns recursos adicionais que podem ser utilizados nos diagramas de classe, que são: ação da transição, condições guardiãs de transição e estados aninhados. Quando ocorre uma transição de estados, pode ser que o sistema tenha que executar uma ação antes de passar para o estado seguinte, denominada **ação de transição**. Isso é denotado colocando o nome da ação logo após o nome do evento, como ilustrado na Figura 5.6, em que o evento "fora do gancho" dispara a ação "tocar sinal de linha". Se houver mais de uma ação, elas são separadas por ponto e vírgula.

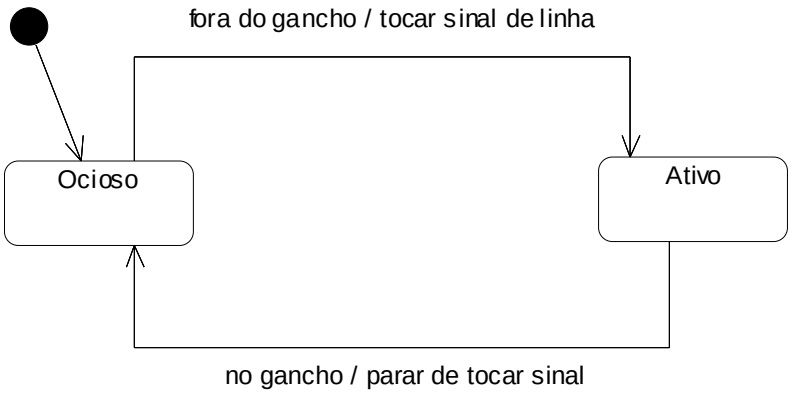


Figura 5.6 – Ação de Transição no Diagrama de Estados

Condições guardiãs podem ser especificadas para restringir o disparo da transição, ou seja, mesmo que o evento ocorra, caso a condição não seja satisfeita a transição não ocorre. Isso é ilustrado na Figura 5.7, em que o sinal de linha só é tocado se o assinante da linha telefônica estiver com sua situação regularizada.

Finalmente, outro recurso importante que pode ser utilizado em diagramas de estados é o de **estados aninhados**. Vários estados podem ser agrupados em um estado aninhado, de forma que a transição que chega ao estado aninhado entra inicialmente em um estado *default*, e se há uma transição saindo do estado aninhado ela é disparada para quaisquer estados contidos no estado aninhado. Por exemplo, na Figura 5.8, depois que o telefone entra no estado "Ativo", a qualquer momento o telefone pode ser colocado no gancho, e esse evento marcará a transição para o estado "Ocioso".

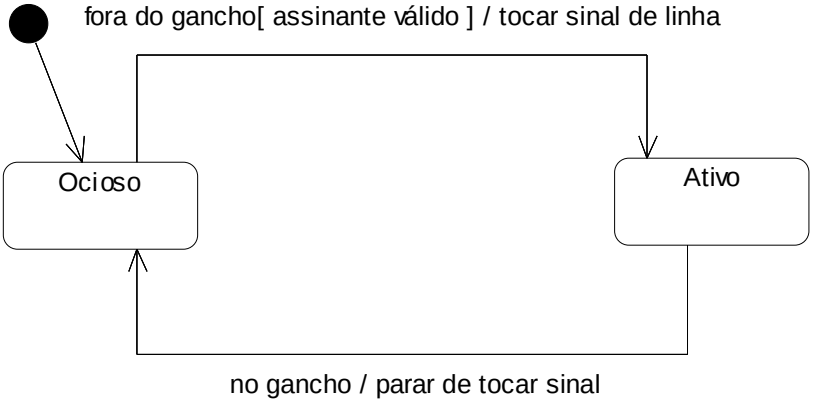


Figura 5.7 – Condição guardiã no Diagrama de Estados

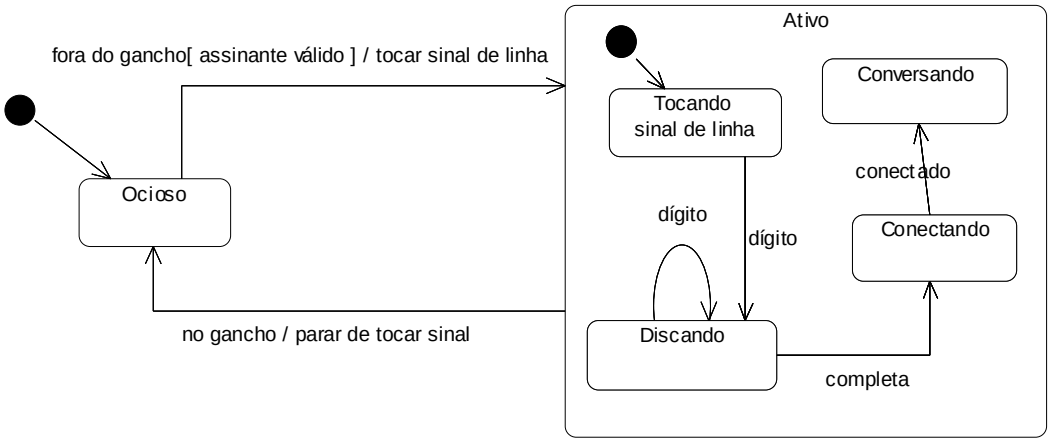


Figura 5.8 – Estados aninhados em um Diagrama de Estados

No Processo Unificado, um diagrama de estados pode ser aplicado a diversos elementos dos modelos OO, entre os quais: conceitos (do modelo conceitual), classes de software, casos de uso e o próprio sistema. Nos exemplos das Figuras 5.5 a 5.8, o diagrama de estados refere-se a um objeto, ou seja, ele mostra os possíveis estados de um telefone diante dos eventos do sistema. No entanto, ele poderia ser aplicado a outros elementos, como por exemplo a casos de uso. Considere o caso de uso Emprestar Livro do sistema de Biblioteca. O caso de uso inicia quando o leitor chega ao balcão com os livros a emprestar. Portanto, pode haver um estado inicial em que a Atendente está “aguardando novos leitores chegarem”. A partir do momento que o Leitor se identifica, o caso de uso passa para o estado “identificando o leitor”. Enquanto novos livros são identificados, o estado passa para “registrando livros”. Finalmente, quando terminam os livros, o estado do caso de uso passa para o estado final, que nesse caso é o retorno ao estado inicial, ou seja, “aguardando novos leitores”.

5.4 – Exercícios propostos

5.4.1. Com base nos requisitos do sistema Passe-Livre, fornecidos no Apêndice A, nos resultados dos exercícios sobre casos de uso (seção 3.9) e no Modelo Conceitual resultante do exercício 4.7.1, elaborar o DSS para os casos de uso "Entrar na Autopista" e "Comprar Gizmo"

5.4.2. Com base nos DSSs obtidos no exercício 5.4.1, elaborar os contratos para pelo menos 3 operações identificadas. Para cada pós-condição, identifique seu tipo.

5.4.3. Elabore o diagrama de estados para os seguintes objetos do Sistema Passe Livre: veículo, gizmo, registro de uso, cancela e semáforo.

5.4.4. Elabore o diagrama de estados para o caso de uso "Comprar Gizmo" do Sistema Passe Livre. Baseie-se no cenário de sucesso principal e nos fluxos alternativos do caso de uso abstrato completo.

5.5 – Exercícios complementares

5.5.1. Elabore o DSS e os respectivos contratos para pelo menos dois dos casos de uso para uma loja que aluga fitas de vídeos e DVDs para clientes cadastrados. Use o caso de uso no formato expandido elaborado no exercício 3.10.1 para definir os eventos e o Modelo Conceitual elaborado no exercício 4.8.1 para ajudar a definir as pós-condições das operações.

5.5.2. Faça modificações na ordem dos eventos do caso de uso do exercício 5.5.1 e elabore o novo DSS e os novos contratos. Comente as diferenças e discuta qual é a melhor solução.

5.5.3. Com base no caso de uso Devolver Livro, elaborado no exercício 3.10.4, elabore o DSS e os respectivos contratos das operações mais importantes. Use o Modelo Conceitual elaborado no exercício 4.8.2 para ajudar a definir as pós-condições das operações.

5.5.4. O que mudaria se alguns requisitos adicionais fossem incluídos no caso de uso Emprestar Livro, por exemplo, se uma multa fosse cobrada por atraso? E se o leitor fosse expulso ou suspenso por atrasar a devolução do livro? E se o leitor que reservou o livro tivesse que ser notificado sobre a devolução?

5.5.5. Com base no modelo conceitual elaborado no exercício 4.8.4, elabore o DSS para um dos casos de uso (escolha o mais significativo) e os respectivos contratos das operações mais importantes do sistema.

5.5.6. Elabore o diagrama de estados para o Livro da Biblioteca.

5.5.7. Considere o enunciado do exercício 3.10.9, sobre um sistema de reparo de buracos de uma cidade. Elabore o diagrama de estados para os objetos ordem de serviço e buraco.

5.5.8. Qual é a importância dos contratos relacionados às operações?

5.5.9. Você acha que o DSS enquadra-se na disciplina de requisitos ou já avança para projeto? Discuta.

5.5.10. Quanto ao diagrama de estados, você considera que faz parte da disciplina de requisitos ou possui elementos de projeto? Justifique.

Capítulo 6 – Disciplina de Projeto – Diagramas de Colaboração

Nos Capítulos 3 a 5 foi abordada a análise orientada a objetos, sem aprofundar os detalhes de como os modelos produzidos seriamv projetados de forma a serem implementáveis em uma linguagem de programação orientada a objetos. Em outras palavras, dados os requisitos do software a ser desenvolvido, os requisitos foram analisados detalhadamente e foram produzidos modelos que representam o quê o software deve fazer. Portanto, os artefatos produzidos durante a análise OO representam o **quê** o sistema faz, sem detalhes que comprometam a solução a uma tecnologia específica. A partir deste capítulo, enfocaremos os artefatos de projeto, que contém detalhes sobre **como** o sistema poderá ser implementado utilizando um computador. Para conseguir isso, será necessário detalhar as informações sobre as classes que comporão o sistema, incluindo o comportamento esperado de cada objeto e a colaboração entre os objetos.

No Processo Unificado, o projeto é conduzido utilizando basicamente dois tipos principais de diagramas UML: o Diagrama de Classes e os Diagramas de Colaboração. Os diagramas de classes serão vistos no Capítulo 7, pois são construídos a partir de informações presentes nos diagramas de colaboração, vistos neste Capítulo.

6.1 – Visão Geral da notação

A UML possui dois diagramas específicos para mostrar a interação entre objetos: o diagrama de colaboração e o diagrama de seqüência. O **Diagrama de Seqüência** é similar ao visto no Capítulo 5, com a diferença que, ao invés de representar os atores e o Sistema, representam-se dois ou mais objetos trocando mensagens. A Figura 6.1 ilustra como um diagrama de seqüência pode ser utilizado para mostrar a interação entre os objetos por meio de mensagens.

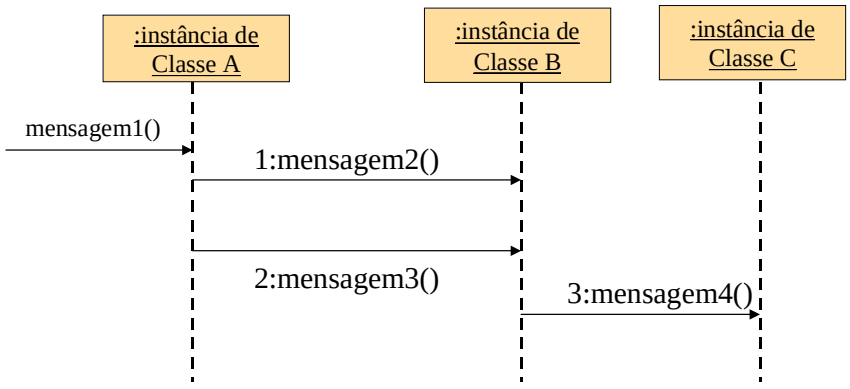


Figura 6.1 – Notação UML para Diagrama de interação: Diagrama de Seqüência

Note que embora tenha sido utilizada a mesma notação para representar os Diagramas de Seqüência do Sistema (Seção 5.1), lá o processo encontrava-se na fase de requisitos e mostravam-se apenas os atores e o sistema, que era visto como uma caixa preta. Tratava-se, portanto, de uma **visão externa** do sistema. Já na fase de projeto, a mesma notação é utilizada com um enfoque diferente: deseja-se detalhar a interação entre os objetos, portanto uma **visão interna** do sistema é desejada neste momento. Para tanto, utilizam-se os diagramas de interação entre objetos.

Nas seções seguintes será mostrada a notação específica para o **Diagrama de Colaboração**. O mesmo diagrama de interação entre objetos, desenhado na forma de um Diagrama de Seqüência na Figura 6.1, pode ser desenhado utilizando a notação do Diagrama de Colaboração, como mostrado na Figura 6.2. Os diagramas de colaboração têm melhor capacidade de expressar informações contextuais, conforme será visto nas seções seguintes, e podem ser mais econômicos em termos de espaço.

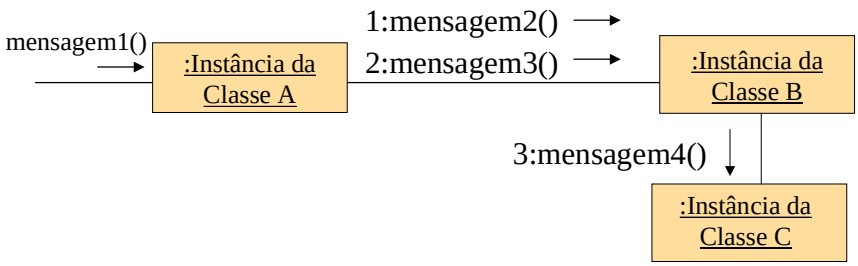


Figura 6.2 – Notação UML para Diagrama de interação: Diagrama de Colaboração

6.1.1 – Notação UML para mensagens entre dois objetos

Conforme mencionado no Capítulo 1, no paradigma OO deve-se projetar o sistema em termos de objetos e da comunicação entre eles. A única forma de um objeto se comunicar com outro é enviando uma mensagem para esse outro objeto, que poderá responder a essa mensagem. Vimos também que um objeto, ao receber uma mensagem, executa um método para atender a essa mensagem. A notação da UML para comunicação entre objetos é ilustrada na Figura 6.3. Alguns detalhes dessa notação serão explicados nas seções seguintes.

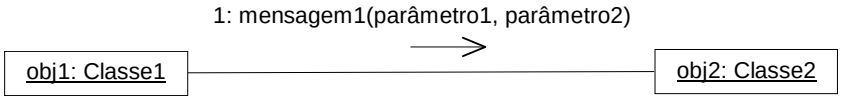


Figura 6.3 – Mensagem entre dois objetos na UML

O objeto chamado **obj1** é uma instância da classe **Classe1**. O nome do objeto poderia ter sido omitido, portanto poderia ter sido colocado dentro do retângulo apenas **:Classe1** (note que o sinal de dois pontos é necessário para indicar que trata-se de um objeto da classe, neste caso um objeto sem nome). Da mesma forma, **obj2** é uma instância da classe **Classe2**. O objeto **obj1** está enviando uma mensagem, em particular a **mensagem1**, ao objeto **obj2**.

Alguns parâmetros estão sendo passados ao objeto **obj2** (**parâmetro1** e **parâmetro2**), que são colocados entre parênteses, separados por vírgula.

Para poder responder a **mensagem1**, a **Classe2** deve possuir um método **mensagem1** em sua interface, ou seja, o método deve estar disponível e ser visível a **Classe1**. Visibilidade entre classes será abordada no Capítulo 7.

A Figura 6.4 mostra um exemplo mais concreto, em que o objeto **linhaEmpto** (da classe **LinhaDoEmprestimo**, ver Figura 4.9) envia uma mensagem ao objeto **copiaLivro** (da classe **CopiaDoLivro**) para solicitar que a situação da cópia seja modificada para "disponível", após a devolução do livro. A classe **CopiaDoLivro** deve possuir um método para alterar o valor do atributo situação (o nome desse método é **mudarSituacao** e ele possui um parâmetro do tipo String)

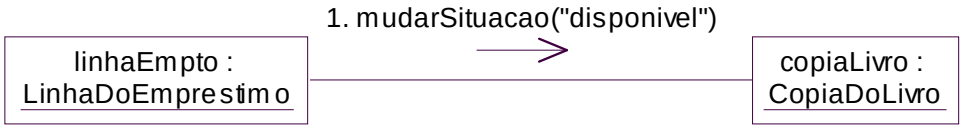


Figura 6.4 – Exemplo de mensagem entre dois objetos

6.1.2 – Ordem das mensagens

Em geral, um Diagrama de Colaboração mostra a comunicação entre vários objetos para realizar um certo comportamento desejado. Por exemplo, se um objeto não contém conhecimento suficiente para responder a uma determinada mensagem, ele pode recorrer a outros objetos que conheça, enviando-lhes mensagens para obter o conhecimento desejado. Nesse sentido, a ordem em que as mensagens são enviadas é freqüentemente muito importante. Por isso, as mensagens são numeradas (ver Figuras 6.1 a 6.4). A Figura 6.5 mostra um outro exemplo em que a ordem das mensagens é importante. Por exemplo, a **linhaDoEmpréstimo** só pode ser criada (mensagem 3) após a obtenção do tipo do leitor (mensagem 2), pois o método criar passa como parâmetro esse tipo de leitor. Outros conceitos e notações incluídos nesta figura serão explicados nas seções seguintes.

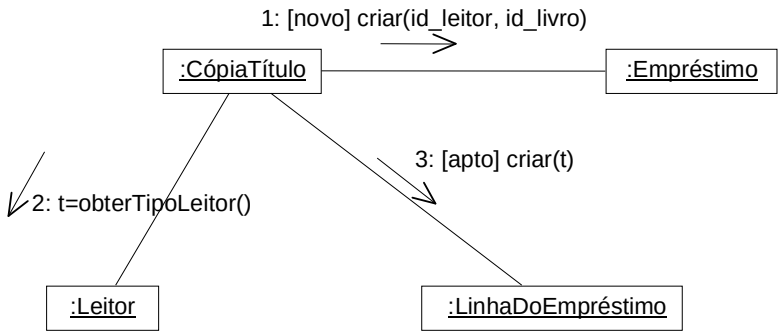


Figura 6.5 – Ordem das mensagens, condicionais e variáveis de retorno

6.1.3 – Variáveis de retorno

Em um Diagrama de Colaboração, pode ser necessário armazenar certos resultados de invocação de mensagens para serem utilizados posteriormente como, por exemplo, para serem passados como parâmetros em outras mensagens. Para isso, pode-se utilizar variáveis, como ilustrado na Figura 6.5. Na mensagem número 2, cujo nome é **obterTipoLeitor**(), obtêm-se o tipo do leitor (por exemplo, aluno, professor, etc.), que é atribuído a uma variável **t**. Esse tipo é passado como parâmetro para a criação da **linhaDoEmprestimo** (mensagem número 3). Nesse caso específico, **linhaDoEmprestimo** utiliza **t** para calcular a data de devolução do livro.

Um objeto de uma classe específica pode ser retornado como resultado da invocação de uma mensagem e pode ser atribuído a uma variável. Depois disso, esse objeto pode ser invocado diretamente pelo objeto que o atribuiu à variável, como exemplificado na Figura 6.6, em que o objeto **c1** é obtido por **e1** pela invocação da mensagem copia() ao objeto **l1**, e então **e1** pode invocar a mensagem **mudarSituacao** desse objeto **c1**.

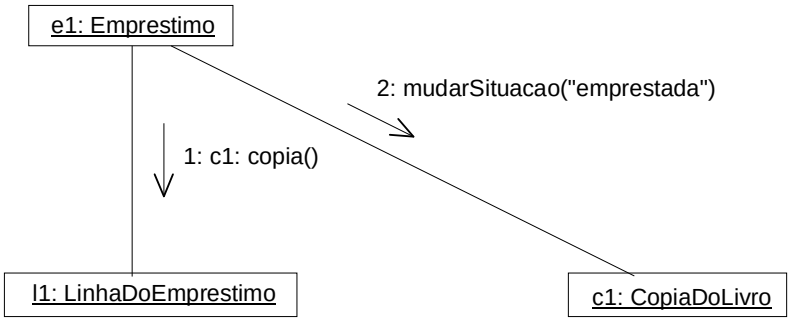


Figura 6.6 – Objeto obtido como retorno de um método

6.1.4 – Execução condicional de mensagem

Pode ser desejável que certas mensagens só sejam executadas em circunstâncias pré-definidas, por exemplo, um empréstimo só pode se concretizar se o leitor estiver apto a emprestar livros (não ultrapassou a quantidade máxima de livros permitidos e não está devendo nenhum livro à Biblioteca). Isso pode ser feito utilizando cláusulas ou predicados booleanos antes do nome da mensagem, como na Figura 6.5, em que a mensagem 1 tem o predicado [novo] como prefixo. Nesse caso, a mensagem só é executada caso a condição contida no predicado seja verdadeira. Analogamente, a mensagem 3 só será executada se a condição [apto] for verdadeira. Pode-se utilizar, dentro dos predicados, nomes de variáveis criadas anteriormente no diagrama de colaboração, ou passadas como parâmetros por outros métodos.

Operadores lógicos e relacionais podem ser utilizados para combinar várias condições, por exemplo **[naoEstaEmAtraso] and [nroLivros < maximoPermitido]** (Figura 6.7). Pode-se desviar o fluxo da execução das mensagens para um método ou outro, simulando um comando condicional, colocando-se **[condição]** em um deles e **[not condição]** no outro (Figura 6.8).

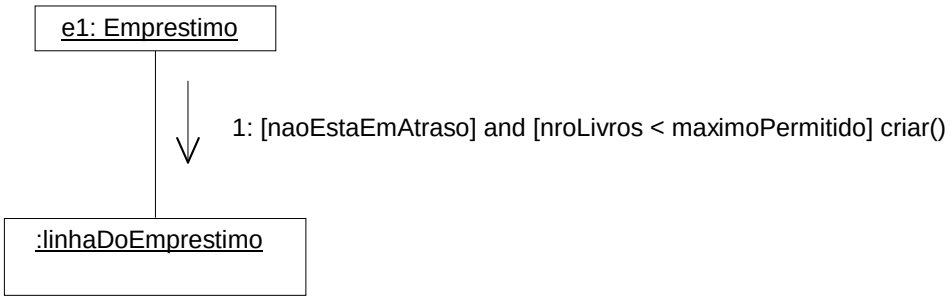


Figura 6.7 – Operadores e condicionais

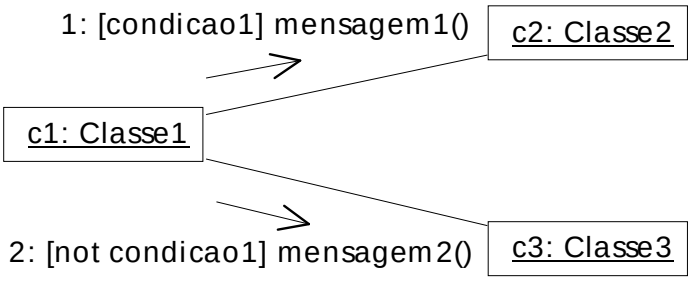


Figura 6.8 – Condicionais opostas

6.1.5 – Repetição de mensagem

Em alguns casos, uma mensagem pode precisar ser enviada várias vezes ao objeto, por exemplo em um laço. Isso pode ser denotado pelo uso de palavras chave tais como **para cada**, **repita**, etc., conforme ilustrado na Figura 6.9. Deve-se notar que a mensagem1 é enviada várias vezes, sempre ao objeto **c2**. Caso não seja possível dizer exatamente quantas vezes a mensagem é enviada, pode-se usar o símbolo *****.

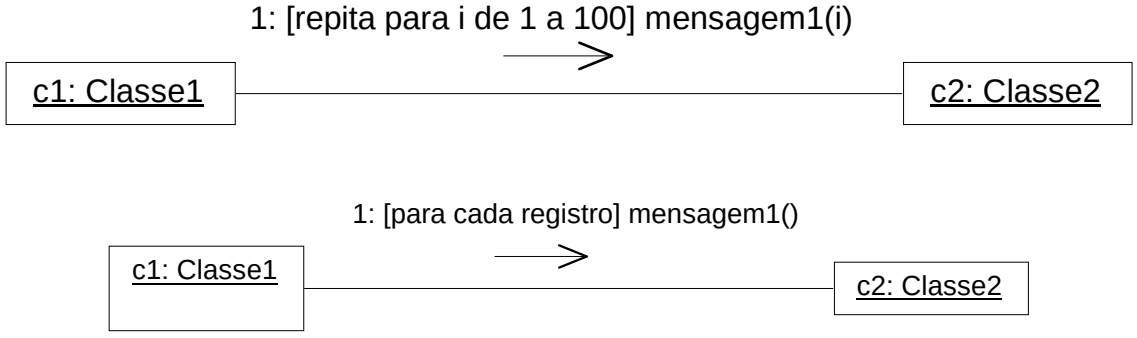


Figura 6.9 – Repetição

6.1.6 – Mensagem para coleção de objetos

Uma mensagem pode ser enviada a vários objetos, ou seja, para uma coleção de objetos, que denominamos um **multi-objeto**. Por exemplo, para obter os títulos dos livros emprestados pelo leitor em uma certa data, deve-se percorrer todas as linhas de empréstimo do empréstimo em questão, obtendo de cada uma delas o título do livro correspondente.

Para denotar um multi-objeto em UML desenha-se um retângulo com várias sombras, como mostrado na Figura 6.10. Deve-se notar que a mensagem **obterTituloDoLivro** é enviada uma única vez para cada um dos objetos **linhaDoEmprestimo** que pertencem à coleção de linhas de empréstimo do empréstimo atual.

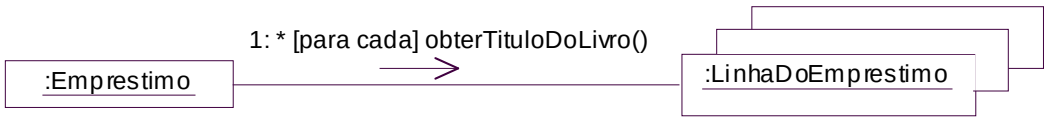


Figura 6.10 – Mensagem para coleção de objetos

A Figura 6.11 mostra um exemplo mais elaborado, no qual cada título de livro obtido é armazenado em uma outra coleção, neste caso uma coleção de *strings* cujo nome é **titulosEmprestados**. Ela é inicialmente criada (coleção vazia), por um método bastante comum em diagramas de colaboração, o **criar**. Para percorrer a coleção de linhas do empréstimo, usa-se um outro método comum na modelagem de diagramas de colaboração, chamado **próximo**, para obter o próximo elemento da coleção, que é atribuído à variável **linha**. O objeto **linha** pode então ser invocado para responder o título do livro, que é finalmente adicionado à coleção **titulosEmprestados.** Isso se repete enquanto houverem linhas de empréstimos a serem percorridas na coleção.

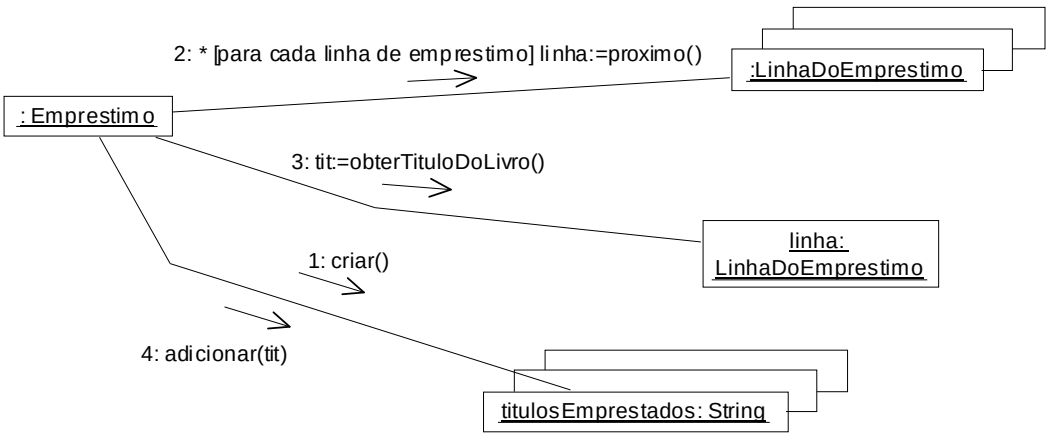


Figura 6.11 – Como percorrer uma coleção de objetos

6.1.7 – Mensagem para o próprio objeto

Uma mensagem pode ser enviada de um objeto para ele mesmo (auto-mensagem), denotando que o próprio objeto entende e processa a mensagem. Por exemplo, na Figura 6.12, o objeto **l1**, da classe Livro, recebe a mensagem **ehDeConsulta**, processa-a e armazena seu resultado na variável boolena cons, que depois é utilizada como condição para mudar ou não a situação da cópia do livro.

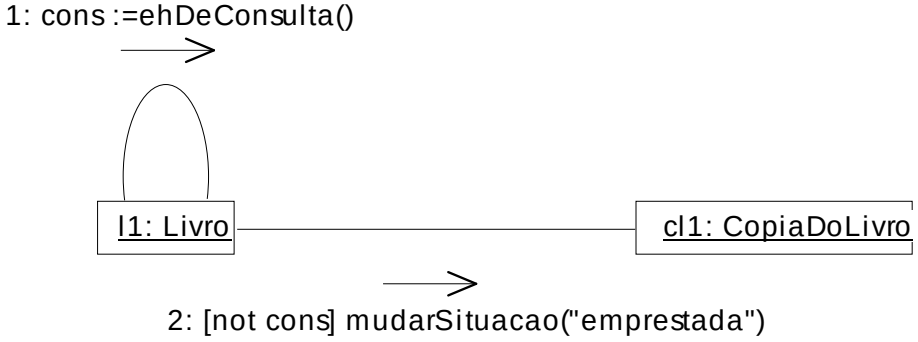


Figura 6.12 – Auto-Mensagem

6.1.8 – Classes x Instâncias

Na maioria das vezes, o diagrama de colaboração mostra troca de mensagens entre objetos das classes, ou seja, um objeto envia uma mensagem a um outro objeto que ele conhece. Porém, pode ser necessário enviar uma mensagem à classe propriamente dita, ao invés de a um objeto. Por exemplo, pode ser necessário saber o nome da classe, ou quais são seus atributos. Nesse caso, a notação é a mostrada mais à esquerda na Figura 6.13.

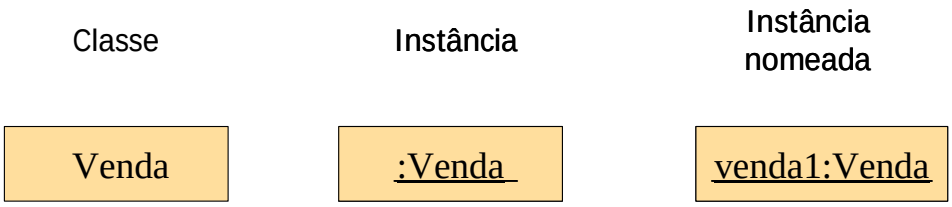


Figura 6.13 – Classe X Instância X Instância Nomeada

A notação utilizada nas seções anteriores foi a de instância, que pode ser tanto uma instância sem nome quanto uma **instância nomeada**. Em geral, uma sugestão básica é dar nome à instância sempre que for utilizá-la posteriormente no mesmo diagrama de colaboração, por exemplo, se for passar o objeto como parâmetro para invocação de um outro método. Outra utilidade em nomear a instância é facilitar a compreensão do projeto OO quando ele tiver que ser comparado ao código-fonte produzido. Instâncias nomeadas provavelmente serão transformadas em variáveis pelo programador que implementará o projeto. Dessa forma, o programador não precisará inventar o nome da variável, ou seja, utilizará o nome de instância fornecido pelo diagrama de colaboração.

6.2 – Atribuição de Responsabilidades

6.2.1 – Tipos de responsabilidade

Na seção 6.1 foi apresentada em detalhes a notação dos diagramas de colaboração. Sabemos que os objetos precisam se comunicar para atingir seus objetivos, ou seja, cumprir suas responsabilidades. Os Diagramas de colaboração mostram escolhas de atribuição de responsabilidade aos objetos. Mas como decidir quem é o melhor candidato para realizar cada uma das operações ou métodos do sistema?

Comecemos definindo o que é uma responsabilidade: é um contrato ou obrigação de um tipo ou classe. Pode-se também pensar que uma responsabilidade é um serviço fornecido por um elemento (classe ou subsistema). Nesse sentido, há dois tipos de responsabilidades básicas: Fazer e Saber.

A responsabilidade do tipo **fazer** implica na execução de algo, por exemplo, criar um objeto, calcular um valor, atribuir um valor a um atributo, iniciar ações em outros objetos (delegação), coordenar e controlar atividades em outros objetos. Já a responsabilidade de **saber** implica em conhecer dados privados encapsulados, conhecer objetos relacionados ou conhecer dados que podem ser derivados ou calculados.

Para projetar um bom diagrama de colaboração, decisões de projeto precisam ser tomadas pelo engenheiro de software. Nesse ponto, a qualidade do projeto depende de um bom entendimento das técnicas e princípios de atribuição de responsabilidade. Várias soluções para um mesmo problema podem ser encontradas por diferentes engenheiros de software. Como saber qual delas é melhor?

6.2.2 – Problemas causados quando a distribuição de responsabilidades é ruim

Para entender melhor os problemas causados por um mau projeto, considere o diagrama de colaboração para a operação “emprestarCopia”, mostrado na Figura 6.14. O que podemos notar de errado nesse diagrama?

Se analisarmos a interação entre os objetos, existe uma forte interação entre o objeto da classe Biblioteca e outros objetos do sistema. Pode-se dizer que toda a lógica está concentrada neste objeto, ou seja, ele coordena a operação, chamando métodos de várias classes.

O problema com esse tipo de solução é que o objeto biblioteca fica muito complexo, pois possui a responsabilidade de se comunicar com muitos objetos e acaba realizando tarefas que deveriam ser de responsabilidade de outros objetos. Por exemplo, é ele quem cria o ItemDeEmprestimo, é ele quem associa o Empréstimo ao ItemDeEmprestimo, é ele quem associa o ItemDeEmprestimo à cópia do livro, etc.

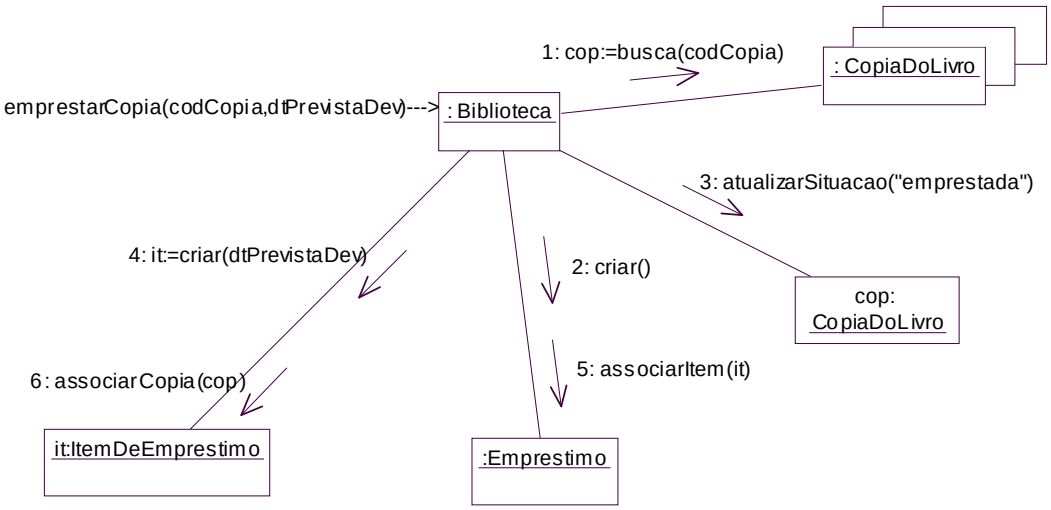


Figura 6.14 – Possível diagrama de colaboração para a operação emprestarCopia

Um projeto como esse pode ser muito difícil de entender e modificar, além de ter menor chance de ser reutilizado em outros desenvolvimentos, pois algumas classes possuem responsabilidades em excesso, muitas das quais não relacionadas ao propósito da classe em si. Para evitar esse problemas, é necessário utilizar princípios bem aceitos na comunidade de orientação a objetos, conhecidos como princípios de atribuição de responsabilidades. Esses princípios podem nos guiar na atribuição de responsabilidades, levando a projetos melhores. Na seção 6.3 serão apresentadas várias soluções para evitar projetos com má distribuição de responsabilidades, como o da Figura 6.14.

6.3 – Padrões GRASP

6.3.1 – Padrões

Desenvolvedores de software experientes criaram um repertório de princípios gerais e boas soluções para guiar a construção de software. Um **padrão** é uma descrição nomeada de um problema e uma solução, que pode ser aplicado em novos contextos. Essas soluções são descritas em um formato padronizado (nome, contexto, problema, solução, entre outros) e podem ser usadas em outros contextos.

Os padrões surgiram com base no trabalho do arquiteto Christopher Alexander [1977], um arquiteto que propôs padrões arquiteturais para construção de casas, bairros, cidades, etc. Em software, os padrões ganharam impulso após a publicação do livro *Design Patterns,* por Gamma e outros [1994], chamada a Gangue dos Quatro (do inglês: *Gang of Four* – GoF). Eles apresentam vinte e três padrões de projeto, que podem ser utilizados durante o projeto de software OO, melhorando o projeto e tornando-o mais flexível e reusável. Neste livro abordaremos vários desses padrões no Capítulo 9.

Padrões usualmente não contêm novas idéias, mas organizam conhecimentos e princípios existentes, testados e consagrados e, portanto, que ajudam a melhorar a produtividade e a qualidade dos artefatos resultantes. Existem padrões em diversos níveis de abstração, tais como padrões de análise, arquiteturais, de processo, de interface com o usuário, de testes, de projeto e de programação.

6.3.2 – Padrões GRASP

Os padrões GRASP (do inglês: *General Responsibility Assignment Software Patterns*) [Larman, 2004] descrevem princípios fundamentais de atribuição de responsabilidade a objetos. Podem ser classificados como padrões de processo, já que ajudam os desenvolvedores no processo de atribuição de responsabilidade. Apesar deles serem utilizados na fase de projeto de software, não propõem soluções para problemas específicos de projeto, como é o caso dos padrões da GoF, mas fornecem princípios para melhor distribuir as responsabilidades entre as classes de projeto.

Deve-se ressaltar que os padrões GRASP devem ser utilizados em conjunto para obter um bom projeto. Alguns deles são bastante relacionados, de tal forma que, usando um, automaticamente o outro também é usado. Alguns são específicos e você logo perceberá quando tem que usá-los, porque o problema que resolvem é bem fácil de identificar. Outros são mais genéricos, tornando mais difícil identificar qual padrão é mais adequado a qual situação.

A preocupação quando se projeta um diagrama de colaboração é, geralmente, como melhor distribuir as responsabilidades entre os objetos. Os padrões GRASP oferecem sugestões efetivas de boas soluções para esse problema. Às vezes é necessário tentar várias soluções até encontrar a melhor delas.

Alguns dos principais padrões GRASP são: Especialista (*Expert*), Criador (*Creator*), Coesão alta (*High Cohesion*), Acoplamento fraco (*Low Coupling*) e Controlador (*Controller*), que serão vistos nas subseções seguintes. Cada um dos padrões possui elementos como problema, solução, exemplo, discussão, benefícios e contra-indicação, se for o caso.

6.3.3 – Padrão Especialista

Este padrão é o ponto de partida ao pensar na atribuição de responsabilidade, ou seja, ele deve ser considerado antes de se pensar em qualquer outra solução.

Problema: qual é o princípio mais básico de atribuição de responsabilidades a objetos ?

Solução: Atribuir responsabilidade ao especialista da informação.

Exemplo: no sistema de biblioteca, quem seria o responsável por calcular a data de devolução de um livro?

Examinando o Modelo Conceitual da Figura 4.15, verifica-se que a data de devolução fica armazenada no atributo **data_prevista_devolução** do objeto LinhaDoEmprestimo. A pergunta que se deve fazer, em termos do padrão Especialista, é: quem possui conhecimentos necessários para calcular a data de devolução? A responsabilidade em questão deve ser atribuída a tal classe. Examinando os requisitos do sistema (requisito R2 da Tabela 3.1), conclui-se que Leitor é o especialista nesta informação, pois conhece o tipo de Leitor (por exemplo, aluno de graduação, aluno de pós-graduação, professor, etc), que é utilizado para calcular a data em que o livro deve ser devolvido. Portanto, pelo padrão Especialista, a responsabilidade calcularDataDevolução é atribuída a Leitor, conforme ilustrado pela mensagem 1 do diagrama de colaboração da Figura 6.15.

Esta figura ilustra como a classe Empréstimo cumpre a responsabilidade de adicionar uma cópia de livro a si própria, ou seja, dada uma cópia do livro (**copiaLivro**), calcula-se a data de devolução e depois cria-se um objeto **linha**, da classe LinhaDoEmprestimo. No momento de criação do objeto **linha**, ele já é automaticamente associado ao empréstimo atual e à cópia do livro. Além disso, o valor **d** calculado pelo objeto Leitor é passado como parâmetro para que **linha** crie o objeto e já atribua o valor inicial ao seu atributo data_prevista_devolução.

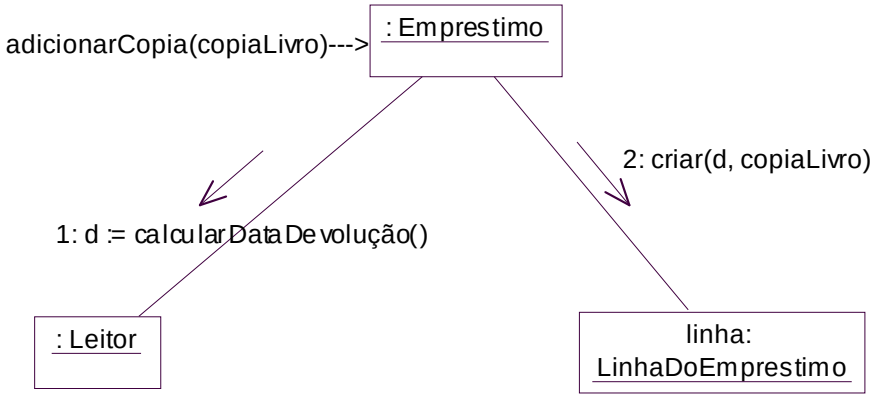


Figura 6.15 – Diagrama de colaboração para adicionarCopia

Nesse diagrama, a associação do empréstimo a **linha** é implícito e a associação de **linha** à **copiaLivro** é deduzida pelo fato do método **criar** receber copiaLivro como parâmetro. Esta é a forma sugerida por Larman [2004] para denotar a criação de associações entre objetos. Deve-se notar que isso implica que, ao criar um objeto, é necessário passar como parâmetros todos os objetos que devem ser associados a ele. Já Wazlawick [2004] prefere mostrar explicitamente no diagrama de colaboração essa criação de associações entre objetos, como pode ser visto na Figura 6.16.

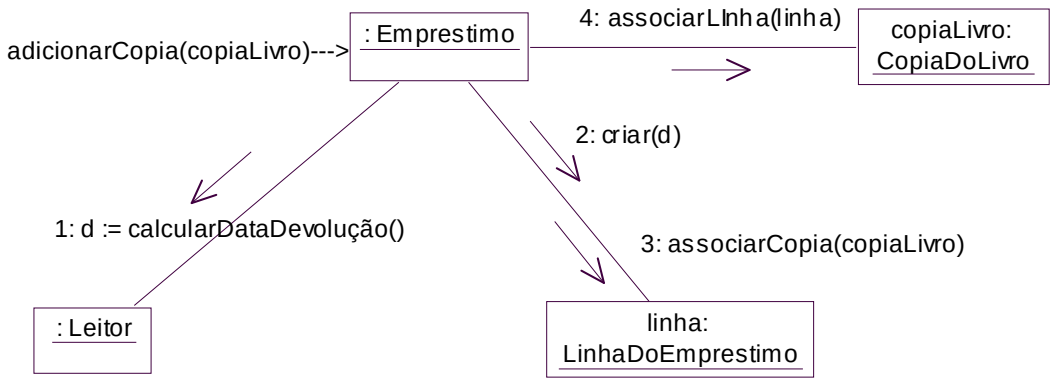


Figura 6.16 – Versão alternativa de Diagrama de colaboração para adicionarCopia

Discussão

Um ponto importante que surge ao usar o padrão Especialista é sobre onde procurar pela classe especialista. Quando o projeto de outros diagramas de colaboração já permitiu identificar várias classes que farão parte do sistema, comece olhando essas classes. Se não encontrar, utilize o Modelo Conceitual do sistema, considerando que os conceitos realmente utilizados como parte dos diagramas de colaboração passarão a ser denominados de classes de projeto.

Também, deve-se lembrar que existem especialistas parciais, que colaboram numa tarefa, portanto pode ser difícil encontrar o especialista, porque a informação fica espalhada por diversas classes. A filosofia do padrão Especialista tem uma analogia no mundo real: se não souber fazer, delegue a responsabilidade a um especialista no assunto.

Benefícios

Usando-se o padrão Especialista, consegue-se manter o encapsulamento, pois cada classe faz o que realmente tem conhecimento para fazer. Favorece-se o acoplamento fraco e a alta coesão, que são conceitos que serão vistos mais a frente, além do comportamento ficar distribuído entre as classes que têm a informação necessária, tornando as classes mais "leves". O reuso é favorecido, pois ao reutilizar uma classe sabe-se que ela oferece todo o comportamento inerente e esperado.

Contra-indicações

O padrão Especialista é contra indicado quando seu uso faz aumentar o acoplamento e reduzir a coesão das classes. Por exemplo, quem é responsável por salvar um Empréstimo no banco de dados? Salvar é uma responsabilidade largamente utilizada por todas as classes de um modelo, portanto atribuir a responsabilidade a cada uma das classes faz com que a classe perca o foco nas responsabilidades que lhe são intuitivamente inerentes. Uma

possível solução para isso é ter uma classe que cuida da persistência do objeto e fazer com que todas as classes herdem dela, conforme será explicado no Capítulo 9.

6.3.4 – Padrão Criador

Este padrão é específico para a responsabilidade de criar novos objetos.

Problema: Quem deveria ser responsável pela criação de uma nova instância de alguma classe ?

Solução: atribua à classe B a responsabilidade de criar uma nova instância da classe A se uma das seguintes condições for verdadeira:

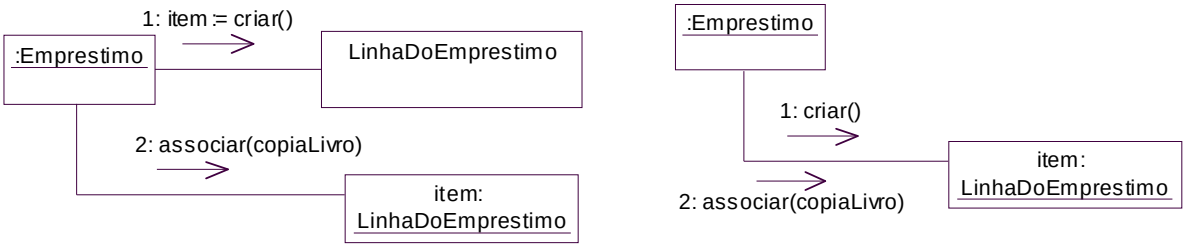
- B agrega objetos de A
- B contém objetos de A
- B registra objetos de A
- B usa objetos de A
- B tem os valores iniciais que serão passados para objetos de A, quando de sua criação

Exemplo: No sistema da Biblioteca, quem é responsável pela criação de uma LinhaDoEmprestimo ?

Se examinarmos o Modelo Conceitual da Figura 4.15, veremos que Empréstimo possui (ou agrega) várias LinhaDoEmpréstimo. Portanto Empréstimo é a classe responsável pela criação de objetos LinhaDoEmpréstimo, conforme mostrado na Figura 6.13.

Discussão

Se examinarmos a sintaxe da UML para o envio de mensagens, concluiremos que a mensagem criar() deve ser enviada à classe LinhaDoEmpréstimo, ao invés de ser enviada ao objeto criado. Portanto, o que fazemos nas Figuras 6.15 e 6.16 é uma simplificação (ou atalho) para a criação de um objeto. A Figura 6.17 ilustra isso: na parte (a), o objeto **it** é criado pela invocação da mensagem criar() à classe LinhaDoEmpréstimo. Posteriormente, este objeto **item** é utilizado (ele recebe a mensagem *associar*). Já na parte (b), o objeto também é criado pela classe LinhaDoEmpréstimo, mas isso fica implícito. Considera-se que antes da mensagem 1 o objeto **item** ainda não existia, que ele passa a existir logo após o retorno da mensagem e pode ser utilizado a partir de então.



(a) Cria e depois usa o objeto

(b) cria e já usa o próprio objeto

Figura 6.17 – Diferentes formas de criar um objeto

Vantagens

O objetivo do padrão Criador é definir como criador o objeto que precise ser conectado ao objeto criado em algum evento. Dessa forma, o acoplamento permanece menor, pois de qualquer maneira o objeto já seria ligado ao seu criador. Objetos agregados, contêineres e registradores são bons candidatos à responsabilidade de criar outros objetos. Pode acontecer também do candidato a criador ser o objeto que conhece os dados iniciais do objeto a ser criado. Isso evita passar muitos parâmetros, por exemplo, se a mensagem de criação tivesse que ser delegada a outra classe.

6.3.5 – Padrão Acoplamento Fraco

Acoplamento é a dependência entre elementos (por exemplo, classes ou subsistemas). Em geral, o acoplamento resulta da colaboração entre esses elementos para atender a uma responsabilidade. Em particular na orientação a objetos, o acoplamento mede o quanto um objeto está conectado a, tem conhecimento de, ou depende de outros objetos. Pode-se dizer que o acoplamento é fraco (ou baixo) se um objeto não depende de muitos outros e que o acoplamento é forte (ou alto) se um objeto depende de muitos outros. Embora subjetiva, essa afirmação nos dá indicações de que um bom projeto é aquele que proporciona o menor acoplamento possível.

O acoplamento alto pode causar vários problemas: mudanças em classes interdependentes forçam mudanças locais, ou seja, se uma classe A está acoplada às classes B e C, mudanças em B e C podem exigir que A seja modificada para preservar seu comportamento. Além disso, quando uma classe está conectada a muitas outras, para entendê-la é necessário entender também essas outras, o que dificulta a compreensão do objetivo de cada classe. Isso implica em dificuldade em reutilizar a classe, pois todas as classes acopladas também precisam ser incorporadas para reuso.

O padrão Acoplamento Fraco tem por objetivo justamente favorecer o menor acoplamento possível entre as classes de projeto.

Problema: como favorecer a baixa dependência e aumentar a reutilização ?

Solução: Atribuir responsabilidades de maneira que o acoplamento permaneça baixo.

Exemplo: No sistema de biblioteca, suponha que queremos realizar a devolução da cópia do livro. Que classe deve ser responsável por essa tarefa? A Figura 6.18 mostra uma solução, na qual atribuiu-se a responsabilidade de devolver cópia do livro ao objeto Leitor. Essa solução aumenta o acoplamento inicialmente concebido no Modelo Conceitual da Figura 4.15, pois inclui uma associação entre Leitor e CopiaDoLivro. Além disso, a mensagem enviada de CopiaDoLivro para LinhaDoEmprestimo, que havia sido concebida no sentido inverso (de LinhaDoEmprestimo para CopiaDoLivro), também aumenta o acoplamento.

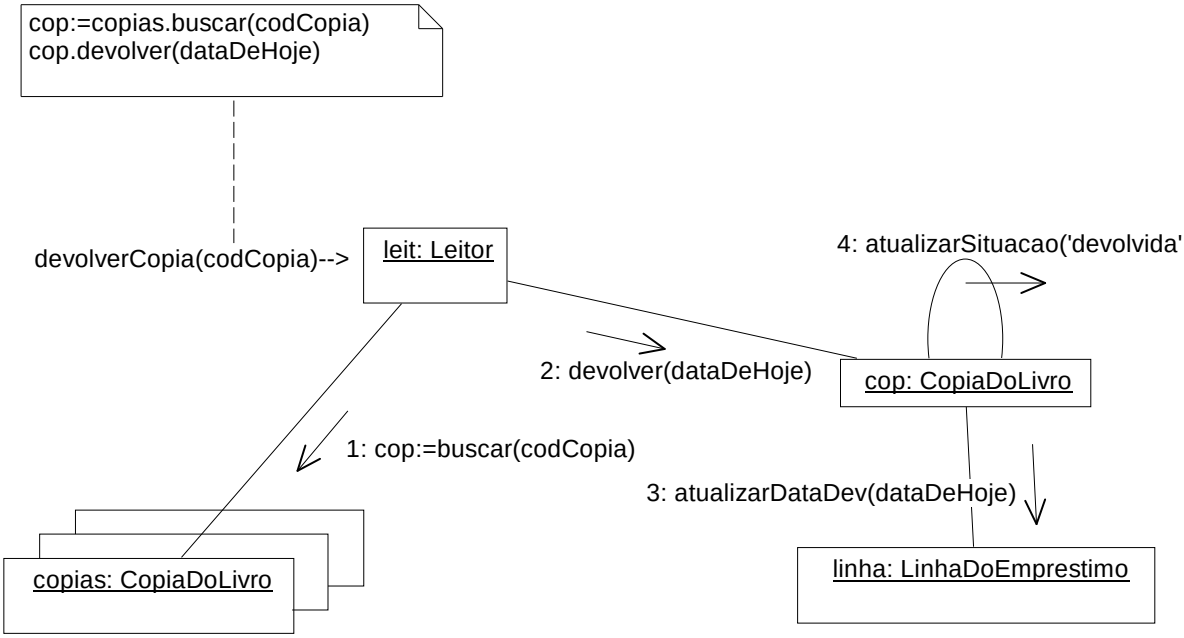


Figura 6.18 – Solução que aumenta o acoplamento

A Figura 6.19 mostra outra solução, melhor em relação à Figura 6.18, mas que também apresenta o problema de aumentar o acoplamento, por incluir um envio de mensagem entre CopiaDoLivro e LinhaDoEmprestimo. A solução foi atribuir a responsabilidade de devolver cópia do livro ao objeto Livro, que conhece suas cópias e pode buscar aquela que possui o código recebido como parâmetro.

A Figura 6.20 apresenta uma terceira alternativa de solução, dessa vez sem aumentar o acoplamento inicialmente concebido no Modelo Conceitual da Figura 4.15. A solução foi atribuir a responsabilidade de devolver cópia do livro ao objeto Empréstimo. Empréstimo conhece suas linhas de empréstimo e, portanto, pode fazer uma busca para encontrar aquela que possui o código de cópia igual ao parâmetro recebido (codCopia). Depois disso, ele pode delegar a responsabilidade de concretizar a devolução a esse objeto encontrado. Veja que o sentido da mensagem entre LinhaDoEmprestimo e CopiaDoLivro permanece, sem aumentar o acoplamento.

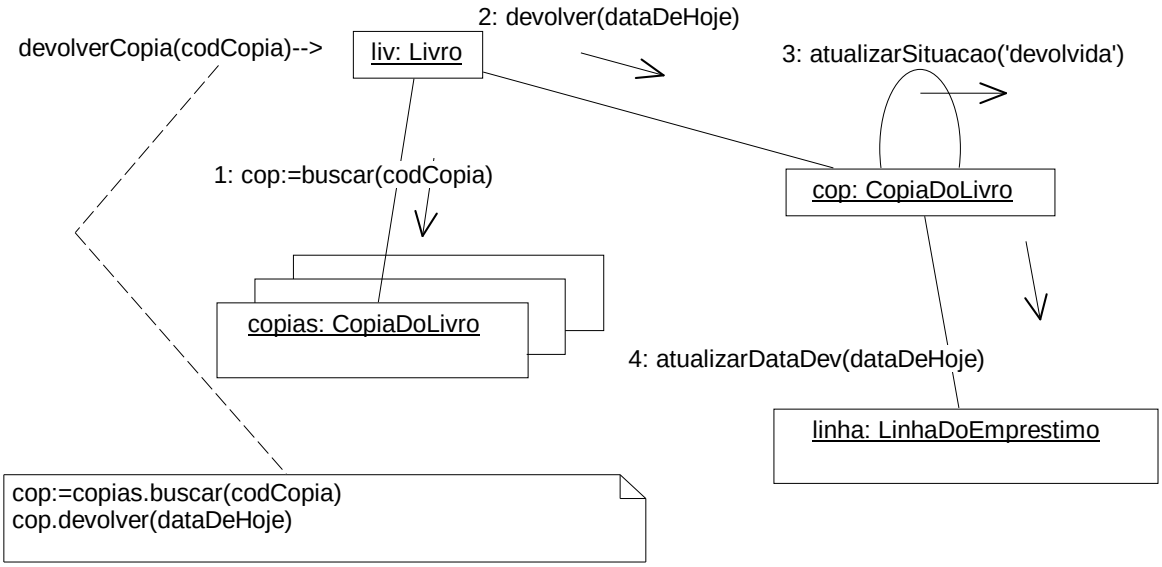


Figura 6.19 – Outra solução que aumenta o acoplamento

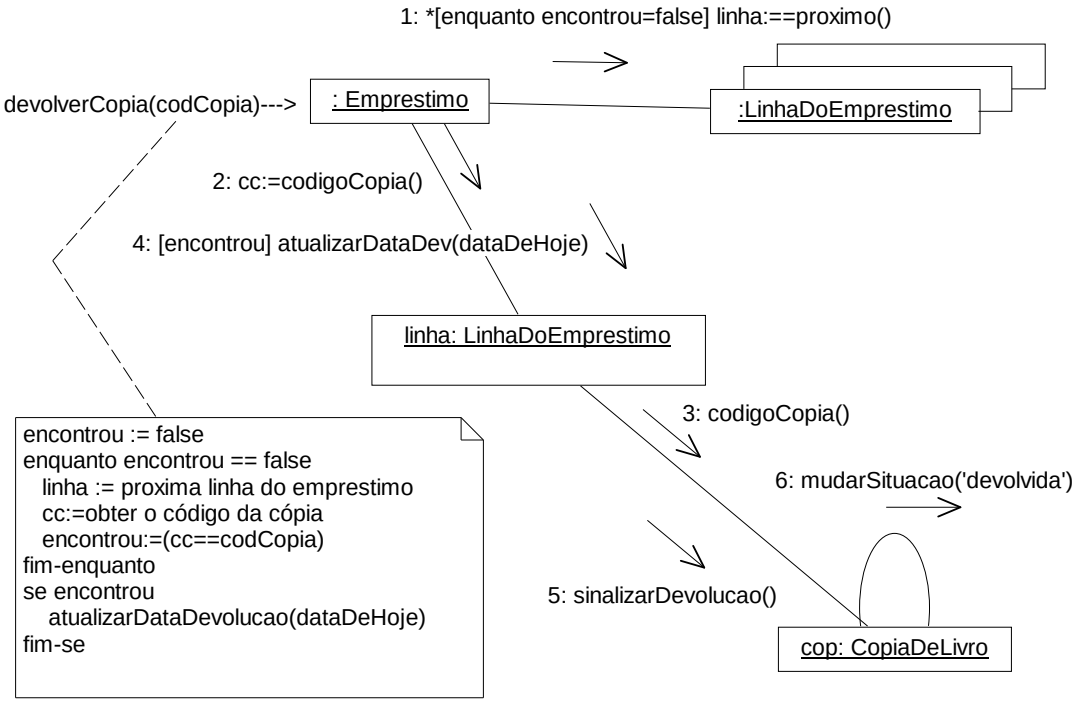


Figura 6.20 – Solução que mantém o acoplamento

Discussão

Existem várias formas de acoplamento entre objetos. Um objeto pode ter um atributo que referencia um objeto de outra classe. Por exemplo, o Empréstimo possui uma referência ao Leitor que realizou o empréstimo. Isso será implementado depois como um atributo na

classe Empréstimo. Porém, não é recomendável colocar explicitamente esse atributo na classe, conforme será visto no capítulo 7, pois o fato de existir a associação já implica na existência do acoplamento.

Uma outra forma de acoplamento é quando um objeto possui um método que referencia um objeto de outra classe. Por exemplo, a classe A pode ter um método que declara e cria um objeto da classe B, o que implica um acoplamento de A para B. Isso pode ocorrer também se a classe A invocar um método da classe B, passando como parâmetro um objeto da classe C. Nesse caso, B será acoplada a C, por meio desse parâmetro. Ou a classe A pode receber como retorno da invocação de um método qualquer um objeto da classe B, por isso fica acoplada à classe B. Além desses casos, diz-se também que, se uma classe A é subclasse da classe B, direta ou indiretamente, então A e B são acopladas.

O extremo de acoplamento fraco não é desejável, por ferir os princípios da tecnologia de objetos, que é o de comunicação por mensagens. Se uma classe não se comunica com outras, ela acaba ficando com excesso de responsabilidades (ver padrão Coesão Alta), o que também é indesejável. Isso leva a projetos pobres: objetos inchados e complexos, responsáveis por muito trabalho.

Vantagens

O acoplamento fraco resulta em classes mais independentes, reduz o impacto de mudanças e favorece reuso de classes. Porém, o padrão Acoplamento Fraco deve ser considerado em conjunto com outros padrões, balanceando suas vantagens e desvantagens. Outra vantagem é que as classes são simples de entender isoladamente, pois não é necessário entender muitas outras classes associadas.

6.3.6 – Padrão Coesão Alta

A **Coesão** mede o quanto as responsabilidades de um elemento (classe, objeto, subsistema,…) são fortemente focalizadas e relacionadas. Um objeto com coesão alta é um objeto cujas responsabilidades são altamente relacionadas e que não executa um volume muito grande de trabalho. Já um objeto com coesão baixa é um objeto que faz muitas coisas não relacionadas ou executa muitas tarefas, o que o torna mais difícil de compreender, reutilizar e manter, além de ser constantemente afetado por mudanças. Isso motivou o estabelecimento do padrão Coesão alta.

Problema: Como manter a complexidade sob controle?

Solução: Atribuir responsabilidade de tal forma que a coesão permaneça alta.
Exemplo: (o mesmo para o acoplamento fraco): No sistema de biblioteca, suponha que queremos realizar a devolução da cópia do livro. Qual classe deve ser responsável por essa tarefa?

A Figura 6.18, já vista anteriormente, ilustra uma solução em que o Leitor fica parcialmente encarregado da devolução da cópia do livro. Em termos de coesão, neste exemplo, isso seria aceitável, mas o que aconteceria se houvesse 50 mensagens recebidas por Leitor? A

resposta é que o leitor perderia sua coesão, pois ficaria responsável por muitas responsabilidades não relacionadas.

Na Figura 6.19 apresentou-se outra solução, em que o Livro ficava parcialmente encarregado da devolução da cópia do livro. Vale aqui o mesmo comentário em relação à primeira solução: o que aconteceria se houvessem 50 mensagens recebidas por Livro?

Finalmente, a Figura 6.20 apresenta a melhor solução em termos de coesão (além de ser melhor em termos de acoplamento, como já tinha sido visto). O objeto empréstimo representa eventos bem definidos no sistema de biblioteca (empréstimo e devolução), por isso é mais intuitivo que ele assuma esta responsabilidade.

Discussão

O padrão Coesão alta, assim como o Acoplamento Fraco, são princípios que devem ser considerados no projeto de objetos, pois má coesão traz acoplamento ruim e vice-versa. Uma regra prática a ser observada é que uma classe com coesão alta tem um número relativamente pequeno de métodos, com funcionalidades relacionadas, e não executa muito trabalho. Pode-se fazer analogia do conceito de alta coesão com o mundo real, por exemplo, pessoas que assumem muitas responsabilidades não associadas podem tornar-se (e normalmente tornam-se) ineficientes.

Vantagens

O padrão Coesão alta traz mais clareza e facilidade de compreensão ao projeto, simplificando as tarefas de manutenção e acréscimo de funcionalidade ou melhorias, além de favorecer o acoplamento fraco e aumentar o potencial de reutilização das classes, pois classes altamente coesas podem ser usadas para uma finalidade bastante específica.

6.3.7 – Padrão Controlador

A solução da Figura 6.20 ainda apresenta um problema: quando ocorre o evento de devolução da cópia, ainda não é conhecido o objeto **empréstimo** ao qual a cópia emprestada se refere. Portanto, é preciso eleger alguma classe, que conheça os empréstimos, para receber a mensagem **devolverCopia**. Essa classe terá que identificar o objeto empréstimo cujo código de cópia seja igual ao parâmetro fornecido.

Essa necessidade surge porque **devolverCopia** é uma operação do sistema, responsável por tratar os eventos que ocorrem em cada caso de uso. Portanto, é necessário termos princípios ao escolher quais classes do projeto serão responsáveis por tratar as operações do sistema, o que é solucionado pelo padrão Controlador.

Problema: Quem deve ser responsável por tratar um evento do sistema ?

Solução: A responsabilidade de receber ou tratar as mensagens de eventos (operações) do sistema pode ser atribuída a uma classe que:

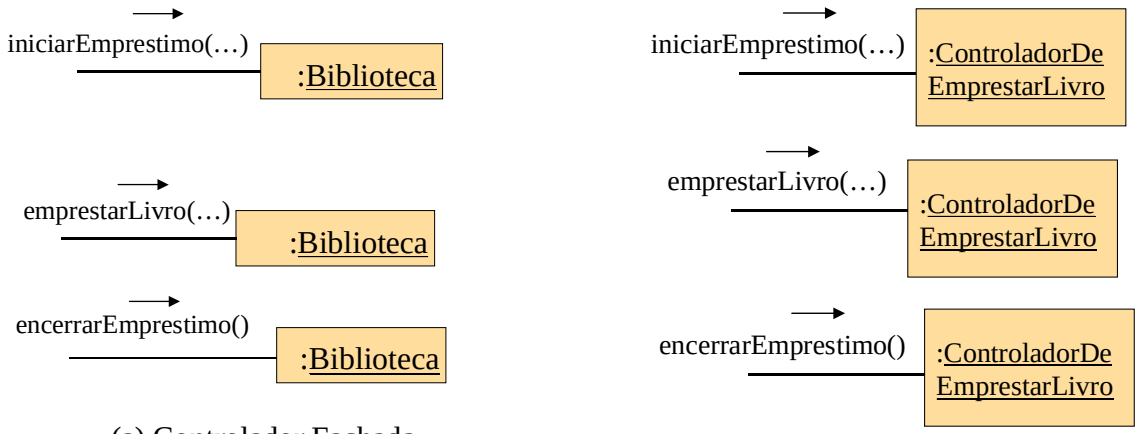
- represente todo o sistema, um dispositivo ou um subsistema – chamado de **controlador fachada** - OU
- represente um cenário de um caso de uso dentro do qual ocorra o evento, chamado de **controlador artificial**, por exemplo um TratadorDe<NomeDoCasoDeUso> ou ControladorDe<NomeDoCasoDeUso>

Exemplo: quem vai tratar os eventos do sistema de biblioteca? Conforme visto na Figura 5.3, o caso de uso Emprestar Livro foi particionado em três operações: iniciarEmpréstimo(), emprestarLivro() e encerrarEmpréstimo(). Que classe será responsável por tratar essas operações?

Um caso de uso possui uma seqüência bem definida de operações, portanto a classe que for escolhida como controladora deve manter informações sobre a execução, para prosseguir ou não para as próximas operações. Por exemplo, se a operação iniciarEmprestimo for mal-sucedida porquê o leitor não está apto a emprestar, a classe controladora não deve permitir que o sistema prossiga para a operação emprestarLivro.

Se optarmos pela solução "Controlador Fachada", podemos escolher a Biblioteca como a classe controladora, por ser considerada como representativa do sistema (Figura 6.21a). Nesse caso, essa classe assumiria a responsabilidade pelas operações, delegando-a às classes que considerar mais apropriadas para executar o comportamento específica.

Já se optarmos pela solução "Controlador Artificial", devemos criar uma classe adicional, por exemplo chamada ControladorDeEmprestarLivro, e ela receberá as respectivas responsabilidades (Figura 6.21b).



(a) Controlador Fachada (b) Controlador Artificial

Figura 6.21 – Padrão Controlador

Discussão

Um controlador fachada deve ser um objeto (do domínio) que seja o ponto principal para as chamadas provenientes da interface com o usuário ou de outros sistemas. Embora seja

prática comum entre programadores, atribuir a responsabilidade diretamente à classes de interface gráfica com o usuário (GUI) não é boa prática de projeto e deve-se evitá-la ao máximo, pois o código fica preso a classes dependentes de tecnologia, ao invés de ficar em classes do domínio de aplicação. Ou seja, quando o comportamento dos casos de uso é atribuído a classes da GUI, será bem mais difícil reutilizar o código em aplicações com outras GUI. Portanto, o ideal é utilizar uma classe do domínio, conforme ilustrado na Figura 6.22, para que a camada de interface não possua nenhuma lógica embutida em seu código.

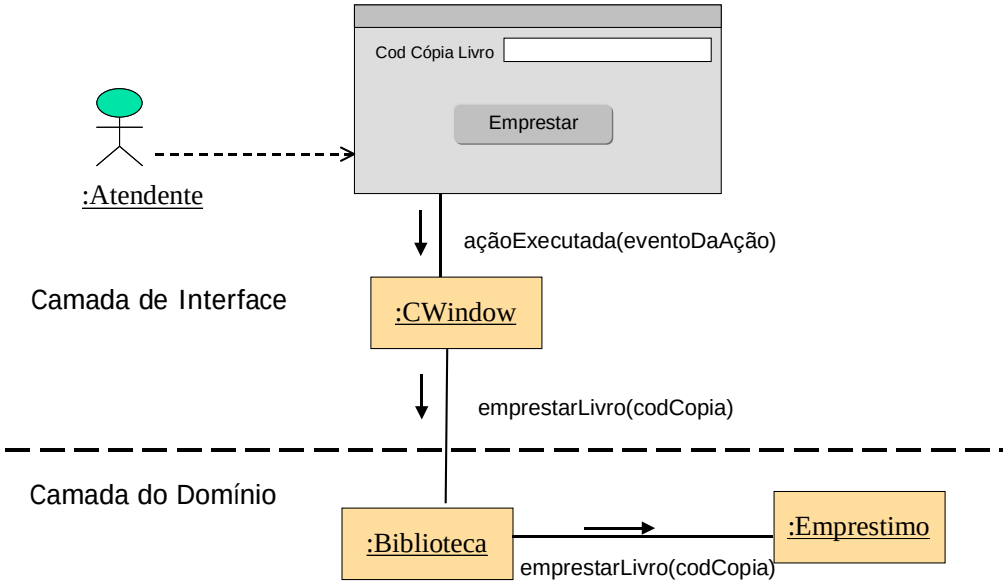


Figura 6.22 – Delegação de responsabilidade ao controlador

O controlador fachada pode ser uma abstração de uma entidade física, por exemplo um TerminalDeAtendimento de uma biblioteca, ou pode ser um conceito que represente o sistema, por exemplo Biblioteca.

Controladores fachada são adequados quando não há muitos eventos de sistema. Caso contrário, as classes ficam inchadas e com baixa coesão. Além disso, não é aconselhável redirecionar mensagens do sistema para controladores alternativos.

Já no caso dos controladores artificiais (ou controladores de casos de uso), deve existir um controlador diferente para cada caso de uso. Por exemplo, o ControladorDeEmprestarLivro será responsável pelas operações iniciarEmpréstimo, emprestarLivro e encerrarEmpréstimo. O controlador artificial não é um objeto do domínio, e sim uma construção artificial para dar suporte ao sistema, por isso pode ser pouco intuitivo de entender por pessoas que não conheçam o padrão. Ele pode ser uma boa alternativa se a escolha de controladores fachada deixar a classe controladora com alto acoplamento e/ou baixa coesão (controlador inchado por excesso de responsabilidades). Assim, ele é uma boa alternativa quando existem muitos eventos envolvendo diferentes processos.

Classes controladoras mal projetadas, ou inchadas, são aquela com coesão baixa (falta de foco e tratamento de muitas responsabilidades). Os sinais típicos de inchaço são:

- uma única classe controladora tratando todos os eventos, que são muitos. Isso é comum com controladores fachada
- o próprio controlador executa as tarefas necessárias para atender o evento, sem delegar para outras classes (coesão alta, não especialista)
- controlador tem muitos atributos e mantém informação significativa sobre o domínio, ou duplica informações existentes em outros lugares

Possíveis curas para controladores inchados são: acrescentar mais controladores, misturar controladores fachada e de casos de uso ou delegar responsabilidades. Um corolário importante é: objetos de interface (como objetos "janela") e da camada de apresentação não devem ter a responsabilidade de tratar eventos do sistema

Benefícios

Com o uso do padrão Controlador, obtem-se um aumento das possibilidades de reutilização de classes e do uso de interfaces "plugáveis". Além disso, pode-se usufruir do conhecimento do estado do caso de uso, já que o controlador pode armazenar o estado do caso de uso, garantindo a seqüência correta de execução das operações.

6.4 – Exercícios Propostos

6.4.1. Com base nos requisitos do sistema Passe-Livre, fornecidos no Apêndice A, nos casos de uso elaborados nos exercícios 3.9.5 e 3.9.6, no Modelo Conceitual elaborado no exercício 4.7.1 e nos DSS e contratos elaborados nos exercícios 5.3.1 e 5.3.2, elaborar o Diagrama de Colaboração para as operações para os quais você elaborou o contrato.

6.4.2. Elabore versões alternativas dos diagramas de colaboração do exercício 6.5.1, considerando outra escolha da classe controladora da operação.

6.4.3. Para os dois exercícios anteriores, identifique os padrões utilizados e defina qual dos projetos é melhor em termos de acoplamento e coesão

6.5 – Exercícios complementares

6.5.1 Projete um diagrama de colaboração para a operação "reservar fita" de um sistema de aluguel de fitas de vídeos, considerando que o objeto **cliente** é responsável por esta operação e deve interagir com os objetos **fita** e **reserva**, verificando se a fita está disponível e criando a reserva. É passado como parâmetro o título do filme.

6.5.2 Re-projete o diagrama do exercício 6.5.1, considerando que deve-se também verificar se o cliente está com sua situação regularizada antes de efetuar a reserva.

Para isso, deve-se também interagir com os objetos da coleção **locação**, referentes a todas as locações recentes do cliente, para verificar se ele tem fitas em seu poder com data de devolução vencida.

6.5.3 Projete um diagrama de colaboração para a operação "devolver fita", considerando que o objeto **locação** é responsável por ela e deve interagir com **fita** para mudar sua situação para "disponível", além de mudar a situação da própria locação para "em ordem". Considere que uma locação refere-se a uma única fita e que a locação conhece a fita à qual se refere.

6.5.4 Re-projete o diagrama do exercício 6.5.3, considerando que o objeto cliente é responsável pela devolução, e é passado como parâmetro o título do filme devolvido.

6.5.5 Projete o diagrama de colaboração para os contratos elaborados no exercício 5.4.3. Mostre os padrões utilizados.

6.5.6 Repita o exercício 6.5.5, mudando sua atribuição de responsabilidade a alguns objetos, e discuta como fica o acoplamento e a coesão.

6.5.7 Projete o diagrama de colaboração para a operação reservarCarro, considerando que contrato da operação é o seguinte:

Operação: reservarCarro(cli, modeloDoCarro, periodoDesejado) ***Referências Cruzadas***: Caso de uso: "Reservar Carro" ***Pré-Condições***: Um cliente apto a emprestar carros já foi identificado e é conhecido do sistema; ***Pós-Condições***: uma nova reserva foi registrada; a nova reserva foi relacionada ao carro do modelo desejado, o atributo periodoReserva recebeu o valor do parâmetro periodoDesejado.

6.5.8 – Re-projete o diagrama de colaboração do exercício 6.5.7, mudando suas escolhas de atribuição de responsabilidades, e comente as vantagens e/ou desvantagens.

Capítulo 7 – Disciplina de Projeto – Visibilidade e Diagramas de Classes de Projeto

7.1 – Introdução

No Capítulo 6 foram introduzidos os diagramas de colaboração, que ilustram a comunicação entre os objetos do sistema para cumprir todas as responsabilidades necessárias para seu funcionamento.

Segundo o PU, na última etapa da fase de projeto deve-se reunir todas as informações presentes nos diagramas de colaboração, principalmente sobre classes e mensagens trocadas, e juntá-las às informações obtidas durante a análise do sistema, com o objetivo de compor as classes de projeto do sistema, que são as classes que serão efetivamente implementadas utilizando uma linguagem de programação orientada a objetos.

O Diagrama de Classes de Projeto é composto de classes relacionadas por associações comuns, de herança ou de agregação. Cada classe possui atributos e métodos. Além disso, no projeto deve-se conhecer de antemão quais classes manterão referências a outras classes, por isso é necessário estabelecer a visibilidade entre classes, explicada na seção 7.2. Depois, com base nos diagramas de colaboração e no modelo conceitual do sistema pode-se finalmente criar os diagramas de classes de projeto (seção 7.3).

7.2 – Visibilidade entre classes

Visibilidade é a capacidade de um objeto ver ou fazer referência a outro: para que um objeto A envie uma mensagem para o objeto B, é necessário que B seja visível para A. Existem quatro tipos de visibilidade: por atributo (B é um atributo de A), por parâmetro (B é um parâmetro de um método de A), localmente declarada (B é declarado como um objeto local em um método de A) e global (B é, de alguma forma, globalmente visível).

7.2.1 – Visibilidade Por Atributo

A visibilidade por atributo ocorre sempre que uma classe possui um atributo que referencia um ou mais objetos de outra classe. Por exemplo, na Figura 7.1 pode-se entender que a classe Empréstimo contém um atributo *leitor*, que é uma referência para o objeto da classe Leitor, mais especificamente para o leitor que efetuou o empréstimo. Assim, objetos da classe Empréstimo podem enviar mensagens ao leitor sempre que precisarem. Dizemos que Empréstimo tem visibilidade por atributo à classe Leitor. No código do método *emprestarCopia()* da classe Emprestimo (Figura 7.1) pode-se ver que é enviada a mensagem *calcularDataDevolução()* ao objeto *leitor*, porque o objeto *empréstimo* o conhece e portanto pode comunicar-se com ele.

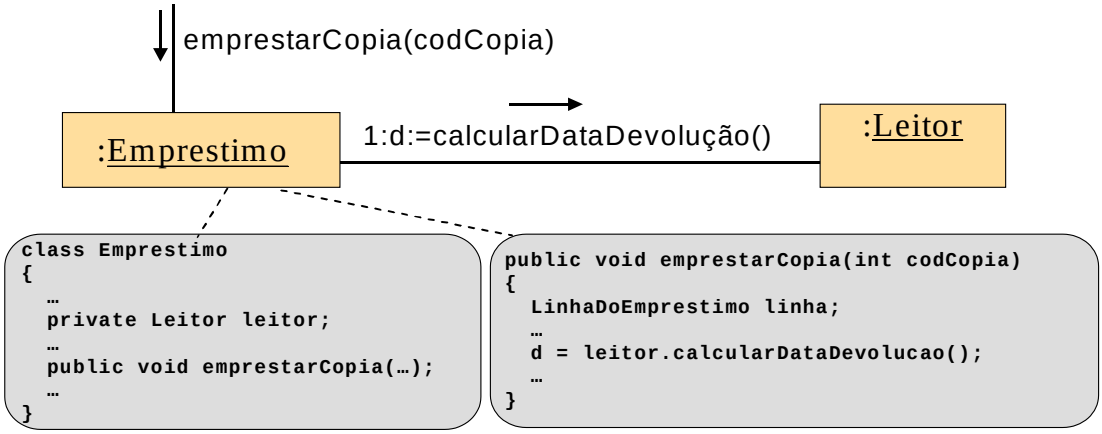


Figura 7.1 – Visibilidade por atributo

A visibilidade por atributo é a forma mais comum de visibilidade. A visibilidade persiste por muito tempo, pois considera-se que o objeto origem terá visibilidade para o destino sempre que precisar. A visibilidade só deixa de existir quando o objeto origem for removido.

Geralmente, a visibilidade por atributo se deve às associações existentes no modelo conceitual, ou seja, para muitas das associações existe visibilidade entre as classes. Resta saber em que sentido deve-se considerar a visibilidade. Por exemplo, existe uma associação de muitos para um entre Empréstimo e Leitor (um empréstimo é feito a um leitor, um leitor faz vários empréstimos). Em tempo de projeto deve-se decidir se a visibilidade entre Empréstimo e Leitor será nos dois sentidos ou se será somente em um sentido (nesse caso, em qual deles). Se considerarmos a visibilidade de Empréstimo para Leitor, a classe Empréstimo terá um atributo *leitor*, denotando o leitor que fez o empréstimo. Já se considerarmos a visibilidade de Leitor para Empréstimo, a classe Leitor deverá ter um atributo *empréstimos* (provavelmente do tipo vetor ou outro tipo de conjunto), denotando o conjunto de empréstimos realizados por um determinado leitor.

Para ajudar a decidir sobre a visibilidade, foi visto no Capítulo 6 que os diagramas de colaboração devem levar em conta os padrões GRASP, de forma que o acoplamento permaneça baixo e a coesão alta. Se no diagrama de colaboração há troca de mensagens entre dois objetos que possuem visibilidade por atributo, o acoplamento é mais forte do que se a visibilidade for dos demais tipos (ver seções seguintes).

Embora uma visibilidade por atributo venha a ser implementada posteriormente como um atributo na classe origem, isso não deve ser mostrado no diagrama de classes. Considera-se que a associação representa esse atributo, e pode-se dar um nome ao papel desempenhado

pelo extremo da associação, para que esse nome venha a ser utilizado durante a programação da classe.

7.2.2 – Visibilidade Por Parâmetro

A visibilidade por parâmetro ocorre sempre que uma classe recebe um objeto como parâmetro em alguma invocação de método e, portanto, conhece o objeto enviado por parâmetro e pode enviar-lhe mensagens. Por exemplo, na Figura 7.2 pode-se entender que a classe Empréstimo recebe o objeto *fita* como parâmetro quando da invocação do seu método *adiciona*. Assim, durante a execução desse método específico (*adiciona*), é possível enviar mensagens ao objeto *fita*, ou seja, *fita* é visível para *emprestimoCorrente* somente temporariamente, por tê-lo recebido como parâmetro. (ver Figura 7.2). No caso do exemplo, o código fonte mostra que não há invocação de nenhum método do objeto *fita*, mas ele é passado adiante como parâmetro na invocação de um outro método, no caso o método *associaFita* da classe ItemDoEmprestimo.

A visibilidade por parâmetro é relativamente temporária, ou seja, ela persiste enquanto persistir o método. Portanto, terminada a execução do método *adiciona*, o objeto *emprestimoCorrente* não terá mais visibilidade para o objeto *fita*.

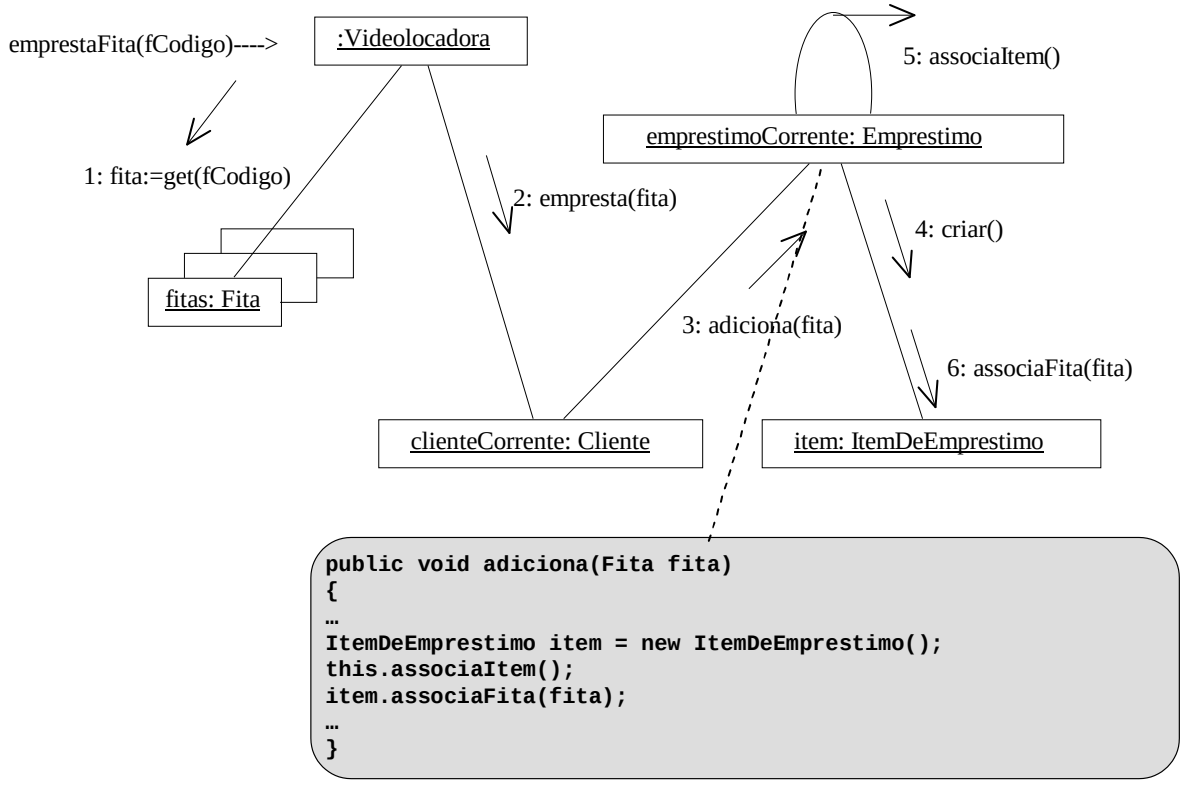


Figura 7.2 – Visibilidade por parâmetro

7.2.3 – Visibilidade Localmente declarada

A visibilidade é dita localmente declarada se o objeto tiver referência a outro que declarou localmente, por meio de uma variável local. Há duas formas de fazer isso: criar uma nova instância local e atribuí-la a uma variável local, ou atribuir o objeto retornado pela invocação de um método a uma variável local. No primeiro caso, considere a Figura 7.2. A variável local *item*, declarada em um método pertencente à classe Empréstimo, faz referência a um objeto da classe ItemDeEmprestimo. Portanto, dizemos que a classe Empréstimo possui visibilidade local para a classe ItemDeEmprestimo.

Esse tipo de visibilidade é relativamente temporária, pois só existe durante a execução do método, da mesma forma que na visibilidade por parâmetro (seção 7.2.2.). Outro exemplo de visibilidade local é quando utiliza-se uma variável local para armazenar o retorno da invocação de um método de outra classe. Observe, por exemplo, o código da Figura 7.3, em que a variável *emprestimoCorrente* é declarada em um método da classe VideoLocadora. Essa variável é utilizada para armazenar o objeto que retorna da invocação do método *clienteCorrente*, da classe Cliente, que por sua vez retorna um objeto da classe Empréstimo. Portanto, dizemos que VideoLocadora tem visibilidade local para Empréstimo.

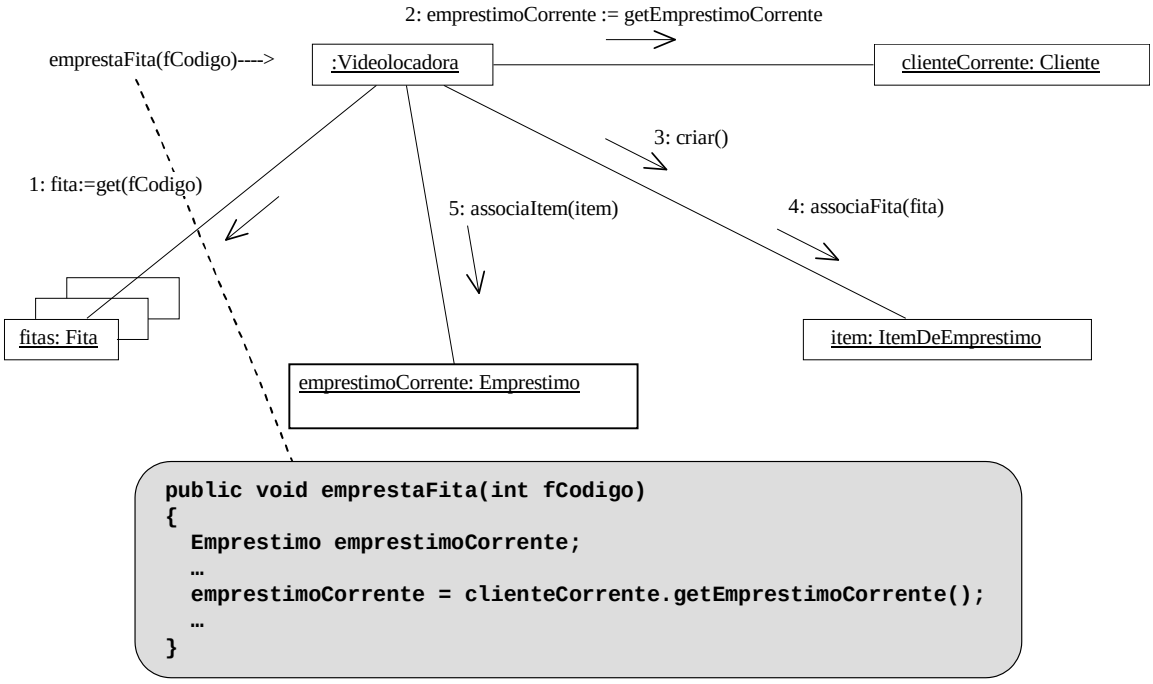


Figura 7.3 – Visibilidade localmente declarada

7.2.4 – Visibilidade Global

O tipo de visibilidade menos comum é a global, que ocorre quando um objeto de uma certa classe é sempre visível a quaisquer outros objetos do sistema. Esse tipo de visibilidade é relativamente permanente, pois persiste enquanto o objeto existir. Uma forma de implementar a visibilidade local é por meio do padrão *Singleton* [Gamma, 1995], descrito

no Capítulo 9. Uma forma óbvia e menos desejável de conseguir visibilidade global é por meio da atribuição de uma instância ou objeto a uma variável global.

7.2.5 – Notação UML para Visibilidade

Em geral, só são mostradas nos diagramas de classes as visibilidades por atributo, por meio de uma seta direcionada na relação de associação entre as classes, partindo da classe que tem visibilidade para a classe para a qual ela tem referência. Isso será mostrado na seção 7.3.

Pode-se também usar estereótipos da UML para denotar a visibilidade nos diagramas de colaboração, conforme indicado na Figura 7.4. Isso é uma indicação explícita do tipo de visibilidade, mais comumente utilizada quando é difícil descobrir o tipo de visibilidade somente pelas informações presentes no diagrama. Em geral, o raciocínio é o seguinte:

- se não houver nenhuma indicação, subentende-se que a visibilidade é por atributo;
- se o objeto X recebe um outro objeto Y como parâmetro, entende-se que ele pode invocar mensagens desse objeto Y; e
- se o objeto cria um novo objeto (usualmente pela mensagem criar() ou novo()), a visibilidade é local.

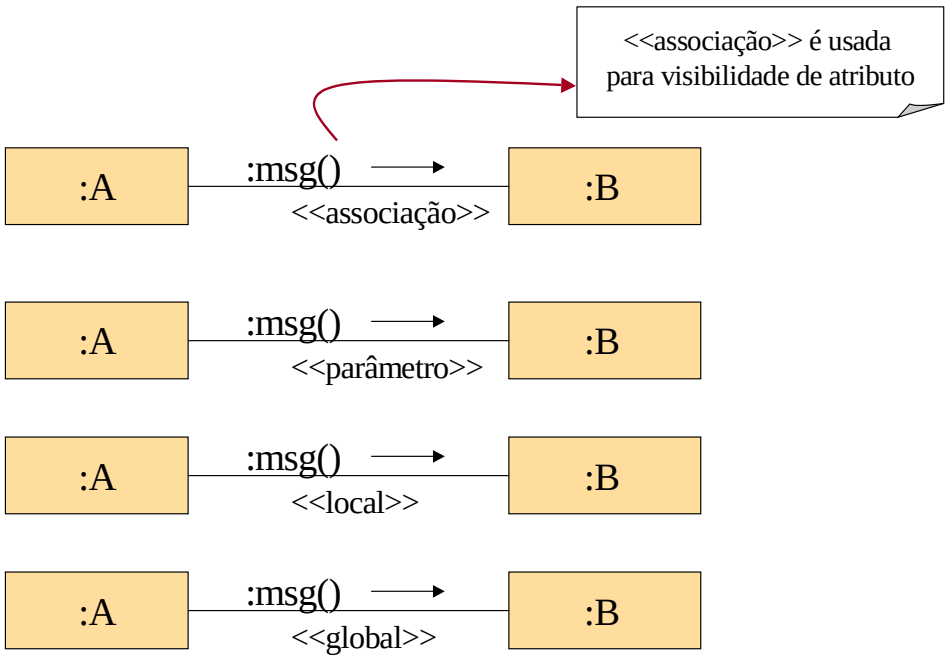


Figura 7.4 – Visibilidade na UML

7.3 – Diagrama de classes de Projeto

O Diagrama de Classes de Projeto apresenta as especificações para as classes de software e respectivas interfaces (por exemplo interfaces Java) a serem implementadas em uma

aplicação. As informações típicas contidas em um diagrama de classes são: classes, associações e atributos (como no modelo conceitual, substituindo-se os conceitos por classes); interfaces, com operações e constantes (trata-se da assinatura de todos os métodos que cada classe oferece, ou seja, os métodos que podem ser invocados por outras classes que tenham visibilidade para ela); métodos (ou seja, as linhas de código que implementam o comportamento esperado da classe); tipos dos atributos (cada atributo deve ter um tipo, que pode ser primitivo, composto, definido pelo usuário, referência a outro objeto ou coleções de objetos, etc); navegabilidade (indica a visibilidade por atributo de uma classe para outra); e dependências (indica visibilidade dos outros três tipos: por parâmetro, local ou global).

7.3.1 – Modelo Conceitual X Diagrama de Classes de Projeto

No Modelo Conceitual são feitas abstrações de conceitos ou objetos do mundo real, também chamados de classes conceituais. A idéia e mostrar os objetos que compõem o sistema, para dar uma visão geral, de forma esquemática e fácil de entender. Já o Diagrama de Classes de Projeto apresenta a definição de classes como componentes de software, ou seja, as classes que o compõem serão efetivamente implementadas usando uma linguagem de programação OO.

Na prática, o digrama de classes pode ser construído à medida que a fase de projeto avança, a partir dos diagramas de colaboração. Cada classe que aparece no diagrama de colaboração automaticamente é incluída no diagrama de classes de projeto. Neste livro preferimos fazer primeiro os diagramas de colaboração, para depois reunir as informações necessárias para o diagrama de classes de projeto.

7.3.2 – Como identificar as classes e atributos do Diagrama de Classes de Projeto

Todas as classes que aparecerem nos diagramas de colaboração devem ser incluídas no diagrama de classes de projeto, porque são classes que receberão e invocarão mensagens para satisfazer o comportamento esperado do sistema. Por exemplo, observando-se os diagramas de colaboração das Figuras 6.15 e 6.20 (Capítulo 6), chega-se às seguintes classes: Leitor, Empréstimo, LinhaDoEmpréstimo e CopiaDoLivro. Analisando o modelo conceitual da biblioteca (Figura 4.15 do Capítulo 4), pode-se deduzir quais são os atributos dessas classes. Pode-se acrescentar tipos de atributos também, se for o caso, para guiar a futura implementação. Se uma ferramenta CASE for utilizada para geração automática de código, os tipos detalhados são necessários. Por outro lado, se o diagrama for usado exclusivamente por desenvolvedores de software, o excesso de informação pode "poluir" o diagrama e dificultar seu entendimento. Além dos atributos identificados durante o projeto, podem ser incluídos outros atributos mais específicos ou para satisfazer restrições técnicas de projeto. Portanto, pode-se começar a esboçar o diagrama de classes de projeto, conforme ilustrado na Figura 7.5. É claro que, no caso do sistema de biblioteca, se outros diagramas de colaboração fossem observados, muitas outras classes seriam identificadas.

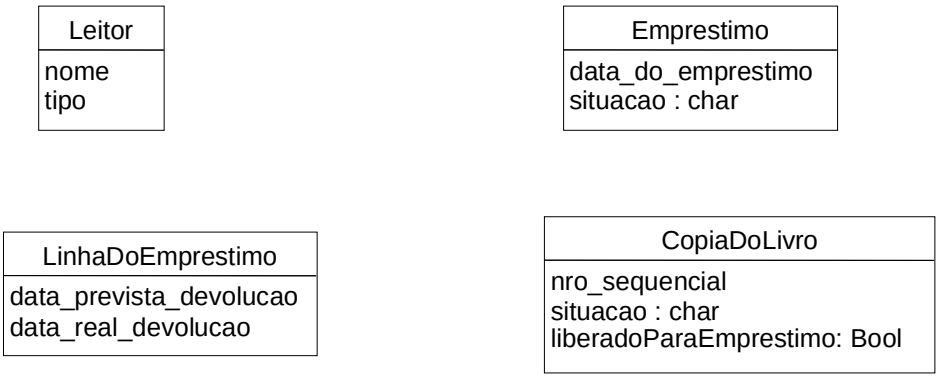


Figura 7.5 – Esboço inicial das classes da biblioteca

7.3.3 – Como identificar as associações e navegabilidade no Diagrama de Classes de Projeto

Para fazer as **associações** entre as classes identificadas, deve-se novamente observar todos os diagramas de colaboração construídos durante a fase de projeto. Todas as mensagens devem ser analisadas. Quando houver visibilidade por atributo de uma classe para outra, subentende-se que há uma associação entre essas classes. Para descobrir a multiplicidade, pode-se observar o modelo conceitual e, se ele não for suficiente, deve-se examinar a lógica dos métodos que são invocados de uma classe para a outra para descobrir qual é a multiplicidade da associação. Os nomes das associações também podem ser derivados a partir do modelo conceitual. Caso a associação não exista no modelo conceitual, deve-se dar um nome a ela durante o projeto.

Para exemplificar a identificação de associações, considere a Figura 7.6, em que se mostram as associações entre as classes identificadas anteriormente para o sistema de Biblioteca (Figura 7.5). Foram incluídas as associações de acordo apenas com os diagramas de colaboração das Figuras 6.15 e 6.20. As multiplicidades e os nomes das associações foram extraídos do Modelo Conceitual da Figura 4.15. A navegabilidade foi determinada da seguinte forma: se A envia mensagem para B, ou se A cria B, ou ainda se A precisa manter uma conexão com B, então nota-se um indício de que A deve ter visibilidade para B, ou seja, deve-se desenhar no diagrama de classes uma seta na associação entre A e B com a seta posicionada no lado de B. Deve-se verificar o envio de mensagens de objetos que possuem visibilidade por atributo, pois para os demais tipos de visibilidade não é necessário mostrar no diagrama de classes, para não torná-lo demasiadamente complexo sem necessidade. Em alguns casos, pode-se mostrar esses outros tipos de visibilidade por meio de dependência, como explicado na Seção 7.3.5.

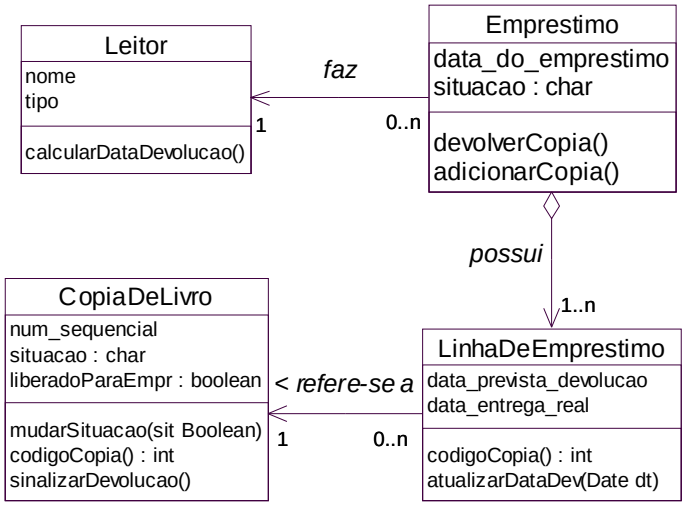


Figura 7.6– Diagrama de classes parcial para a biblioteca

Examinando-se o diagrama de colaboração da Figura 6.15, conclui-se que há navegabilidade entre as classes Emprestimo e Leitor (no sentido de Empréstimo para Leitor) e Emprestimo e LinhaDoEmprestimo, por haver visibilidade por atributo entre elas. Similarmente, examinando-se o diagrama de colaboração da Figura 6.20, nota-se que há navegabilidade entre Emprestimo e LinhaDoEmprestimo e entre LinhaEmprestimo e CopiaDoLivro. Essas navegabilidades são denotadas pelas setas dirigidas na Figura 7.6.

7.3.4 – Como identificar herança e agregação no Diagrama de Classes de Projeto

Associações de herança e agregação também se propagam para o diagrama de classes. Freqüentemente podem aparecer novas associações de herança durante o projeto, quando mais detalhes das classes passam a ser conhecidos e pode-se desejar abstraí-los para aumentar o reuso de código na fase seguinte. Por exemplo, na Biblioteca, percebe-se que tanto a reserva quanto o empréstimo tem vários atributos em comum, portanto pode-se decidir pela criação de uma classe que abstraia essas duas, por exemplo, uma classe denominada “Transação”, com os atributos comuns tanto a Empréstimo quanto a Reserva, fazendo com que essas duas últimas herdem da primeira, conforme ilustrado na Figura 7.7.

7.3.5 – Como identificar métodos no Diagrama de Classes de Projeto

Além de mostrar os atributos de cada classe, o diagrama de classes de projeto deve mostrar também os métodos que serão implementados em cada classe. A idéia é mostrar apenas a assinatura do método, ou seja, o nome do método, seus parâmetros de entrada e suas saídas. Linguagens de programação distintas podem ter sintaxes distintas para métodos. Portanto, recomenda-se que seja utilizada a sintaxe básica UML para métodos, que é a seguinte: *nomeMétodo(Par1, Par2, … Parn): Retorno*, onde Par1, Par2,... Parn são os parâmetros de entrada do método e Retorno é o tipo de dado ou objeto retornado pelo método. Na UML, os métodos são exibidos no terceiro compartimento do retângulo que representa a classe,

conforme ilustrado na Figura 7.6 (por exemplo, *calcularDataDevolucao* é um método da classe Leitor).

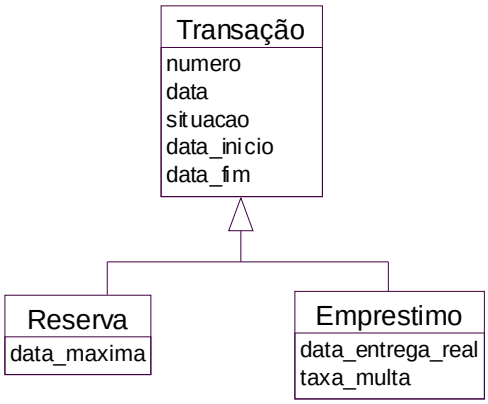


Figura 7.7– Herança no diagrama de classes

7.3.5.1 – Métodos que não se deve incluir no diagrama de classes

Mas como saber que métodos incluir em cada classe? Comecemos dizendo quais métodos não incluir nas classes. Em geral, os métodos de acesso, que são aqueles responsáveis por atribuir e recuperar valor de atributos da classe (como *setNome*, *getNome*, etc.), embora importantes, não devem ser incluídos para não sobrecarregar a classe com muitos métodos. Assim, considera-se que a classe deverá implementá-los, mas eles não são mostrados no diagrama de classes explicitamente.

Métodos referentes a mensagens enviadas a coleções de objetos também não devem ser mostrados no diagrama de classes, pois dependem da implementação. Por exemplo, o método *próximo()* na Figura 7.8 é enviado à coleção de linhas de empréstimo para obter a próxima linha a ser considerada durante a devolução da cópia. Não incluímos o método *próximo()* na classe LinhaDoEmprestimo e consideramos que ele será implementado pelo mecanismo escolhido para representar a coleção, seja ele uma lista, um arquivo, etc. Da mesma forma, não seriam mostrados métodos como buscar(), adicionar(), etc, enviados à coleções de objetos. O método *criar()* também não deve ser incluído no diagrama de classes de projeto, pois considera-se que a linguagem OO fornece o método criador de alguma forma. Por exemplo, em Java utiliza-se o construtor da classe e em Smalltalk a palavra chave *new*.

7.3.5.2 – Métodos a incluir no diagrama de classes

Passemos agora para os métodos que devem ser incluídos no diagrama de classes de projeto. Em primeiro lugar, as operações (identificadas por meio dos diagramas de seqüência do sistema, seção 5.1) devem ser incluídas nas classes controladoras, de acordo com o projeto OO realizado anteriormente, no qual foi feita a escolha do padrão Controlador mais adequado (seção 6.3.7). Por exemplo, na Figura 7.8, a operação *devolverCopia*, foi atribuída à classe controladora fachada Empréstimo, e portanto um método de mesmo nome é incluído na classe Empréstimo (ver Figura 7.6).

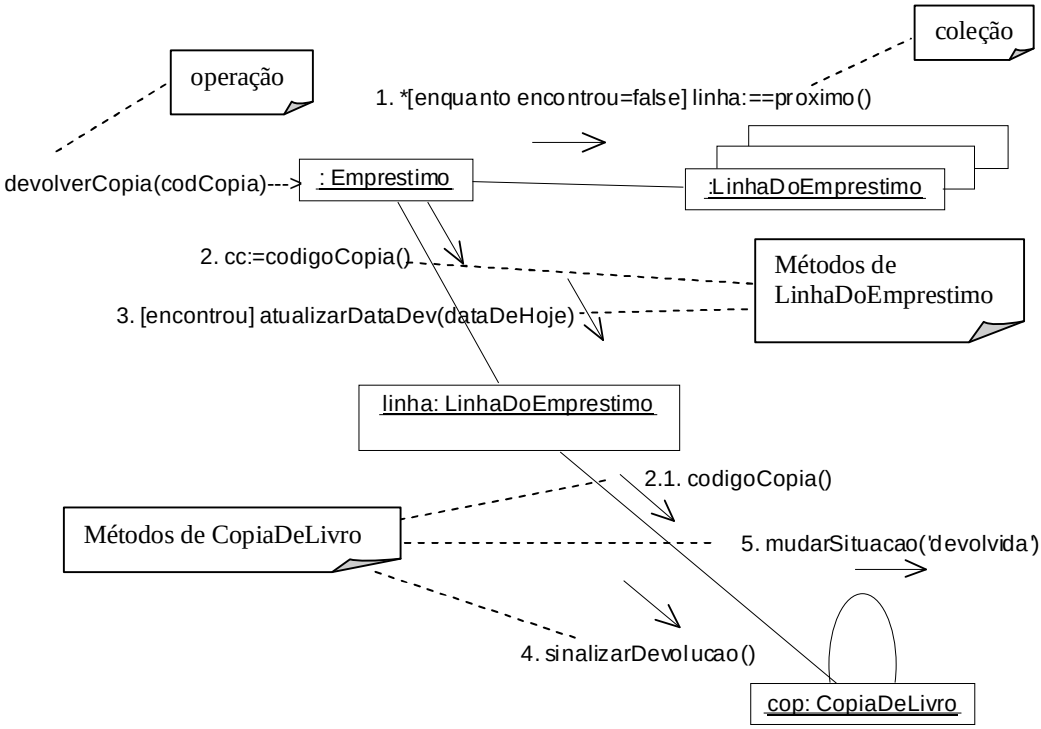


Figura 7.8– Identificação de métodos no diagrama de colaboração

A última regra é incluir os métodos nas classes que recebem a mensagem correspondente. Por exemplo, no diagrama de colaboração da Figura 7.8 pode-se deduzir a existência dos seguintes métodos de LinhaDoEmprestimo e CopiaDeLivro

- LinhaDoEmprestimo: codigoCopia(), atualizarDataDev() e
- CopiaDeLivro: mudarSituacao(), codigoCopia() e sinalizarDevolucao()

Todos os diagramas de colaboração produzidos durante o projeto OO devem ser analisados para identificar os métodos a serem incluídos no diagrama de classes de projeto do sistema. Para fins do exemplo da biblioteca, analisemos mais um diagrama de colaboração, mostrado na Figura 7.9. Por meio dele chegamos a mais dois métodos a serem incluídos no diagrama de classes: a operação *adicionarCopia()* é incluída como um método na classe controladora Emprestimo e o método *calculaDataDevolucão* é incluído na classe Leitor. O método *criar()* não é incluído, conforme explicado na seção 7.3.5.1. A Figura 7.6 mostra o diagrama de classes com os métodos já identificados.

7.3.6 – Dependência entre classes

Opcionalmente, pode-se incluir no diagrama de classes de projeto as dependências entre classes. Uma classe é dita dependente de outra se ela de alguma forma invoca métodos ou cria objetos dessa outra classe, mas não tem visibilidade por atributo para ela, ou seja, a classe tem visibilidade de outros tipos (por parâmetro, local ou global). A dependência é ilustrada em UML por meio de uma linha tracejada com uma seta no sentido da classe

dependente, ou seja, a ponta da seta fica na extremidade referente à classe da qual se depende.

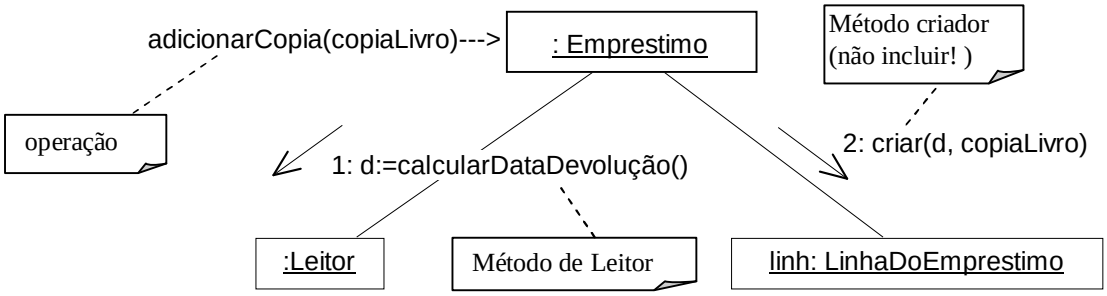


Figura 7.9– Identificação de métodos no diagrama de colaboração

Pode ser útil visualizar a dependência entre classes para, por exemplo, fazer manutenção no sistema, quando é preciso ter idéia do que uma certa mudança pode causar em outras classes. Sabendo-se que uma classe A depende de uma classe B, se algo for mudado na interface da classe B, pode ser que a classe A tenha que mudar a forma de invocar um certo método, por exemplo.

Como exemplo de dependência, a Figura 7.10 ilustra a dependência entre as classes CopiaDeLivro e Leitor. Imagine que o método sinalizarDevolucao precisasse regularizar a situação do leitor diante da devolução do livro. Como não tem visibilidade para ele, poderia receber como parâmetro o objeto *leitor*, e então poderia invocar algum de seus métodos para executar essa responsabilidade. Para que isso fosse cumprido, ao invocar o método atualizarDataDev de LinhaDeEmprestimo, o objeto de Empréstimo deveria passar *leitor* como parâmetro, para que este, por sua vez, pudesse passar para CopiaDeLivro.

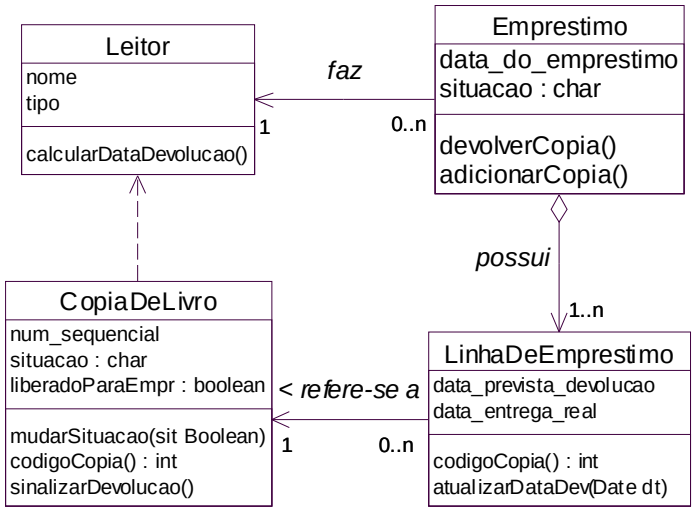


Figura 7.10– Diagrama de Classes com associação de dependência

7.3.7 – Mais detalhes sobre a notação para diagrama de classes

A Figura 7.11 mostra mais alguns conceitos sobre diagramas de classes que podem ser úteis/necessários durante o projeto OO. Um atributo pode ter seu valor inicial atribuído no diagrama (1). Se um atributo da classe for desejado, ao invés de um atributo dos objetos instanciados a partir da classe, sublinha-se o nome do atributo (2). Isso significa que o atributo pertencerá à classe e não a seus objetos individualmente. Pode ser interessante, em determinados projetos, ilustrar no diagrama de classes atributos que podem ser derivados ou calculados por outros meios, por exemplo, por chamada de métodos ou operações aritméticas. Isso é ilustrado colocando-se uma barra invertida antes do nome do atributo (3). Atributos derivados provavelmente não serão implementados na linguagem de programação e não serão armazenados na base de dados, mas pode ser útil durante o projeto saber que o atributo é inerente à classe.

Métodos abstratos (4) são denotados com itálico. Um método é abstrato se seu comportamento não existe na superclasse, mas é definido apenas em suas subclasses. Métodos públicos são aqueles visíveis a quaisquer classes que precisem invocá-los. São denotados por um sinal de + (5). Já os métodos privados são visíveis apenas na classe na qual estão definidos e são denotados por um hífen (6). Por fim, os métodos protegidos, denotados pelo sinal #, são aqueles visíveis na classe em que estão definidos, bem como em quaisquer subclasses dela (7).

Assim como para atributos, métodos também podem se referir à classe em si, sendo denotados em UML pelo sublinhado (8). Métodos da classe podem ser invocados diretamente à classe, ao invés de a instâncias da classe. Por exemplo, se *data1* é um objeto da classe Date de Smalltalk (Figura 7.12) contendo a data de hoje (25 de julho de 2005), pode-se invocar o método *data1.monthName*, que retornará a cadeia "julho". Já o método *nameOfMonth* pertence à classe Date, e deve ser invocado diretamente a ela, por exemplo *Date.nameOfMonth(7)*, que também retornará a cadeia "julho". Note que no segundo caso é necessário passar como parâmetro o número do mês para o qual se deseja saber o nome, enquanto no primeiro caso o próprio objeto *data1* contém uma data em particular, e deseja-se saber o nome do mês correspondente.

7.4 – Exercícios Propostos

7.4.1. Com base nos diagramas de colaboração fornecidos no Apêndice D e no Modelo Conceitual fornecido no Apêndice C para o Sistema Passe Livre, elaborar o Diagrama de Classes de Projeto, considerando apenas as classes que aparecem nos diagramas mencionados. Não se esqueça de incluir itens como: classes, associações, navegabilidade, multiplicidade, nome das associações, atributos (com tipos) e métodos (com tipo de parâmetros e retorno).

7.4.2. Aperfeiçoe o diagrama produzido no exercício 7.4.1., incluindo herança, agregação e dependência entre classes.

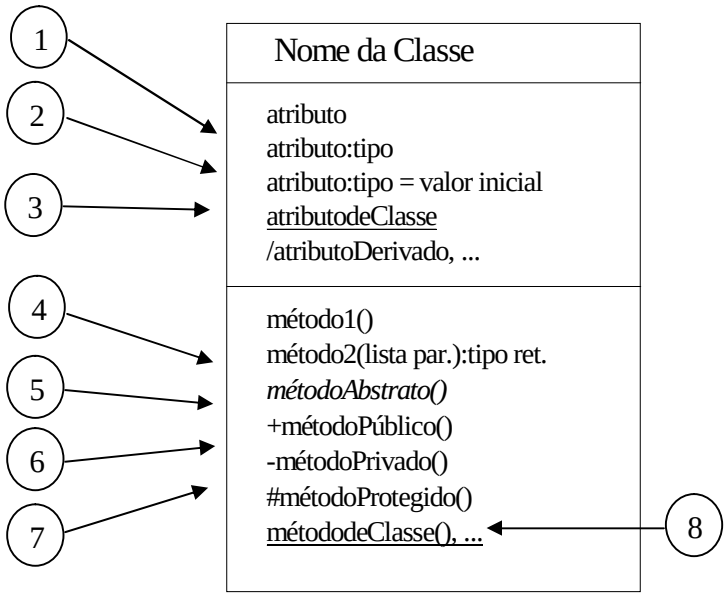


Figura 7.11 – Resumo da notação UML para Diagrama de Classes

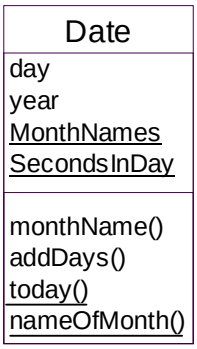


Figura 7.12 – Parte da classe Date de Smalltalk

7.5 – Exercícios complementares

7.5.1. Projete o diagrama de classes para o sistema de aluguel de fitas de vídeos, considerando o projeto OO feito nos exercícios do capítulo 6.

7.5.2. Projete o diagrama de classes do sistema de biblioteca universitária cujo projeto foi elaborado nos exercícios do capítulo 6.

7.5.3. Tente projetar o diagrama de classes para o sistema de reparo de buracos do exercício 3.10.9, para o qual não se prosseguiu para a fase de projeto, mas foi feita apenas a análise (casos de uso e modelo conceitual).

7.5.4. Considerando o exercício 7.5.3, relate as dificuldades em identificar as mensagens de cada classe sem ter feito o diagrama de colaboração.

7.5.5. Também considerando o exercício 7.5.3, como você determinaria a visibilidade entre classes sem fazer o projeto dos diagramas de colaboração?

7.5.6. Projete pelo menos dois diagramas de colaboração para o sistema de reparo de buracos do exercício 3.10.9, por exemplo para algumas das operações dos casos de uso “Entrar Ordem de Serviço de Buraco” e “Encerrar Ordem de Serviço” e identifique os métodos nele contidos. Desenhe agora o diagrama de classes e compare sua solução com a do exercício 7.5.3. Comente as dificuldades do projeto do diagrama de colaboração em relação a maior facilidade em identificar os métodos.

7.5.7. Tente projetar o diagrama de classes para o sistema ATM do exercício 3.10.11 para o qual não se prosseguiu para a fase de projeto, mas foi feita apenas a análise (casos de uso e modelo conceitual).

7.5.8. Considerando o exercício 7.5.7, relate as dificuldades em identificar as mensagens de cada classe sem ter feito o diagrama de colaboração.

7.5.9. Comparando as dificuldades do exercício 7.5.3 com as dificuldades do exercício 7.5.7, o que pode ter influenciado essa dificuldade? Discuta.

Capítulo 8 – Disciplina de Implementação – Como Mapear um Projeto OO para uma Linguagem de Programação OO

Os artefatos produzidos durante o Projeto OO (Capítulos 6 e 7) encerram a fase de elaboração do PU. Mais especificamente, foram produzidos diagramas de colaboração e diagramas de classes de projeto. Na próxima fase, de Construção, trata-se de implementar esses modelos utilizando uma linguagem de programação orientada a objetos. Dessa forma, será obtido um software executável, que poderá ser implantado e colocado em operação.

Assim, este capítulo fornece algumas diretrizes sobre como mapear o projeto OO em código, em particular na linguagem de programação Java. Vale ressaltar que muitas decisões terão que ser tomadas durante o projeto e a programação, e essas decisões moldarão a forma como a implementação será conduzida. Isso é discutido na seção a seguir. Este material não se propõe a fornecer detalhes de implementação em Java, mas apenas dar noções gerais que permitam assimilar os conhecimentos sobre o paradigma orientado a objetos em uma linguagem de programação concreta. Para aprofundar os conhecimentos sobre Java, recomenda-se a leitura de livros sobre programação, como por exemplo o livro Java – Como Programar [Deitel, 2000].

8.1 – O impacto das Decisões de Projeto

Os resultados obtidos no projeto são o ponto de partida para implementação de um software de qualidade, mas muito trabalho ainda tem que ser feito. A A/POO, seguindo o PU, produz modelos que refletem principalmente a camada de negócios do sistema final. Assim, outras camadas deverão ser implementadas para dar suporte a outros requisitos do sistema, por exemplo, requisitos não funcionais de segurança, distribuição, etc. Além disso, o mecanismo de persistência terá que ser providenciado, caso a base de dados escolhida não seja orientada a objetos. A questão da persistência é tratada na Seção 9.2.

Várias decisões tomadas durante o projeto da arquitetura do sistema terão impacto, seja positivo ou negativo, na forma como a implementação será conduzida. Por exemplo, se for escolhido um modelo em três camadas: persistência, negócios e interface com o usuário, o projeto OO descrito nos capítulos 6 e 7 refletirá basicamente as classes pertencentes à camada de negócios, ou seja, o projeto das demais camadas terá que ser feito separadamente. Assim, outros artefatos de projeto detalhado tem que ser produzidos, tais como o esquema da base de dados e o projeto da interface gráfica com o usuário.

Normalmente a camada de persistência pode ser reusada, por exemplo, pois existem frameworks tais como o Struts[1] [Goodwill, 2002] e o Hibernate[2] [Bauer, 2004] que provêem essa funcionalidade. Muitas alterações podem ocorrer para acomodar restrições de

[1] http://struts.apache.org/
[2] http://www.hibernate.org/

implementação e problemas podem surgir e precisam ser solucionados. Portanto, é necessário estar preparado para possíveis mudanças e desvios de projeto.

Outro ponto a considerar é que, normalmente, não se chega à fase de implementação totalmente sem código já implementado. É bastante comum criar alguns trechos de código ou protótipos durante a fase de elaboração, e esse código pode ser utilizado como ponto de partida na construção do sistema concreto. Além disso, conforme dito no Capítulo 7, se ferramentas CASE tiverem sido utilizadas adequadamente na fase de projeto, uma parte significativa das classes pode ser automaticamente gerada por tais ferramentas, poupando trabalho braçal de criação das classes. Exemplos de código gerado automaticamente pelas ferramentas incluem a declaração das classes, com seus atributos e métodos construtores (por exemplo o *criar*), métodos de atribuição de valor a um atributo (*set*) e recuperação de valor de atributo (*get*), entre outros.

8.2 – Declaração das Classes

O Diagrama de Classes resultante da fase de projeto OO indica todas as classes que devem ser implementadas na camada de negócios do sistema. Conforme foi visto no Padrão Controlador (Seção 6.3.7), uma classe é escolhida para acolher as operações que realizam o comportamento dos casos de uso do sistema. No exemplo da Biblioteca, segundo a Figura 6.2.5(a), por exemplo, a classe Biblioteca foi escolhida como controladora fachada para abrigar as operações *iniciarEmprestimo*, *emprestarCopia* e *encerrarEmprestimo*. Portanto deve-se criar a classe Biblioteca, sem atributos específicos, mas com a declaração dos métodos referentes a essas operações.

Portanto, a primeira recomendação é criar uma classe para cada classe controladora. Em Java, a classe é armazenada em um arquivo com extensão **.java**, como o exemplificado[3] na Figura 8.1 para a classe Biblioteca. Um nome de pacote (*package*) é utilizado para identificar o sistema implementado (neste caso o pacote chama-se *bibliot*). Pode-se importar classes necessárias para o funcionamento do programa (*import*). Comentários em Java são delimitados por /* e */.

Toda classe em Java deve implementar um método construtor (denominado **criar** nos capítulos anteriores), que possui o mesmo nome que a própria classe. Por exemplo, o método para criar novos objetos Leitor é denominado Leitor() e recebe como parâmetros todos os dados necessários para criar o objeto relativo ao novo leitor e atribuir valor inicial aos seus atributos. Contudo, pode haver diferentes implementações de construtores, diferenciadas apenas pelo número de parâmetros. Por exemplo, a Figura 8.2 mostra duas possibilidades de método construtor para a classe Leitor. No primeiro, o construtor é invocado com apenas um parâmetro e o resultado é o seguinte: um novo objeto da classe Leitor é retornado, o valor passado como parâmetro é atribuído ao atributo *nome* desse novo leitor e o valor 1 é atribuído ao atributo *tipo* desse novo leitor (esse é o valor default, ou seja, considera-se que o leitor é um aluno de graduação). No segundo construtor, o

[3] Os códigos exibidos nas figuras deste capítulo não são compiláveis, mas apenas para efeitos de ilustração. No Capítulo 12 pode-se encontrar referências para um sítio da Web contendo código-fonte do Sistema Passe Livre.

método recebe dois parâmetros, utilizados para iniciar os valores de seus dois atributos (nome e tipo, respectivamente). Métodos *set* e *get* devem ser disponibilizados para cada atributo da classe.

"Biblioteca.java"

```
package bibliot;
import java.util.Iterator;
import java.util.Vector;
/* Esta classe controla as operações referentes ao empréstimo de livros
da biblioteca. */
public class Biblioteca
{
  public void iniciarEmprestimo();
  public  int emprestarLivro(int codCopia, Date dataDevolucao);
  public void encerrarEmprestimo();
}
...
```

Figura 8.1 – Exemplo de parte da classe "Biblioteca" em Java

"Leitor.java"

```
package bibliot;
/* Esta classe representa os leitores da biblioteca. */
public class Leitor
{
/* atributos */
private String nome;
private int tipo; /* tipo=1 (aluno grad) = default
                     tipo=2 (aluno pos-grad)
                     tipo=3 (professor)                */
/* métodos de Leitor */
  public void Leitor(String umNome);
  public void Leitor(String umNome, int umTipo);
  public void setNome(String umNome);
  public String getNome();
  public void setTipo(int umTipo);
  public int getTipo();
/*  construtor com apenas um parametro */
  public Leitor(String umNome)
      {
         this.nome = umNome;
         this.tipo = 1;
      }
/* construtor com dois parâmetros */
  public Leitor(String umNome, int umTipo)
      {
         this.nome = umNome;
         this.tipo = umTipo;
      }
  public getNome()
      {
         return this.nome;
      }
  ...
}
```

Figura 8.2 – Exemplo de parte da classe "Leitor" em Java

8.3 – Atributos referenciais

Conforme visto no Capítulo 7, em um diagrama de classes representamos explicitamente todos os atributos de cada classe, por exemplo, *nome* e *tipo* na Figura 8.2 são atributos explícitos da classe Leitor, que são do String e Inteiro, respectivamente. Um atributo referencial é um atributo que referencia um outro objeto complexo e não um tipo primitivo (tal como uma string, por exemplo). Os atributos referenciais são sugeridos pelas associações e pela navegabilidade em um diagrama de classes e estão normalmente implícitos (ao invés de explícitos como os demais atributos). Por exemplo, no diagrama de classes da Figura 7.10 existe uma associação entre Empréstimo e Leitor, o que implica em uma referência a Leitor na classe Empréstimo. Assim, na declaração da classe Empréstimo deve haver um atributo referencial correspondente a essa associação, conforme ilustrado na Figura 8.3 (linha anotada com a observação número 1). Da mesma forma, há na Figura 7.10 uma referência implícita de Empréstimo para as respectivas linhas de empréstimo e, portanto, deve-se fazer referência a elas na declaração de Empréstimo. Para tal, deve-se escolher alguma estrutura de dados em Java para implementar esta referência. Na Figura 8.3 mostra-se como isso poderia ser feito usando a classe List de Java (linha anotada com a observação número 2).

"Emprestimo.java"

```
package bibliot;

/* Esta classe representa os emprestimos feitos na biblioteca. */

public class Emprestimo
{
/* atributos */
private Date data_do_emprestimo;
private Char[] situacao;
private Leitor leitor;    /*  observação número 1 */
private List linhas = new ArrayList();/*  observação número 2 */

...

/* métodos de Emprestimo */
  public void adicionarCopia(int codCopia);
  public void devolverCopia(Date dataDev, int codCopia);
  public Emprestimo(Date umaDataDev, Leitor umLeitor);

  public Emprestimo(Date umaDataDev, Leitor umLeitor)
      {
         this.data_do_emprestimo = new Date();;
         this.situacao = 'E';
         this.leitor = umLeitor;
      }
  ...
}
```

Figura 8.3 – Exemplo de atributo referencial para a classe "Emprestimo"

Conforme visto na Seção 4.2, um recurso importante de projeto é o uso de nome de papel para identificar o papel da classe na associação e fornecer, conseqüentemente, algum contexto semântico sobre a sua natureza. Se houver um nome de papel no diagrama de classes, recomenda-se que ele seja utilizado como base para o nome do atributo referencial durante a geração de código, conforme ilustrado na Figura 8.4.

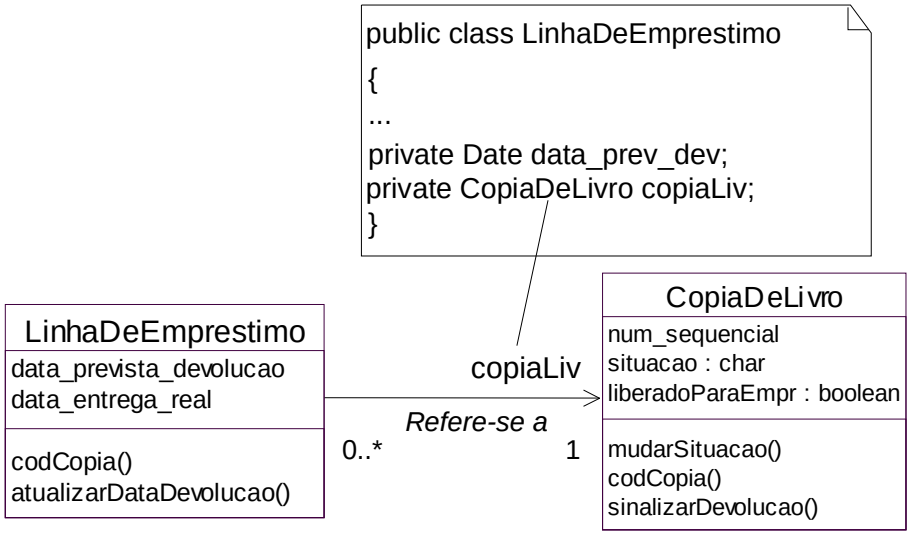


Figura 8.4 – Utilização de nome do papel como nome do atributo referencial

8.4 – Herança

Considere que o Leitor tenha um atributo em comum com o Atendente da biblioteca, digamos o nome. Neste caso podemos utilizar a herança para reusar o comportamento comum entre as duas classes, por exemplo criando uma superclasse Usuário que será usada como base para criar as subclasses Leitor e Atendente, conforme ilustrado na Figura 8.5. Em Java, a herança é conseguida com o uso da palavra chave *extends*, conforme ilustrado na Figura 8.6. Com isso, a subclasse herda todos os atributos e métodos da superclasse, podendo sobrepô-los caso necessário. No caso de Leitor, a subclasse deve definir métodos *set* e *get* para o novo atributo (*tipo*), e deve sobrepor o método construtor, já que o construtor da superclasse não atribui valor inicial ao novo atributo. Para chamar o comportamento da superclasse, utiliza-se o método *super*(), passando como parâmetros os atributos do construtor, que nesse caso é apenas o nome do leitor.

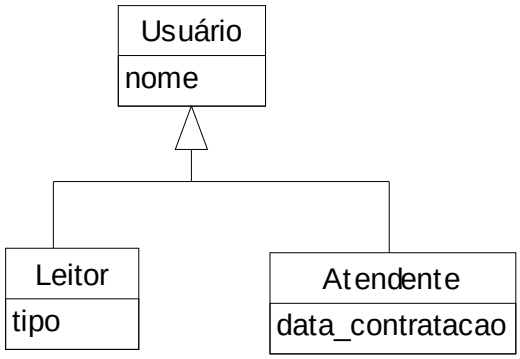


Figura 8.5 – Uso de Herança

"Leitor.java"

```
package bibliot;

/**
* A classe Leitor herda as propriedades da classe Usuario.
*/
public class Leitor extends Usuario
{
      private int tipo;  /* atributo da subclasse */
      /* O construtor da classe Leitor é feito novamente e invoca o
         construtor da superclasse.  */
      public Leitor(String nome, int tipo)
      {
        super(nome);
        this.cargo = cargo;
      }
      /* métodos set e get dos atributos da subclasse*/
      public getTipo()
      {
         return this.tipo;
      }
      public setTipo(int umTipo)
      {
         this.tipo = umTipo;
      }
```

Figura 8.6 – Herança em Java

8.5 – Criação de métodos a partir dos diagramas de colaboração

A seqüência de mensagens de um diagrama de colaboração é traduzida para uma série de comandos de programação na definição do método. Por exemplo, considere o diagrama de colaboração para a operação *devolverCopia* da Figura 8.7. A mensagem *devolverCopia* é enviada a Leitor, pelo uso do padrão Controlador Fachada (ver Seção 6.3.7). Isso implica que um método *devolverCopia()* deve ser definido em Leitor, por exemplo "public void devolverCopia (int codCopia)". O parâmetro codCopia informa o código da cópia que está sendo devolvida. Analisando o diagrama, verifica-se que a mensagem *devolverCopia()* é

delegada a cada empréstimo feito por leitor, para encontrar o empréstimo pendente referente à cópia que está sendo devolvida no momento. Isso implica que o método *devolverCopia()* é definido também em Emprestimo, embora com uma assinatura diferente, pois deve retornar um valor booleano (verdadeiro ou falso) para indicar se encontrou o empréstimo procurado. A assinatura do método, nesse caso, é public boolean devolverCopia (int codCopia).

Embora possuam o mesmo nome, o conteúdo desses dois métodos pode ser bem diferente um do outro. De fato, o método de Leitor deve percorrer cada um dos empréstimos realizados por um leitor até encontrar o empréstimo referente à copia do livro devolvido. Já o método de Empréstimo deve percorrer as linhas de empréstimo e para cada uma delas verificar se ela se refere ao livro recebido como parâmetro. Portanto, embora o nome seja o mesmo e o intuito do método seja semelhante, o conteúdo, refletido nas linhas de código do método, pode ser bem diferente. A Figura 8.8 ilustra a diferença entre esses dois métodos. Deve-se abrir um parêntese aqui para explicar resumidamente aos novatos o funcionamento do padrão Iterador, explicado no Capítulo 9. Sempre que uma coleção precisar ser percorrida, pode-se declarar um Iterador para ela (*Iterator*) e utilizar os métodos desse iterador sempre que quiser saber quem é o próximo elemento da coleçao (*next*) e se a coleção já chegou ao fim (*hasNext*).

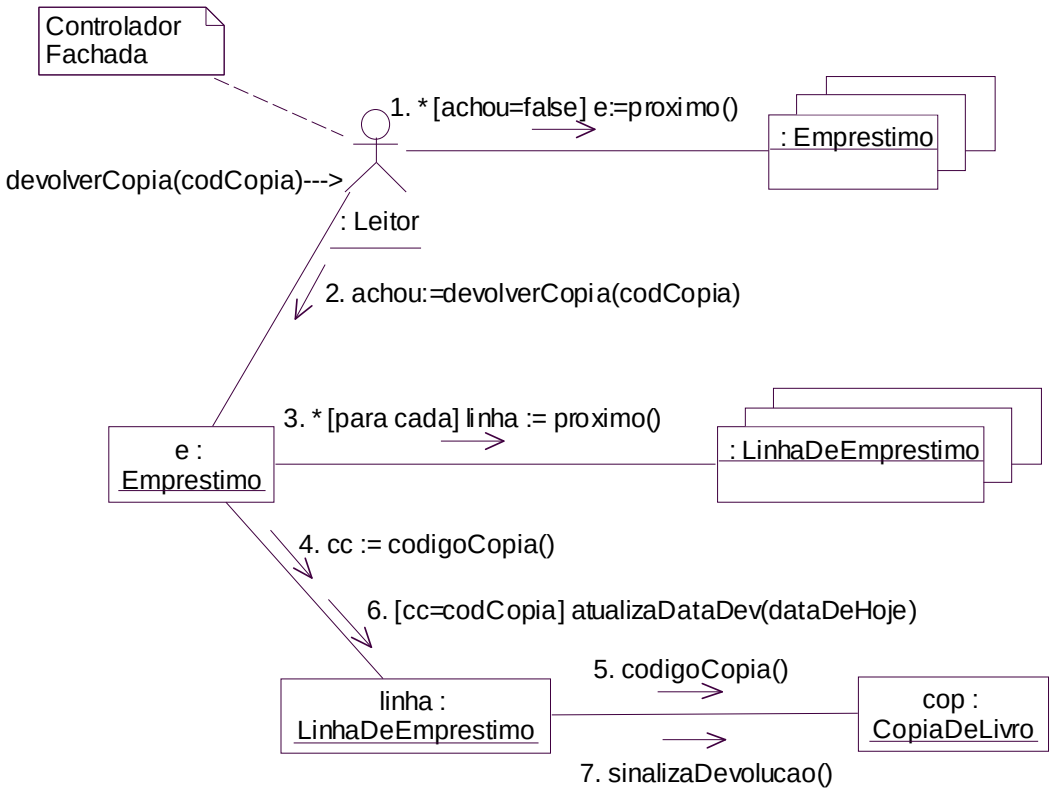


Figura 8.7 – Diagrama de colaboração para a operação devolverCopia()

Classe Leitor	Classe Empréstimo

public class Leitor
{
  private String nome;
  private Char[] tipo;
  private Boolean achou=false;
  private List emprestimos = new ArrayList();
 	
  public void devolverCopia(int codCopia)
  {
  Iterator i = emprestimos.iterator();
  while (i.hasNext()) && (!achou) {
  Emprestimo e = (Emprestimo) i.next();
  achou=e.devolverCopia(codCopia)}
  }
}
	public class Emprestimo
	{
	  private Date data_de_emprestimo;
	  private Char[] situacao;
	  private int cc=0;
	  private List linhas = new ArrayList();
	
	public boolean devolverCopia(int codCopia)
	  {
	  Iterator i = linhas.iterator();
	  Date dataDeHoje = new Date();
	  while (i.hasNext()) {
	  LinhaDeEmprestimo linha =
	  LinhaDeEmprestimo) i.next();
	  cc=linha.codigoCopia();
	  if (cc==codCopia)
	  {linha.atualizaDataDev(dataDeHoje);
	  return true;}
	  }
	  return false;
	  }
	}

Figura 8.8 – Métodos de mesmo nome implementados em classes diferentes

8.6 – Projeto da Interface Gráfica com o Usuário

Nas seções anteriores viu-se o mapeamento do projeto da camada de negócios do sistema para código em Java. No entanto, para que a aplicação seja executada é necessário que ela tenha uma interface gráfica com o usuário, por exemplo, usando um formulário como o da Figura 8.9 (a). Este formulário pode ficar armazenado em uma classe da camada GUI, que no exemplo da Figura 8.9 (b) é chamada JanelaEmpr. O comportamento do sistema quando o atendente pressiona o botão Emprestar da Figura 8.9 (a) é invocar o método *emprestarLivro,* que dispara a ação executada na classe JanelaEmpr e delega a responsabilidade de atender a essa ação para uma classe do domínio, neste caso por meio da chamada do método *emprestarLivro* da classe controladora Biblioteca que, por sua vez, invoca o método da classe Empréstimo (Figura 8.9 (c)).

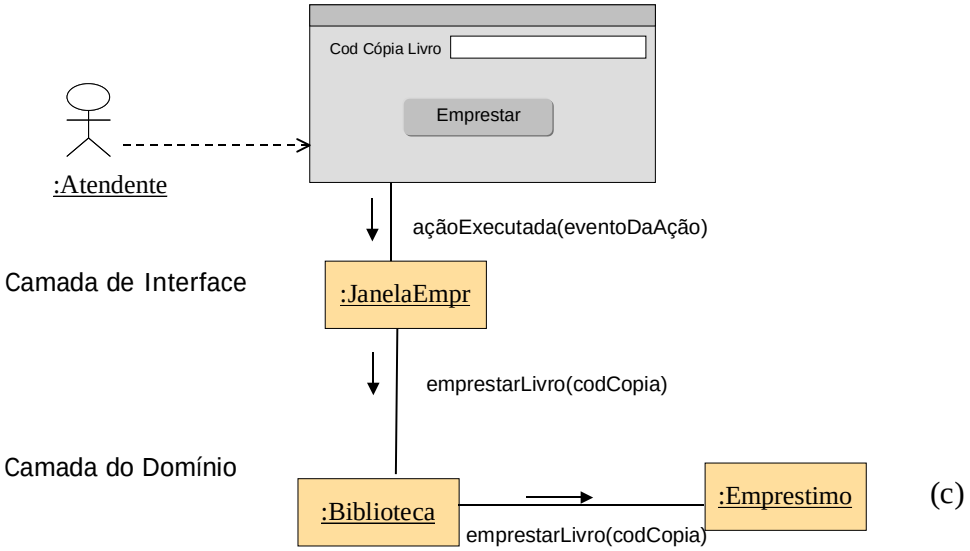


Figura 8.9 – Comunicação entre as camadas do sistema

Projetos feitos desta maneira, ou seja, com clara separação entre a camada GUI e a camada de negócios, são mais reusáveis e mais fáceis de manter e evoluir, pois a camada GUI não precisa ter conhecimento das regras de negócio e pode ser facilmente entendida, substituída por outra ou alterada.

Outra boa sugestão para implementação do sistema é separar também a camada de persistência de dados, ou seja, o armazenamento dos objetos em base de dados deve ser feito em outra camada de software, para facilitar futuras evoluções. Por exemplo, ao clicar no botão Emprestar, o método responsável por salvar os dados é delegado a uma camada específica do software, que se encarrega de armazenar os dados, seja em base de dados relacional, orientada a objetos ou arquivo XML, entre outras.

No Capítulo 9 são apresentados alguns padrões de projeto, por meio dos quais apresentam-se soluções alternativas para tratar de problemas que ocorrem ao implementar um sistema orientado a objetos.

8.7 – Exercícios Propostos

8.7.1. Com base no Diagrama de Classes de Projeto do Sistema Passe Livre, elaborado no exercício 7.4.1, descreva programas em Java para declaração das classes e suas interfaces.

8.7.2. Implemente o corpo de alguns métodos do sistema Passe Livre e teste-os.

8.7.3. Projete o esquema de uma base de dados relacional para armazenar os objetos de acordo com o projeto obtido no exercício 7.4.1.

8.7.4. Considerando o caso de uso no formato completo concreto do exercício 3.9.6, para o caso de uso “Comprar Gizmo”, faça uma relação dos métodos e operações que seriam invocados em resposta aos eventos de cada um dos widgets da GUI projetada.

8.8 – Exercícios complementares

8.8.1. Descreva as classes do Sistema de locação de fitas de acordo com o projeto obtido no exercício 7.5.1.

8.8.2. Esboce o algoritmo de alguns métodos do sistema do exercício 7.5.1, por exemplo, os métodos *devolverFita*, *calcularDataDevolucao*, *calcularPrecoLocacao*, etc.

8.8.3. Descreva as classes do Sistema de Biblioteca Universitária de acordo com o projeto resultante do exercício 7.5.2.

8.8.4. Faça o esboço do método que implementa em Java algumas das operações do sistema do exercício 7.5.3. Por exemplo, considerando que a lógica é similar ao sistema de Biblioteca visto no exemplo, descreva o método para a operação *devolverCopia* da Figura 6.20 e o método *adicionarCopia* da Figura 6.15.

8.8.5. Tente programar do zero, sem projeto anterior, algumas classes do sistema de locação de carros que foi modelado no exercício 4.8.6. Analise o resultado e comente o que poderia ser melhorado em termos de coesão e acoplamento se o projeto tivesse sido feito conforme recomenda o PU.

8.8.6. Quais as dificuldades em programar usando uma linguagem OO sem realizar a análise/projeto OO?

Capítulo 9 – Uso de Padrões para o Projeto de Software

9.1 – Conceitos básicos sobre padrões

Padrões de software (*software patterns*) documentam soluções para problemas que ocorrem freqüentemente durante o desenvolvimento de software, aumentando a produtividade e qualidade do software. Assim, os padrões permitem que desenvolvedores menos experientes reusem soluções já consagradas por outros desenvolvedores. O reúso é feito quando um desenvolvedor se depara com um problema para o qual existe um padrão documentado e, ao invés de elaborar a solução para o problema, ele simplesmente reutiliza a solução oferecida pelo padrão.

A idéia de padrões de software foi inspirada nos padrões de Alexander [1977, 1979], que são padrões para a área de arquitetura (construção de casas, bairros, cidades, etc.). Em software, Beck e Cunningham [1987] foram os primeiros a propor padrões de software. Em seguida, vários outros trabalhos foram surgindo [Coplien, 1992, 1995; Coad, 1992, 1995], mas o que alavancou a área de padrões foi o lançamento do livro de Gamma e outros [1995], que propuseram vinte e três padrões de projeto úteis para resolver problemas comuns encontrados durante o projeto orientado a objetos. Muita informação sobre a história dos padrões de software pode ser obtida no sítio do grupo Hillside (http://www.hillside.net), que é uma corporação sem fins lucrativos dedicada a melhorar a comunicação humana sobre computação pelo encorajamento que faz para que as pessoas codifiquem, por meio de padrões, seus conhecimentos comuns e práticas de projeto.

Os padrões são considerados uma forma poderosa de melhorar o reúso, não somente de código mas também de análise, projeto, processo, arquitetura, etc. A idéia é que, se um especialista documenta, por meio de um padrão, a solução para um problema que ocorre com freqüência em um determinado contexto, outro desenvolvedor menos experiente na resolução daquele problema pode imediatamente aplicar a solução, possivelmente após algum tipo de adaptação ou personalização para o caso específico. Assim, economizaremos tempo por não precisarmos inventar uma nova solução, mas apenas reusar o que já é conhecido. Isso se aplica a problemas em quaisquer níveis de abstração – mais especificamente para o desenvolvimento de software, podemos pensar em pares "problema/solução" [Buschmann 96].

Assim, existem documentados na literatura padrões em diversos níveis de abstração, desde padrões de processo, padrões arquiteturais, padrões de análise, padrões de projeto e padrões de programação, entre outros. Neste livro, focaremos os padrões de projeto, por serem bastante úteis para resolver os problemas encontrados no projeto OO.

Uma outra vantagem da utilização de padrões é que, em sistemas complexos, o problema maior pode ser decomposto em problemas menores, sendo que para cada um desses problemas menores um padrão é utilizado. Assim, os padrões podem ser considerados como peças que são encaixadas para compor o sistema final, atacando a complexidade.

Como um padrão documenta uma solução para um problema, devemos escolher o melhor formato para que essa documentação seja melhor compreendida pelo usuário do padrão. Existem várias propostas de formato para documentar os padrões, como a de Alexander [1979], a canônica [Coplien, 1995] e a de Gamma [1995]. Cada um dos formatos possui elementos essenciais e outros opcionais. Dependendo do tipo de padrão, pode ser que um formato se encaixe melhor. Os elementos essenciais em um padrão são: nome, contexto, forças, problema e solução. Alguns dos elementos opcionais são: conseqüências, usos conhecidos, contexto resultante, exemplo, amostra de código e padrões relacionados. Neste livro, consideraremos o formato proposto por Gamma, explicado na seção a seguir, já que focaremos os padrões de projeto.

9.2 – Padrões de Projeto

Padrões de projeto são específicos para documentar soluções relacionadas a problemas de projeto, seja ou não de software orientado a objetos. Em particular, os padrões de projeto propostos por Gamma aplicam-se melhor a projetos OO, embora vários deles possam ser pensados também para sistemas não OO. Projetistas familiarizados com certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los [Gamma 95]. Um padrão é um conjunto de informações instrutivas que possui um nome e que capta a estrutura essencial e o raciocínio de uma família de soluções comprovadamente bem sucedidas para um problema repetido que ocorre sob um determinado contexto e um conjunto de repercussões [Appleton 97].

O formato proposto por Gamma [1995] para documentar os padrões de projeto contém os seguintes elementos:

- *Nome*: descreve um nome para o padrão, por meio do qual ele será referenciado e, portanto, passará a fazer parte do vocabulário do desenvolvedor;
- *Classificação*: Gamma classifica os padrões segundo duas perspectivas: propósito e escopo, que não serão detalhadas neste livro por não serem essenciais para a compreensão dos objetivos almejados;
- *Intenção*: descreve o que o padrão faz, caracterizando o problema e a solução;
- *Também Conhecido Como*: apresenta outros possíveis nomes conhecidos para o padrão;
- *Motivação*: descreve um cenário que ilustra um problema de projeto e como estruturas de classes e objetos do padrão resolvem o problema;
- *Aplicabilidade*: descreve em que situações o padrão pode ser aplicado, exemplos de projetos “pobres” para os quais o padrão pode estar dirigido e como reconhecer essas situações;
- *Estrutura*: descreve a estrutura básica da solução por meio de diagramas de classes;
- *Participantes*: descreve as classes, objetos ou componentes participantes do padrão e suas responsabilidades;
- *Colaborações*: descreve como os participantes colaboram entre si para cuidar de suas responsabilidades;
- *Implementação*: descreve técnicas, dicas ou questões específicas de linguagem necessárias para implementar o padrão;

- *Código Exemplo*: descreve exemplos concretos de aplicações do padrão. Pode apresentar códigos-fonte, diagramas, figuras, etc.;
- *Conseqüências*: descreve os efeitos de uso do padrão. Define o estado ou configuração do sistema depois da aplicação do padrão, incluindo as conseqüências boas ou ruins;
- *Usos Conhecidos*: descreve exemplos da aplicação do padrão em sistemas reais. Recomenda-se a inclusão de pelo menos três usos conhecidos;
- *Padrões Relacionados*: descreve como o padrão está relacionado com outros padrões que se referem ao mesmo problema. Pode referenciar outros padrões usados em conjunto ou outras soluções para o problema, ou ainda, variações do padrão.

Nas seções 9.3 a 9.7 são apresentados 5 dos 23 padrões de projeto de Gamma [1995]. Como não podemos reproduzir aqui os padrões na íntegra, fornecemos apenas os principais elementos, que são necessários para compreender o padrão. Caso o leitor queira realmente aplicar o padrão, deve ler o texto integral do padrão documentado no livro.

9.3 – Padrão Composto

O Quadro 9.1 resume o padrão Composto (várias partes foram omitidas e podem ser encontradas no livro de Gamma e outros [1995]. Possíveis aplicações que podem utilizar este padrão em seu projeto são aquelas em que há uma agregação (um composto e suas partes constituintes) recursiva, ou seja, há um composto que possui várias partes, e cada uma dessas partes também pode ser composta de outras partes.

Consideremos os papéis desempenhados pelos usuários do Passe Livre. Em nosso modelo conceitual (Figura 11.8) previmos três papéis diferentes: Analista Financeiro, Atendente e Proprietário. O que aconteceria se tivéssemos um atendente que fosse proprietário de um veículo? Uma possível solução seria incluir dois objetos para representar esse mesmo indivíduo, um da classe Atendente e outro da classe Proprietário. Note que há uma certa recursão nos papéis desempenhados pelo usuário, já que um papel pode englobar outros papéis, e assim por diante, com diversos níveis de recursão. Portanto, o padrão Composto poderia ser utilizado para garantir que as operações invocadas em um papel composto sejam as mesmas invocadas em um papel concreto. A Figura 9.1 ilustra como ficaria o projeto do Sistema Passe Livre caso este padrão fosse utilizado.

Como contra-exemplo, no Sistema Passe Livre, a Área de Pedágio é composta de diversas partes: cancela, semáforo, sensores, etc. Mas cada uma das partes já é um elemento final, não havendo recursão. Portanto o padrão Composto não se aplica.

Quadro 9.1 – Padrão Composto

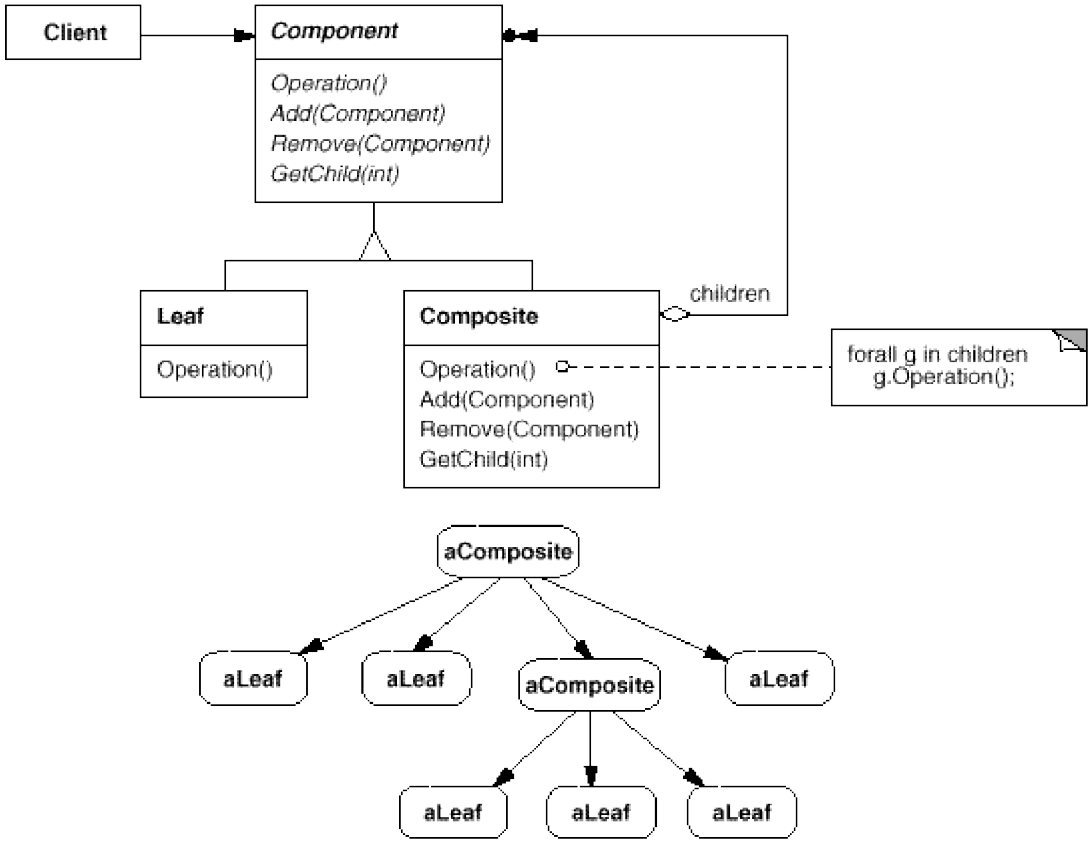
Nome: Composto (*Composite* em inglês)

Intenção: compõe objetos em estruturas de árvore para representar hierarquias todo-parte. O padrão Composto deixa o cliente tratar objetos individuais e composição de objetos uniformemente.

Motivação: Editores gráficos permitem aos usuários construir diagramas complexos, agrupando componentes simples. Uma implementação simples para isso seria definir uma classe para primitivas gráficas tais como Texto, Linhas e outras classes que agem como depósitos (contêineres) para essas primitivas. O problema é que o código que usa essas classes deve tratar primitivas e objetos do contêiner diferentemente, tornando a aplicação mais complexa. O padrão Composto cria uma classe abstrata que representa primitivas e seus contêineres.

Aplicabilidade: Use o padrão composto para: representar hierarquias de objetos todo-parte; permitir aos usuários ignorar a diferença entre composições de objetos e objetos individuais (todos os objetos na estrutura são tratados uniformemente).

Estrutura:



Participantes:
Componente - declara a interface para os objetos na composição; implementa o comportamento padrão para a interface comum de todas as classes, quando apropriado; declara uma interface para acessar e gerenciar os componentes filho; define uma interface para acessar o pai de um componente na estrutura recursiva, implementado-o se for apropriado.
Folha - representa objetos "folha" na composição; uma folha não tem filhos; define o comportamento para objetos primitivos na composição.
Composto - define o comportamento para componentes que têm filhos; armazena componentes filho; implementa operações relacionadas aos filhos na interface Componente
Cliente - manipula objetos na composição por meio da interface Componente

Colaborações: *Clientes* usam a interface *Componente* para interagir com objetos na estrutura composta. Se o receptor é uma *folha* então o pedido é manipulado diretamente. Se o receptor é um *Composto* então os pedidos são enviados para seus componentes filhos.

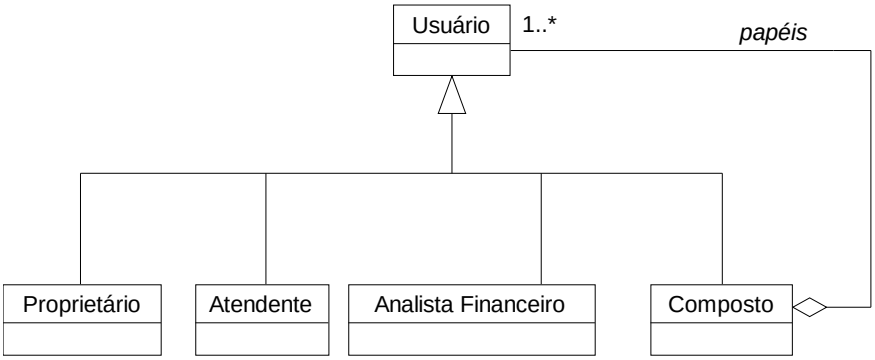


Figura 9.1 – Exemplo do padrão Composto no Sistema Passe Livre

9.4 – Padrão Estado

No quadro 9.2 faz-se um resumo do padrão Estado, que pode ser aplicado quando um objeto tiver comportamentos associados ao seu estado atual. No Sistema Passe Livre, várias situações são propícias à aplicação do padrão Estado, como para modelar os objetos Cancela, Sensor do Gizmo, Sensor de Saída, Veículo, Registro de Uso, etc. Na Figura 9.2, ilustra-se o uso desse padrão para modelar a classe Registro de Uso, que pode estar em um dos seguintes estados: a receber (quando o registro de uso acabou de ser criado e ainda não foi realizado o fechamento mensal); em cobrança (intervalo de tempo em que o fechamento foi realizado, mas o banco ainda não debitou o valor da conta do proprietário); pago (o banco realizou o débito com sucesso); e pendente (o banco não conseguiu realizar o débito por falta de fundos e portanto deverá ser pago à parte). Apesar da modelagem com o padrão Estado parecer adequada, se analisarmos mais demoradamente o comportamento do

registro de uso, veremos que vários dias decorrem desde o estado inicial (a receber) até o estado final (pago). Assim, talvez não seja vantajoso manter em memória um objeto para cada possível estado, visto que provavelmente vários deles não seriam utilizados para nada. Por isso, neste caso, o uso do padrão é desaconselhado, sendo melhor prática de projeto utilizar um campo de *status* para armazenar a situação do registro de uso.

Quadro 9.2 – Padrão Estado

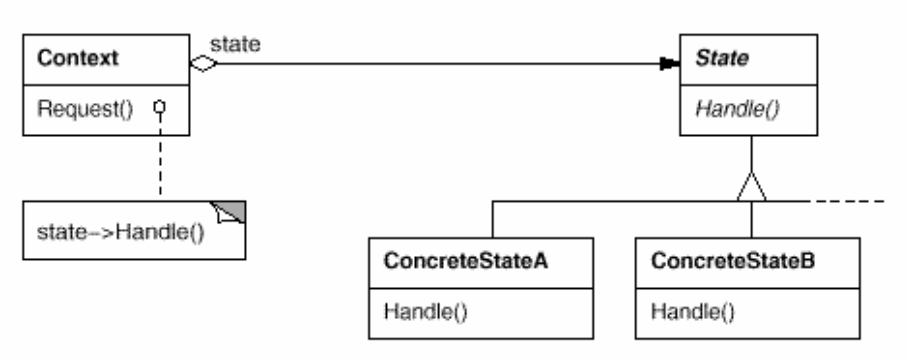
Nome: Estado (*State* em inglês)

Intenção: Permite que um objeto altere seu comportamento de acordo com mudança interna de estado. Parecerá que o objeto mudou de classe.

Motivação: Considere uma classe ConexãoTCP que representa uma conexão de rede. Um objeto desta classe pode estar em vários estados: "Estabelecida", "Em escuta", "Fechada", por exemplo. Mas quando o objeto recebe uma requisição vinda de outro objeto, sua reação dependerá do estado em que se encontrar no momento. O padrão Estado descreve como a classe ConexãoTCP pode exibir comportamentos diferentes para cada estado, sem no entanto utilizar inúmeros comandos condicionais ou estruturas CASE.

Aplicabilidade: Use o padrão Estado em quaisquer das seguintes situações: a) O comportamento de um objeto depende de seu estado e ele precisa mudar de comportamento em tempo de execução, dependendo de tal estado; b) Operações possuem comandos grandes, de múltiplas partes condicionais, que dependem do estado do objeto e, em geral, esse estado é representado por uma ou mais constantes enumeradas, sendo freqüentemente necessário repetir essa estrutura condicional em diversas operações. O padrão Estadofaz com que cada ramo da condicional fique em uma classe separada permitindo que o estado do objeto seja tratado como um objeto em si e que esse estado possa variar independentemente de outros objetos.

Estrutura:



Participantes:
Contexto - apresenta uma interface única com o mundo externo
Estado - classe abstrata básica
EstadoConcretoA, EstadoConcretoB, … - diferentes estados da máquina de estados como

classes derivadas da classe abstrata básica

Colaborações: Fazer com que um objeto comporte-se de uma forma determinada por seu estado. Agregar um objeto Estado e delegar comportamento a ele. O comportamento específico de estado é definido nas classes derivadas de Estado. Manter um ponteiro para o estado atual na classe Contexto. Para mudar o estado da máquina de estados, mudar esse ponteiro.

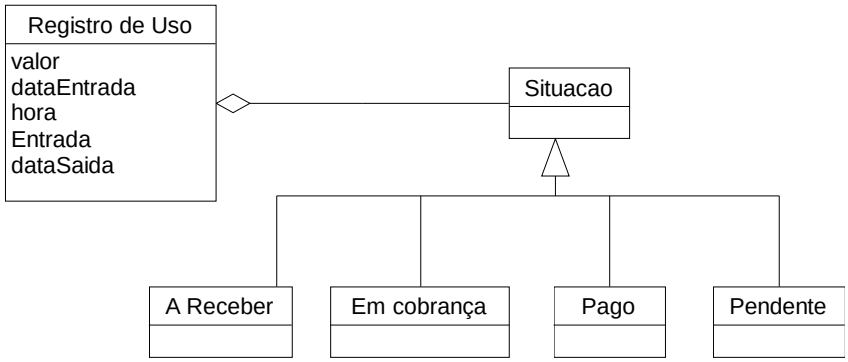


Figura 9.2 – Exemplo do padrão Estado no Sistema Passe Livre

Um outro exemplo no Sistema Passe Livre, agora mais adequado para aplicação do padrão Estado, é no projeto da classe Veículo, ilustrada na Figura 9.3. Um veículo que possui gizmo pode estar em um dos seguintes estados: fora da autopista; entrando na autopista (quando o gizmo foi detectado em uma área de pedágio de entrada); em circulação (quando está efetivamente na autopista); e saindo da autopista (quando o gizmo foi detectado em uma área de pedágio de saída). Esse caso é mais adequado para o uso do padrão porque nos interessa acompanhar de perto os estados do objeto, provavelmente mantendo em memória seus possíveis estados para facilitar a mudança de estado quando esta ocorrer.

Digamos que houvesse uma operação “saldo a pagar” que o proprietário pudesse consultar a qualquer momento, mesmo quando ele estivesse em circulação (digamos que em quiosques em postos de gasolina). Dependendo do estado do veículo, a implementação dessa operação seria diferente. Por exemplo, se ele estiver fora da autopista ou entrando na autopista, basta somar os registros de uso com estado *a receber*, *em cobrança* ou *pendente*. Já se ele estiver em circulação ou saindo, deve-se acrescentar o valor do uso atual, mesmo que parcial. A operação “saldo a pagar” é portanto um exemplo de comportamento dependente de estado, devendo ser implementado em cada estado específico. No caso desse exemplo, poderia haver uma implementação *default* na classe Veículo, sobreposta nos casos em que o veículo estivesse nos dois últimos estados.

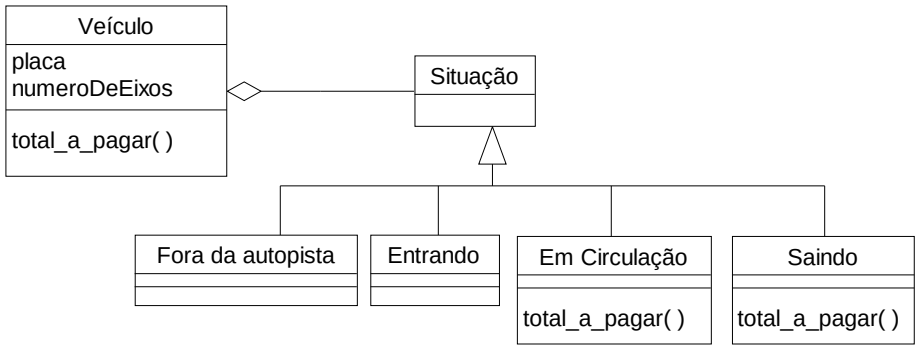


Figura 9.3 – Outro Exemplo do padrão Estado no Sistema Passe Livre

9.5 – Padrão Observador

O Quadro 9.3 resume o padrão Observador, cuja íntegra pode ser encontrada no livro de Gamma e outros [2005]. O principal tipo de aplicação em que se recomenda o uso do padrão Observador é aquele em que existem informações centralizadas em uma única fonte (*sujeito*), com diversas visões (*observadores*) que devem ser atualizadas sempre que a informação for modificada, como ilustrado na Figura 9.4.

Quadro 9.3– Padrão Observador

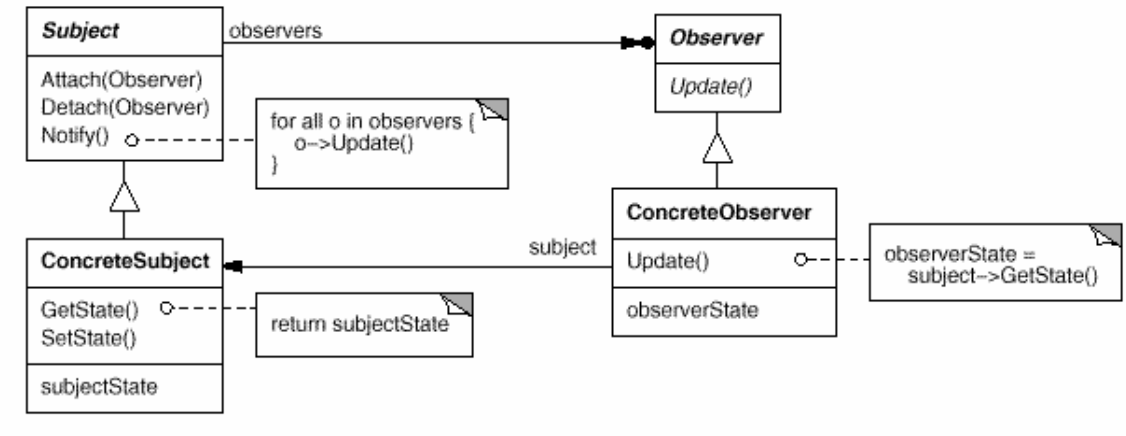
Nome: Observador (*Observer* em inglês)

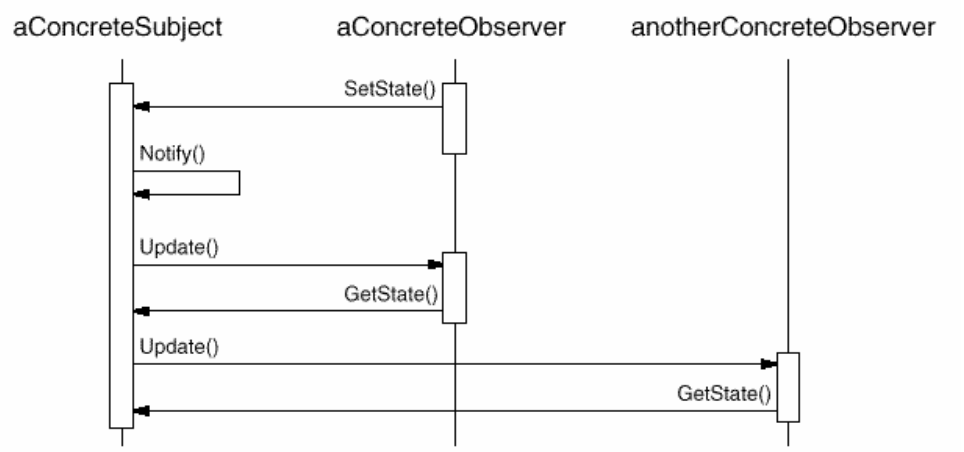
Intenção: Definir uma dependência de um-para-muitos entre objetos, de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

Motivação: Quando se utiliza orientação a objetos, particiona-se o sistema em um conjunto de classes que interagem, mas fica difícil manter a consistência entre objetos relacionados. Qualquer tentativa de manter a consistência aumenta o acoplamento entre as classes. Por exemplo, em uma ferramenta integrada, pode-se separar os dados em si das diversas visões que têm desses dados, de forma que mudança em uma das visões seja refletida nas demais, mesmo que as visões não tenham conhecimento umas das outras.

Aplicabilidade: Use o padrão Observador em quaisquer das seguintes situações: a) quando uma abstração tem dois aspectos, um dependente do outro. Encapsular esses aspectos em objetos separados permite variar e reutilizá-los independentemente; b) quando uma mudança em um objeto requer mudar outros, e não se sabe quantos objetos devem ser mudados; c) quando um objeto deve ser capaz de notificar outros objetos sem assumir quem são esses objetos, isto é, não é desejável que esses objetos estejam fortemente acoplados.

Estrutura:





Participantes:
Sujeito – contém os dados sendo em observação; pode modificar os dados; possui uma lista dos observadores.
Observador – classe interessada nas mudanças ocorridas no *Sujeito*.
SujeitoConcreto – armazena o estado de interesse do objetos de *ObservadorConcreto*. Envia uma notificação aos seus observadores quando muda de estado.
ObservadorConcreto – mantém uma referência ao objeto SujeitoConcreto. Armazena o estado que deve estar consistente com o do sujeito.

Colaborações: O observador pode incluir novos observadores, bem como exclui-los. Uma modificação no estado do sujeito pode ser solicitada por um dos observadores. Então, o sujeito muda seu estado e notifica os demais observadores sobre essa mudança. O observador possui uma implementação específica para se atualizar, se assim desejar (no diagrama de seqüência acima, o observador obtém o estado atual para poder se atualizar).

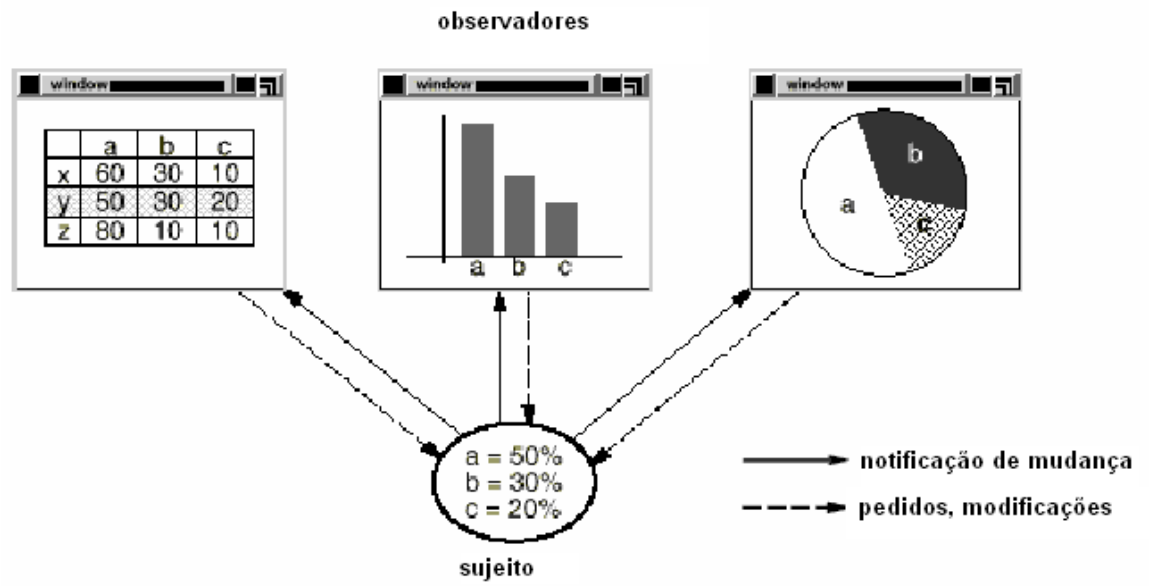


Figura 9.4 – Exemplo de aplicação em que o padrão Observador é recomendado

No sistema Passe Livre podemos citar várias ocorrências do padrão Observador. Na Figura 9.5 ilustra-se seu uso para modelar o sub-sistema Pedágio, em que modificações na informação sobre o "Sensor de Gizmo" (sujeito) devem ser notificadas a todos os seus observadores (cancela, semáforo, painel, etc.).

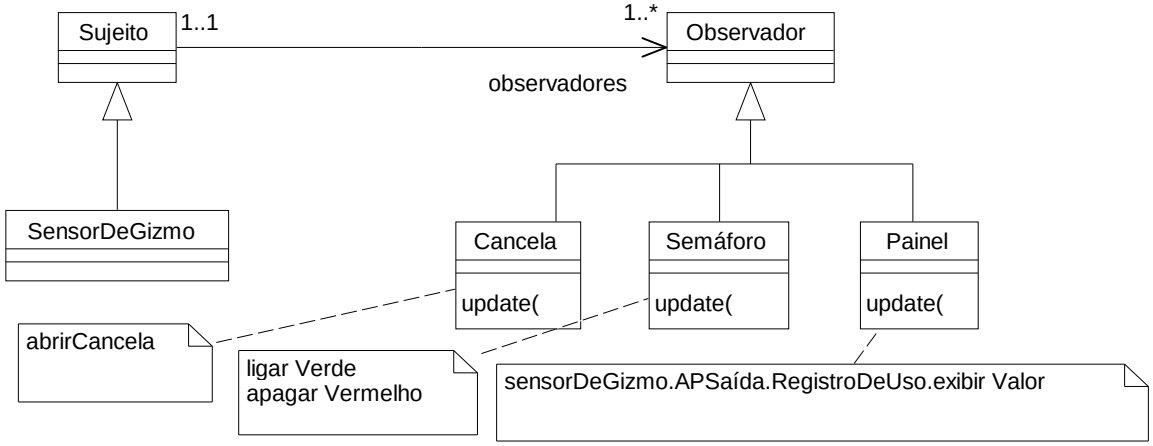


Figura 9.5 – Exemplo de aplicação do padrão Observador no Sistema Passe Livre

9.6 – Padrão Iterador

O padrão Iterador (Quadro 9.4) é bastante útil em qualquer projeto OO, tendo inclusive implementação disponível em diversas linguagens (Java, C#, etc.).

Quadro 9.4– Padrão Iterador

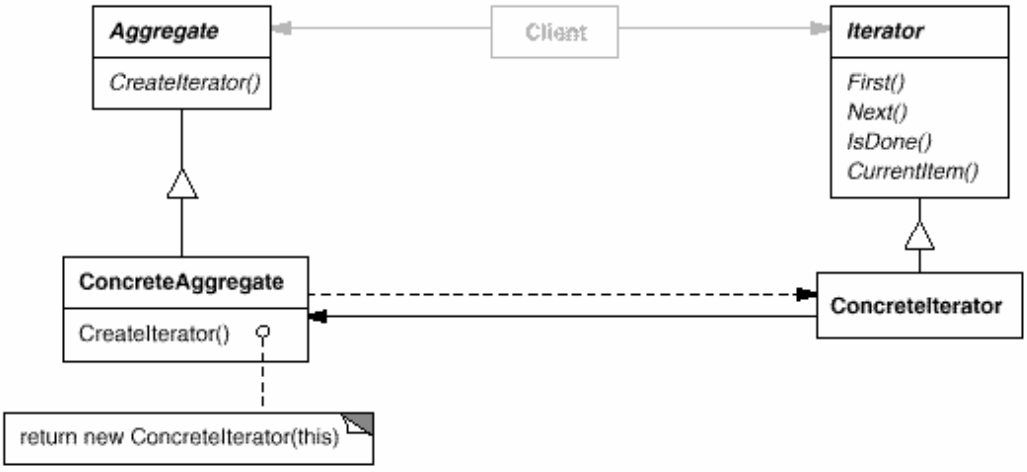
Nome: Iterador (*Iterator* em inglês)

Intenção: Existe a necessidade de percorrer agregados quaisquer, em diversas ordens, sem conhecer sua representação subjacente

Motivação: além de acessar os elementos sem conhecer a estrutura interna do agregado, pode ser desejável percorrer o agregado de diferentes formas, sem poluir a interface com inúmeros métodos para isso e mantendo controle do percurso. A solução é criar uma classe abstrata *Iterador*, que terá a interface de comunicação com o cliente. Para cada agregado concreto, criar uma subclasse de *Iterador*, contendo a implementação dos métodos necessários para percorrer o agregado.

Aplicabilidade: Use o padrão Iterador quando: a) para acessar o conteúdo de um objeto agregado sem expor sua representação interna; b) para apoiar diferentes trajetos de objetos agregados; c) para conseguir uma interface uniforme para percorrer diferentes estruturas de agregados (isto é, para apoiar a iteração polimórfica).

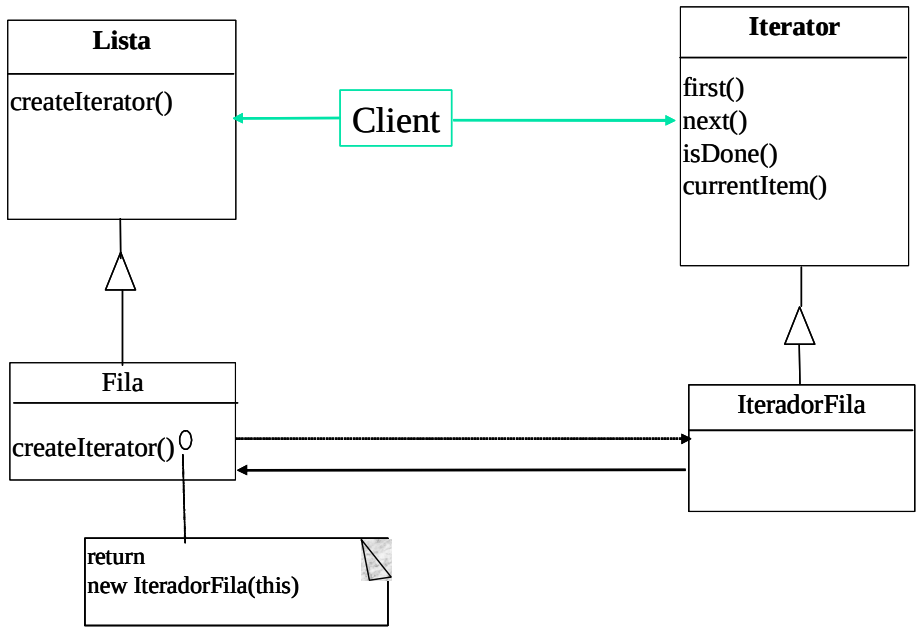
Estrutura:



Participantes:
Agregado - abstração para o agregado a ser percorrido.
AgregadoConcreto - representa o agregado real a ser percorrido. Pode possuir vários iteradores percorrendo-o simultaneamente.
Iterador - abstração para as operações básicas de iteração.
IteradorConcreto - iterador que realiza efetivamente o trajeto no agregado real.

Colaborações: Cada tipo de *agregado* possui a implementação de um *iterador* correspondente. Assim, quando o cliente cria um agregado, pode invocar o método *CriarIterador*() para obter um iterador, e usar sempre os métodos *Primeiro*, *Proximo*, *estaPronto* e *ItemCorrente* para percorrer o agregado, seja ele qual for.

Exemplo:



No Sistema Passe Livre, que foi implementado em Java, utilizou-se diversas vezes o padrão Iterador por meio da API Java (pacote java.util), que contém uma interface Iterator com os métodos: *next*(), *hasNext*() e *remove*(). O método *next*() é uma combinação do *next*() e *currentItem*() do padrão. A sub-interface *ListIterator* especifica esses métodos para operarem nos objetos *List* e adiciona os métodos apropriados aos objetos *List*. A interface *List* tem um método *listIterator*(), que retoma um objeto *ListIterator* que itera sobre ele. Isso permite ao programador mover-se e iterar sobre objetos *List*. Na Figura 8.8 (Capítulo 8) há um trecho de código em Java que mostra o uso do Iterador.

9.7 – Padrão Fábrica Abstrata

O padrão Fábrica Abstrata (Quadro 9.5) é muito utilizado em programação orientada a objetos, principalmente quando é necessário instanciar objetos pertencentes a um conjunto de classes relacionadas, mas não é desejável que o nome dessas classes apareça explicitamente no código, pois o código pode precisar ser configurado para funcionar com outros conjuntos de classes similares.

Quadro 9.5– Padrão Fábrica Abstrata

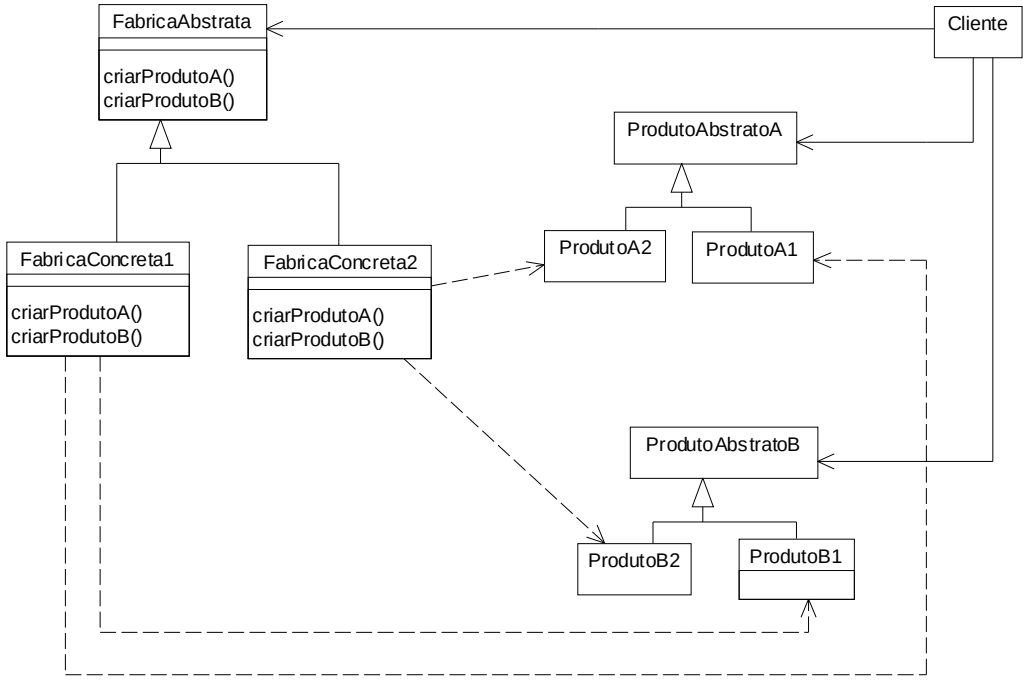
Nome: Fábrica Abstrata (*Abstract Factory* em inglês)

Intenção: Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar a classe concreta ao qual pertencem.

Motivação: Considere um ambiente de interface com o usuário que possua suporte para várias plataformas, tais como Windows e Linux. Operações como criarBarraDeRolagem e criarJanela são específicas de cada um dessas plataformas. É indesejável que o código da aplicação invoque essas operações aos respectivos objetos diretamente no código-fonte, pois ao mudar de plataforma todo o código teria que ser reescrito.

Aplicabilidade: Use o padrão Fábrica Abstrata quando: a) o sistema deveria ser independente de como seus produtos são criados, compostos e representados; b) o sistema deve ser configurado com uma de múltiplas famílias de produtos; c) uma família de produtos relacionados é projetada para ser usada em conjunto, sendo necessário garantir esta restrição; ou d) é desejável oferecer uma biblioteca de classes de produtos, revelando apenas sua interface, mas não sua implementação.

Estrutura:

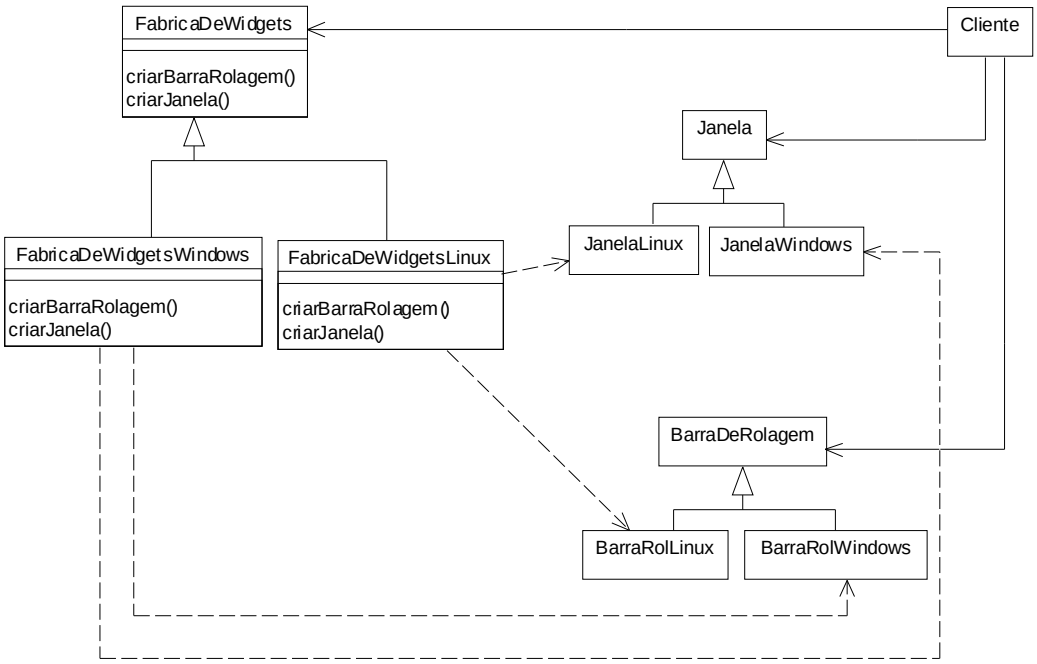


Participantes:
FabricaAbstrata – declara uma interface para as operações que criam os objetos do produto abstrato.
FabricaConcreta – implementa as operações para criar os objetos do produto concreto.

ProdutoAbstrato – declara uma interface para o objeto do tipo de produto.
ProdutoConcreto – define um objeto do produto a ser criado pela respectiva fábrica concreta. Implementa a interface de ProdutoAbstrato.
Cliente – usa apenas a interface declarada pelas classes FabricaAbstrata e ProdutoAbstrato.

Colaborações: em geral apenas uma instância da classe FabricaConcreta é criada em tempo de execução, que cria produtos com implementação específica. Para criar produtos diferentes, os clientes devem usar uma fábrica diferente. A FábricaAbstrata delega a criação de produtos para sua subclasse de FabricaConcreta.

Exemplo:



No Sistema Passe Livre, utilizou-se o padrão Fábrica Abstrata para implementar uma fábrica de mecanismos de persistência de objetos, de forma que seja possível facilmente alterar a forma de persistência dos objetos, por exemplo, de banco de dados relacional Mysql para Oracle ou para arquivos XML, como exemplificado na Figura

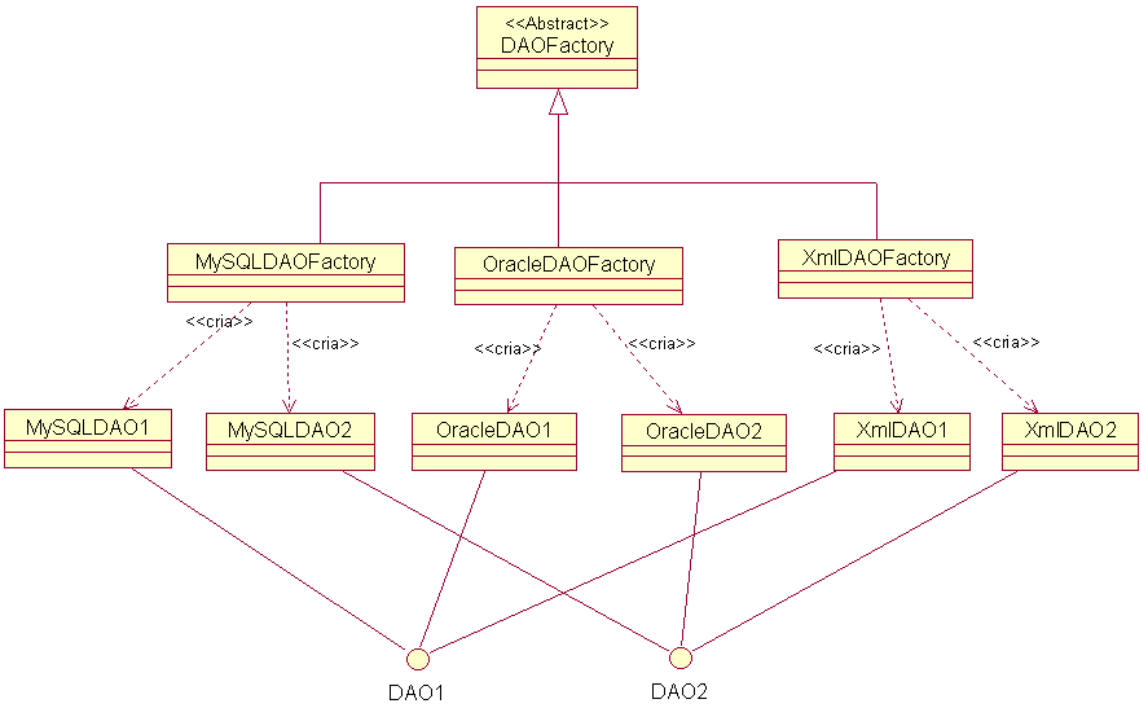


Figura 9.6 – Exemplo de aplicação do padrão Fábrica Abstrata no Sistema Passe Livre

9.8 – Exercícios Propostos

9.8.1 – Reveja o projeto do Sistema Passe Livre e utilize o padrão Estado para modelar outras classes dependentes de estado. Comente a adequação do uso do padrão nesses casos.

9.8.2 – Em que outros casos no Sistema Passe Livre você utilizaria o padrão Iterador? Explique.

9.8.3 – Comente pelo menos mais um caso em que o padrão Fábrica Abstrata poderia ser utilizado no Sistema Passe Livre.

9.9 – Exercícios complementares

9.9.1 – Considere um sistema para venda de equipamentos de informática, em que os computadores são montados a partir de partes adquiridas prontas. Algumas partes podem ser compostas de partes menores. Um usuário pode desejar um equipamento completo, mas também pode adquirir partes individuais ou compostas. Use o padrão Composto para produzir um projeto OO para esse sistema. Justifique porque ele é adequado, considerando que a operação principal do sistema é a que calcula o preço do equipamento.

9.9.2 – Considere um jogo RPG, em que um personagem pode estar em vários estados e seu comportamento diante dos eventos varie de acordo com esses estados. Como você usaria o padrão Estado para modelar essa situação? Mostre seu modelo.

Capítulo 10 – Exemplo – Sistema Passe Livre

10.1 – Descrição do Sistema Exemplo

O sistema escolhido para exemplificar o desenvolvimento orientado a objetos usando o PU é um Sistema para Controle de pedágios, denominado **Sistema Passe Livre**. Uma empresa ganhou a concessão para administrar várias rodovias estaduais e deseja implantar um sistema automático de cobrança de pedágios. Nas rodovias, chamadas autopistas, existe um controle em todas as entradas e saídas. Os proprietários compram um dispositivo eletrônico chamado gizmo, ativam-no pela Internet e depois o afixam no interior do veículo. Ao entrar na autopista, o sistema registra o ponto de entrada e ao sair da autopista passando por um posto de pedágio, o sistema registra a saída e o valor a ser cobrado de acordo com o tipo do carro e da distância percorrida. A Figura 10.1 ilustra o funcionamento da autopista. Os requisitos detalhados do Sistema Passe Livre encontram-se no Apêndice A. Resumidamente, ao entrar na área de pedágio o gizmo do veículo é detectado por um sensor, que envia os dados do veículo ao sistema e caso sua situação esteja regularizada a cancela é aberta e o veículo pode transitar pela rodovia. Ao sair da rodovia, novamente o gizmo é identificado, o valor do pedágio a ser cobrado é calculado e mostrado em um painel e a cancela é aberta para que o veículo possa sair. Ao sair da área de pedágio, um sensor notifica a saída para que a cancela seja fechada.

Figura 10.1 – Ilustração do Sistema Passe Livre

A cobrança é feita mensalmente por débito bancário e o cliente pode optar por um dos bancos autorizados e escolher o melhor dia para débito, dentre três datas oferecidas pelo

sistema. Se o cliente tornar-se inadimplente, o gizmo é cancelado e não poderá mais ser utilizado ao trafegar pela autopista. O cliente que não possui gizmo em seu veículo ou que possui um gizmo cancelado deve utilizar outra entrada para a autopista, na qual recebe um bilhete e paga-o ao sair da autopista.

10.2 – Definição dos casos de uso do Sistema

Conforme visto no Capítulo 2, o PU é fortemente dirigido por casos de uso. Isso significa que os casos de uso são utilizados o tempo todo, desde a coleta inicial dos requisitos até a validação do produto final.

Assim, o primeiro passo é a identificação dos casos de uso. Em nosso caso específico, faremos isso com base nos requisitos do Sistema Passe Livre (ver Apêndice A). Nesta etapa queremos apenas definir quais são os casos de uso, mas eles ainda não serão detalhados, o que será feito posteriormente, a cada ciclo de desenvolvimento. Como a próxima etapa será planejar os ciclos de desenvolvimento, precisamos ter a noção de quantos casos de uso teremos, sua complexidade e prioridade.

Analisando-se cada requisito, deve-se determinar qual caso de uso é responsável por atendê-lo, ou seja, pergunta-se como o requisito pode ser satisfeito por meio de uma interação entre o usuário e o sistema. Caso ainda não exista um caso de uso para satisfazer o requisito, ele é criado e incluído na tabela de referência cruzada, conforme explicado na Seção 3. Se o caso de uso já existe, basta atualizar a referência cruzada com o número do requisito correspondente.

Por exemplo, considere o requisito 1.1:

"O sistema deve apoiar a compra de *gizmos*. O *gizmo* pode ser comprado em *postos* autorizados por uma taxa que deve ser paga à vista. No ato da compra, o *proprietário* do *veículo* informa ao *atendente* do *posto* seus detalhes pessoais, como endereço, carteira de habilitação e CPF; os dados do *veículo* e seus dados bancários. O *proprietário* deve optar por um dos *bancos* autorizados e escolher a melhor data para débito, dentre as disponibilizadas pelo sistema, que são: dia 1, 10 ou 20 de cada mês".

Pelo enunciado, percebe-se a necessidade do proprietário comprar um gizmo em um posto autorizado. Sabe-se que esse posto tem um funcionário que opera o sistema. Portanto, deduz-se que haverá um caso de uso, ao qual daremos o nome de "Comprar Gizmo", pelo qual esse requisito será satisfeito.

De maneira análoga, ao avaliar os requisitos 1.2 a 1.5, percebe-se que deve ser criado um caso de uso para cada um deles. São eles: "Alterar Senha Proprietário", "Alterar Proprietário/veículo", "Cancelar Gizmo" e "Reativar Gizmo". Entretanto, ao passar para o requisito 1.6:

"*Proprietários de mais de um veículo devem comprar um gizmo para cada veículo*"

verifica-se que ele deve ser satisfeito pelo mesmo caso de uso já criado anteriormente ("Comprar Gizmo"), pois faz parte da funcionalidade do proprietário que está em uma loja comprando um gizmo. Assim, para que o requisito seja satisfeito, o caso de uso "Comprar gizmo" deve prever o caso de um proprietário já registrado, e permitir que ele compre mais gizmos.

Paralelamente à identificação dos casos de uso, pode-se descrevê-los, usando o formato resumido, para facilitar seuagrupamento por ordem de prioridade. A Tabela 10.1 exemplifica alguns casos de uso para o Sistema Passe Livre, já descritos no formato resumido, bem como sua correspondência com os requisitos do sistema, para garantir que todos os requisitos foram atendidos. A lista completa pode ser encontrada no apêndice B, em que os casos de uso foram agrupados em três sub-sistemas: administrativo, financeiro e de pedágio.

Tabela 10.1 – Alguns Requisitos x Casos de Uso – Sistema Passe Livre

REQUISITOS x CASOS DE USO NO FORMATO RESUMIDO	
1.1; 1.2; 1.6	**Comprar Gizmo** O Proprietário informa ao Atendente de uma loja autorizada que deseja adquirir um ou mais gizmos. O Atendente armazena no Sistema as informações pertinentes ao Proprietário e aos veículos. O Proprietário paga uma taxa à vista ao Atendente. O Atendente fornece ao Proprietário um nome de acesso, senha, os gizmos, assim como os números das placas dos respectivos veículos em que deverão ser afixados.
1.2	**Autenticar Usuário** O Usuário na tela de autenticação do Sistema Passe-Livre entra com o nome de acesso e senha. O Sistema confirma autenticação.
1.4	**Desativar Gizmo** O Proprietário faz sua autenticação no Sistema com nome de acesso e senha. Seleciona os veículos que terão os gizmos desativados. Se o Proprietário for adimplente o Sistema confirma a operação.
1.5	**Reativar Gizmo** O Proprietário informa ao Atendente de uma loja autorizada que deseja reativar um gizmo e informa a placa do veículo para o qual o gizmo será reativado. O Proprietário paga uma taxa ao Atendente. O sistema reativa o gizmo para funcionamento no veículo desejado.
1.15;1.16; 1.17	**Emitir Cobrança** O relógio do sistema detecta a ocorrência de uma data e hora específicas de fechamento e inicia o processamento da cobrança aos proprietários que escolheram esta data de cobrança. O sistema envia aos bancos os dados necessários para débito em conta corrente dos valores processados. O Sistema envia um aviso ao Analista Financeiro de que o processamento do fechamento ocorreu.
1.17; 1.22; 1.23	**Consultar Extrato** O Proprietário faz sua autenticação com o nome de acesso e senha, seleciona o mês / ano. O Sistema exibe os dados do extrato.
1.24	**Consultar Proprietários Inadimplentes** O Analista Financeiro solicita uma lista dos Proprietários inadimplentes. O sistema exibe as informações solicitadas.
1.7; 1.8; 1.9; 1.10; 1.26	**Entrar na Autopista** O sensor de gizmo detecta a entrada do veículo na área de pedágio em uma entrada da autopista. O Sistema identifica o veículo pelo código de seu gizmo e autoriza a abertura da cancela. O sistema registra a entrada do veículo, hora/data, código da rodovia, código da área de pedágio e código da pista. O veículo entra na autopista.
1.11; 1.12; 1.13; 1.14; 1.26	**Sair da Autopista** O sensor de gizmo detecta a entrada do veículo na área de pedágio em uma saída da autopista. O Sistema identifica o veículo pelo código de seu gizmo e calcula o total a ser pago pelo proprietário em função da distância percorrida desde sua entrada na autopista.O Sistema autoriza a abertura da cancela. O veículo sai da autopista.

Após elaborar essa lista, pode-se classificar os casos de uso em ordem de prioridade, o que ajuda na fase seguinte do processo de desenvolvimento, na qual define-se quantos ciclos de iteração serão realizados e quais são os casos de uso a serem tratados em cada um dos ciclos. No caso da Tabela 10.1, os casos de uso não estão em ordem de prioridade, são apenas um subconjunto de todos os casos de uso do sistema Passe Livre.

10.3 – Definição da arquitetura do sistema

Outra característica importante do PU é ser centrado na arquitetura, conforme já mencionado no Capítulo 2. Desde os primeiros estágios de desenvolvimento deve-se especificar a arquitetura do sistema em construção, e refiná-la na medida em que novos casos de uso são incorporados ao sistema.

No Sistema Passe Livre, fazendo-se uma breve leitura dos requisitos, verifica-se que ele pode ser subdividido em quatro sub-sistemas: o primeiro para uso no posto de venda de gizmos (sub-sistema administrativo), o segundo para o controle financeiro associado à arrecadação de pedágios (sub-sistema financeiro), o terceiro para controle de cada pedágio (sub-sistema de pedágio) e o quarto para cuidar do armazenamento dos dados necessários ao funcionamento do sistema (sub-sistema Núcleo Central). Esses quatro subsistemas devem comunicar-se entre si, sendo fortemente dependentes um do outro, conforme ilustrado na Figura 10.2, que representa a interação dos elementos do sistema em um nível mais alto de abstração, somente pensando-se nos módulos relacionados ao negócio em si.

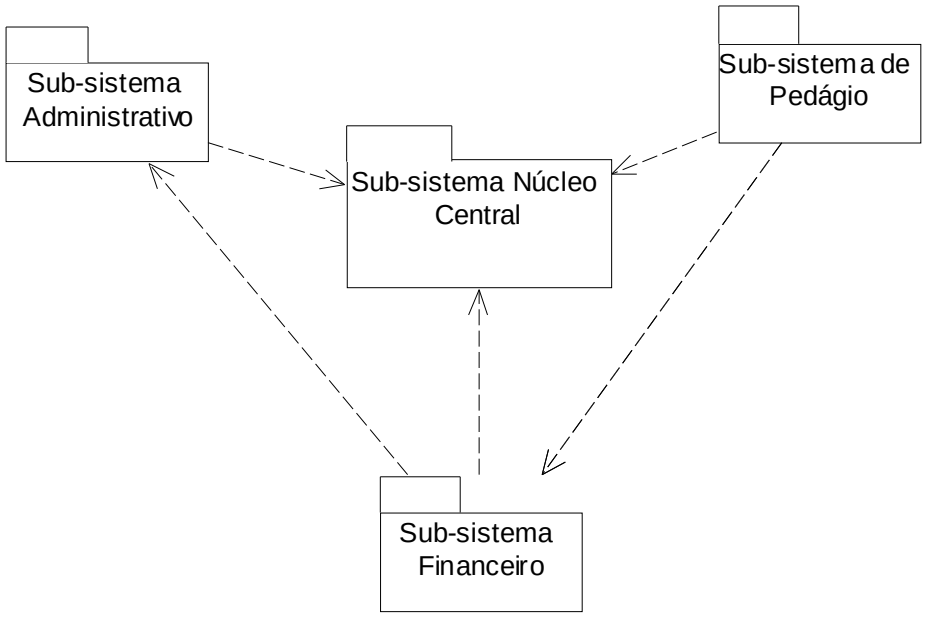


Figura 10.2 – Esboço dos vários sub-sistemas do Sistema Passe-Livre

As classes pertencentes a cada pacote serão identificadas durante a etapa de elaboração do PU, mas pode-se adiantar que:

- O Sub-sistema Administrativo conterá basicamente o comportamento correspondente ao proprietário e ao gizmo.
- O Sub-sistema Financeiro tratará da cobrança enviada ao banco para que seja feito o débito na conta do proprietário
- O Sub-sistema de pedágio implementará a entrada e saída de veículos da autopista.
- O sub-sistema Núcleo Central cuidará de registrar as diversas tabelas necessárias para o funcionamento do sistema, tais como rodovia, banco, atendente, pedágio, multas, juros, etc.

Os pacotes referentes aos sub-sistemas definidos na Figura 10.2 fazem parte da camada de negócios do Sistema Passe Livre. Assim, um segundo nível de detalhamento da arquitetura considera outras camadas a serem implementadas em software, como por exemplo, a camada GUI e a camada de persistência, discutidas nos Capítulos 8 e 9. Por exemplo, se for escolhida uma arquitetura em 3 camadas, a arquitetura do sistema Passe Livre pode ser como a da Figura 10.3. Considera-se que os pacotes da Figura 10.2 pertencem à camada de Negócios da Figura 10.3.

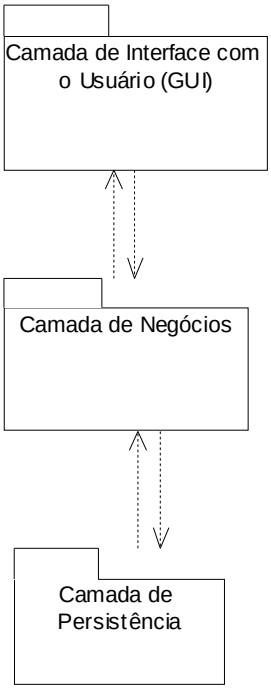


Figura 10.3 – Arquitetura do Sistema Passe-Livre em 3 camadas

A arquitetura pode ser ainda mais detalhada considerando-se a tecnologia a ser empregada na implementação. Por exemplo, a Figura 10.4 mostra a arquitetura do sistema Passe Livre quando implementada usando tecnologia Web e diversos frameworks para apoio à persistência, distribuição, etc.

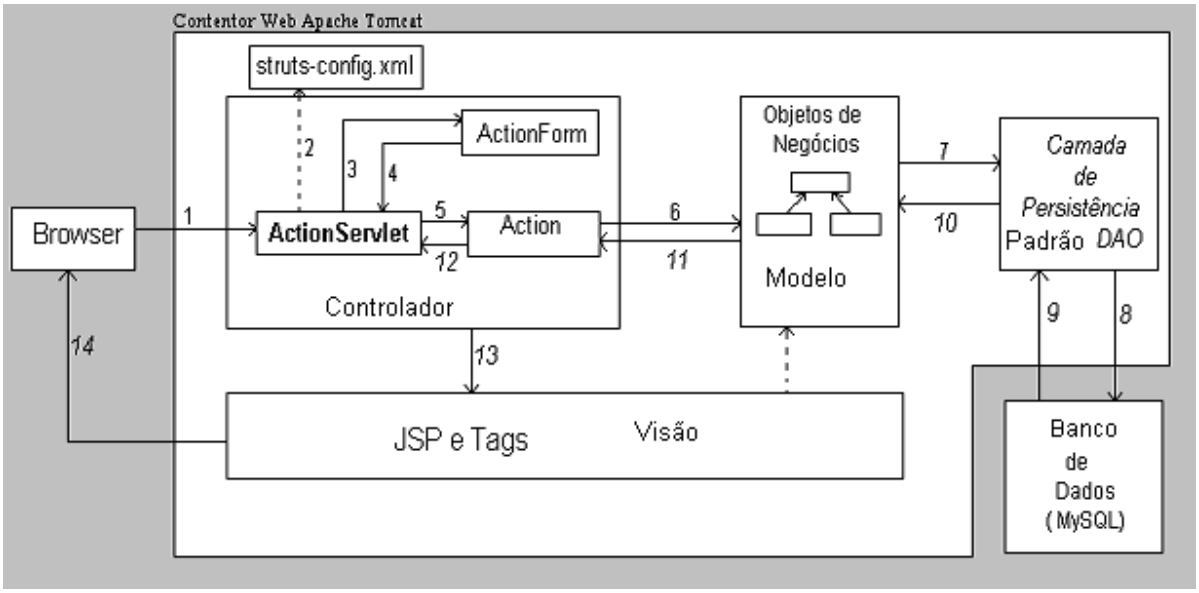


Figura 10.4 – Arquitetura detalhada do Sistema Passe-Livre para Web

10.4 – Criação do Diagrama de Casos de Uso do Sistema

O Processo Unificado estabelece que o diagrama de casos de uso deve ser definido logo no início da concepção do sistema, para ter uma visão global do escopo do sistema, dos atores que interagem com o sistema e das principais funcionalidades. Posteriormente, durante as iterações, ele pode ser refinando conforme necessário. As Figuras 10.5, 10.6, 10.7 e 10.8 contêm o Diagrama de Casos de Uso do Sistema Passe Livre, divididos de acordo com os sub-sistemas da Figura 10.2 para facilitar a compreensão, mas na verdade pode-se considerá-los como um único Diagrama de Casos de Uso.

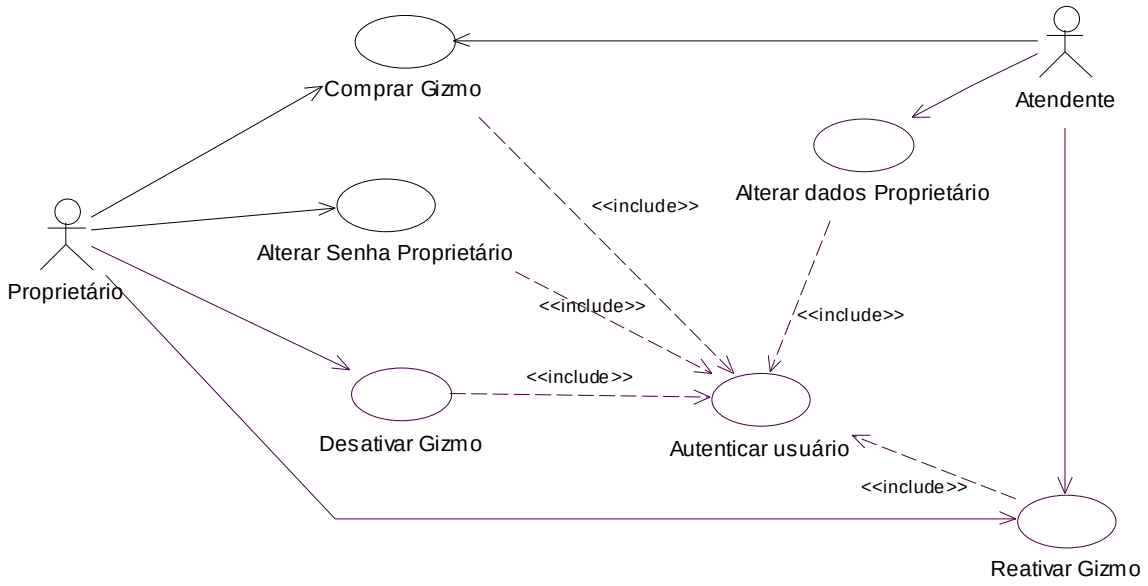


Figura 10.5 – Diagrama de Casos de Uso do Sistema Passe-Livre (sub-sistema Administrativo)

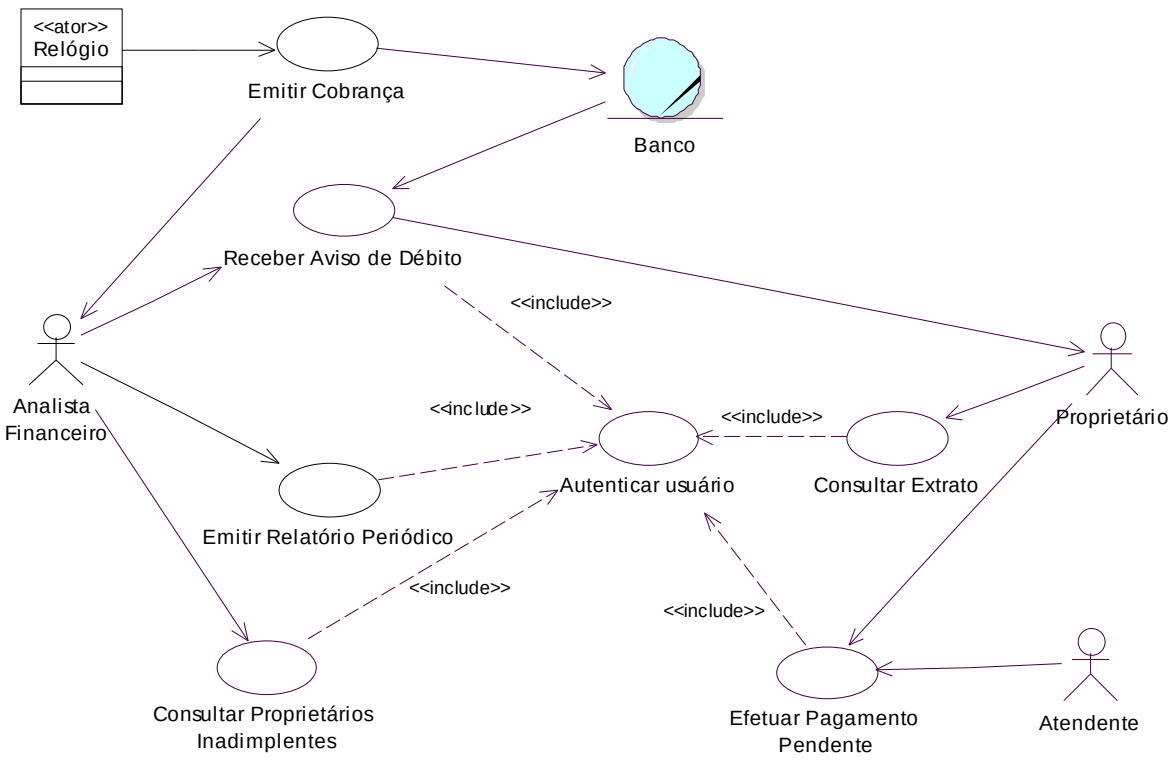


Figura 10.6 – Diagrama de Casos de Uso do Sistema Passe-Livre (sub-sistema financeiro)

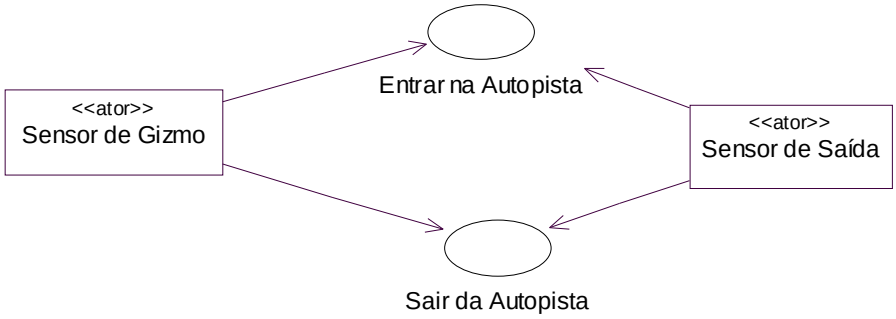


Figura 10.7 – Diagrama de Casos de Uso do Sistema Passe-Livre (sub-sistema de pedágio)

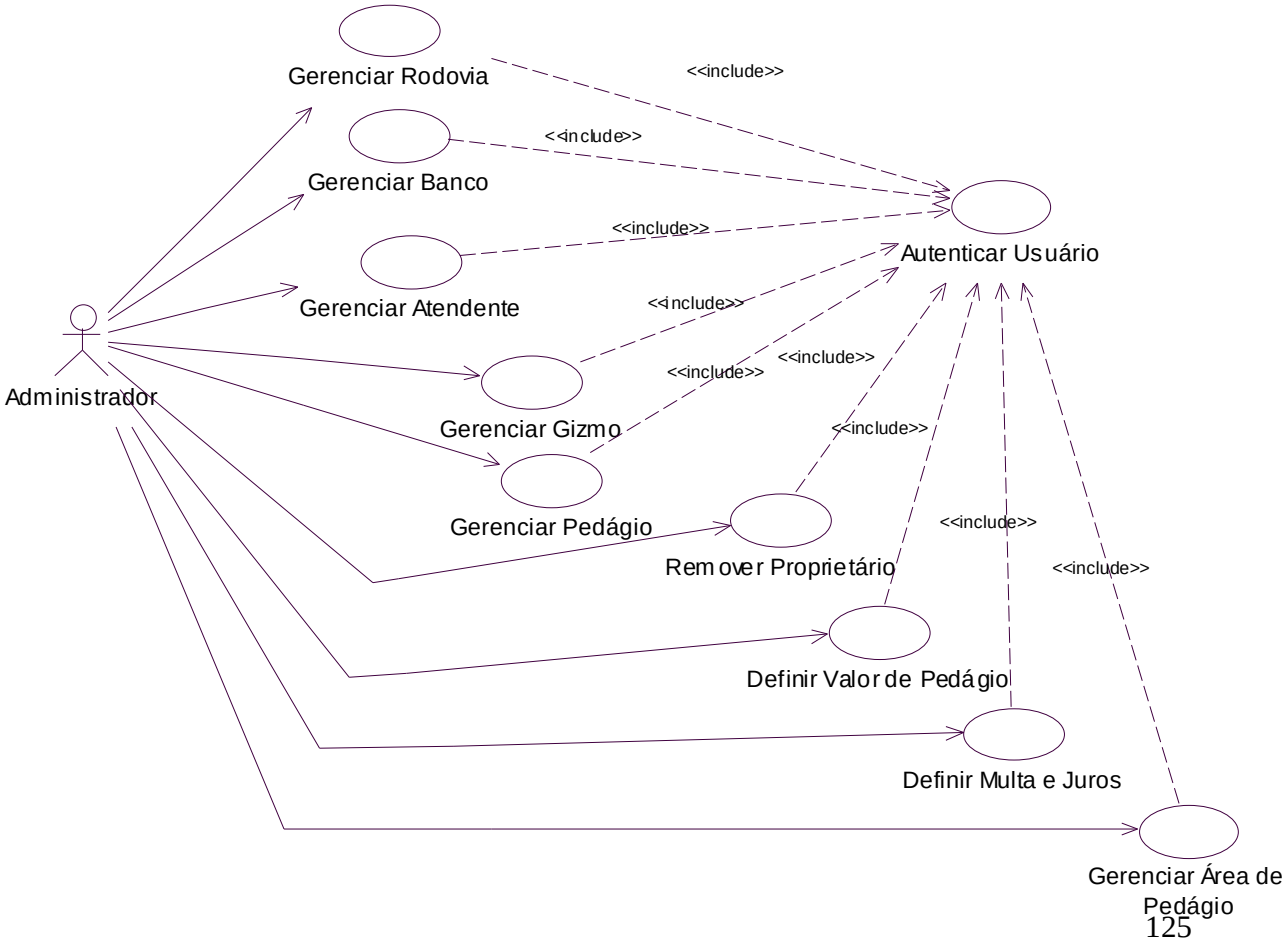


Figura 10.8 – Diagrama de Casos de Uso do Sistema Passe-Livre (sub-sistema Núcleo Central)

Note que há mais casos de uso nos diagramas 10.5 a 10.8 do que os listados na Tabela 10.1 A Tabela completa contendo todos os casos de uso versus os requisitos que o cobrem pode ser encontrada no Apêndice B.

Um recurso das ferramentas CASE é ter representações gráficas diferentes para os diversos tipos de ator que podem aparecer em um caso de uso. Por exemplo, a Rational Rose [IBM, 2004] oferece uma diversidade de ícones, como por exemplo os mostrados na Figura 10.9.

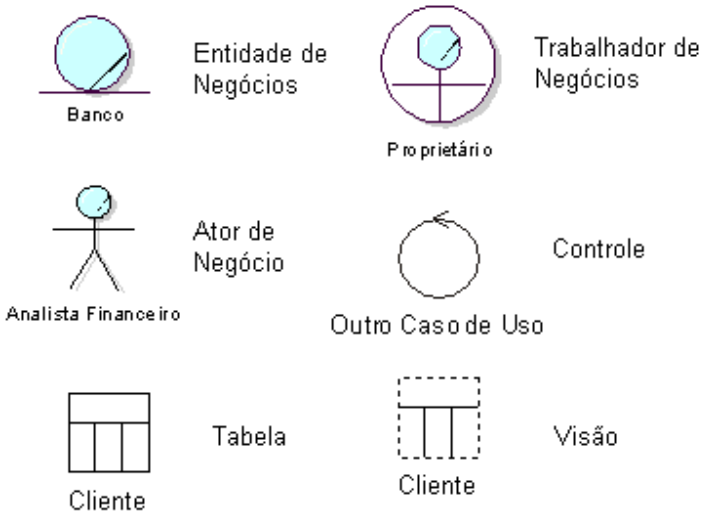


Figura 10.9 – Diversas Representações para atores

10.5 – Definição dos ciclos de iteração

Depois de definir os casos de uso, o próximo passo, conforme visto no Capítulo 2, é atribuir prioridades a eles e agrupá-los em ciclos de desenvolvimento, de acordo com sua complexidade e riscos associados, de forma que os casos de uso mais sujeitos a causar problemas sejam atacados nos primeiros ciclos de desenvolvimento.

Para o Sistema Passe Livre, foram definidos três ciclos de desenvolvimento, sendo os dois primeiros explicados nos capítulos seguintes. Os casos de uso alocados ao ciclo 1, por serem aqueles com maior complexidade, que envolvem diversas entidades do sistema e processamento lógico mais elaborado, foram:
1 – Comprar Gizmo
2 – Autenticar Usuário
3 – Consultar Extrato
4 – Emitir Relatório Periódico

5 – Entrar na Autopista
6 – Sair da Autopista
7 – Emitir Cobrança.

Os casos alocados ao ciclo 2 tratam de questões importantes para o sistema, mas sua complexidade é menor. Foram alocados com o objetivo de que o sistema já possa entrar em operação no fim do ciclo 2. São eles:
1 – Receber Aviso de Débito
2 – Desativar Gizmo
3 – Reativar Gizmo
4 – Efetuar Pagamento Pendente
5 – Consultar Proprietários Inadimplentes
6 – Alterar dados do proprietário
7 – Alterar senha proprietário.

Foram deixados para o ciclo 3 os casos de uso referentes a arquivos diversos, que não impedem a operação do sistema, pois esses arquivos poderiam ser incluídos diretamente na base de dados em ciclos anteriores. São eles:
1 – Gerenciar Rodovia
2 – Gerenciar Banco
3 – Gerenciar Atendente
4 – Gerenciar Gizmo
5 – Gerenciar Pedágio
6 – Remover Proprietário
7 – Definir Valor de Pedágio
8 – Definir Multa e Juros
9 – Gerenciar Área de Pedágio.

A divisão foi relativamente satisfatória em termos de tamanho de cada ciclo de iteração, de forma que o sistema poderá ser implementado em 3 ciclos praticamente de tamanho igual, digamos de 2 meses cada. Deve-se notar que os casos de uso referentes à entrada e saída do veículo da autopista são mais complexos, por envolver software e hardware específicos. Uma possibilidade é averiguar a existência de sistemas prontos para esses casos de uso. No caso deste livro, faremos uma simulação desta parte do sistema, ao invés de implementar concretamente os sensores físicos.

10.6 – Como atacar cada ciclo

Tendo-se definido os casos de uso alocados a cada ciclo, parte-se para a análise e projeto detalhados desses casos de uso, seguindo até a implementação e implantação. Ou seja, cada ciclo de desenvolvimento passa pelas fases: Concepção, Elaboração, Construção e Transição do Processo Unificado, produzindo ao final do ciclo um incremento que acrescenta uma parte operacional do sistema. Dentro de cada fase, várias iterações são realizadas, por exemplo, cada caso de uso pode ser desenvolvido separadamente até chegar ao conjunto de casos de uso estabelecido para o ciclo.

10.7 – Exercícios complementares

10.7.1. Faça uma outra divisão dos casos de uso em quatro ciclos evolutivos e justifique sua decisão.

10.7.2. Explique como funciona o ator Relógio (figura 10.6). Quais as implicações dessa decisão (incluir esse ator)?

Capítulo 11 – Ciclo de Desenvolvimento 1 do Sistema Passe Livre – Fase de Requisitos

11.1 – Casos de Uso no formato completo abstrato

Conforme dito no Capítulo 10, no primeiro ciclo de desenvolvimento foram alocados os sete casos de uso a seguir: 1 - Comprar Gizmo, 2 - Autenticar Usuário, 3 - Consultar Extrato, 4 – Emitir Relatório Periódico, 5 - Entrar na Autopista, 6 - Sair da Autopista e 7 - Emitir Cobrança.

Para cada um desses casos de uso, deve-se fazer sua descrição no formato completo abstrato, conforme ilustrado na Figura 11.1 para o caso de uso Entrar na Autopista. Na Figura 11.2 mostra-se o caso de uso Autenticar o usuário. A idéia é pensar apenas na interação entre os atores e o sistema para cumprir o comportamento esperado pelo caso de uso, sem prender-se a detalhes ligados à tecnologia ou implementação, pois isso será feito posteriormente, no projeto OO. Outros exemplos de casos de uso no formato completo abstrato para o sistema Passe Livre podem ser encontrados no Apêndice B.

A descrição do caso de uso na forma estendida é obtida pela análise dos requisitos e quando necessário, são feitas algumas restrições para facilitar a realização do caso de uso, podendo incluir os requisitos mais complexos em ciclos posteriores. Por exemplo, no sistema Passe Livre, considerou-se que nunca há simultaneamente dois veículos na mesma pista dentro da área de pedágio, pois se abrirmos a possibilidade de vários carros entrarem ao mesmo tempo, o projeto fica muito mais complexo. Também não são tratados os casos em que há falhas de hardware nos sensores, cancelas ou semáforos. Entretanto, em um sistema real isso terá que ser levado em consideração, mesmo que em ciclos posteriores.

Alguns requisitos não funcionais podem ser incluídos nos casos de uso, por exemplo, no passo 4 da Figura 11.1 menciona-se um requisito não funcional, que é o de que a autorização tem que ser feita em menos de 3 segundos (requisito de desempenho).

Nos fluxos alternativos, deve-se considerar todas as possibilidades de ocorrência de eventos que causem problemas que devam ser tratados pelo sistema. No entanto, as restrições que foram incluídas como pré-condições não devem ser tratadas nos fluxos alternativos, pois se considera que nunca ocorrerão. Por exemplo, no passo 6, o sensor de saída poderia falhar na detecção da passagem do veículo, o que faria com que a cancela permanecesse levantada, a luz verde ligada e a vermelha apagada. Mas essa falha não é tratada como fluxo alternativo, pois nas pré-condições foi especificado que os equipamentos funcionam perfeitamente.

Caso de Uso: Entrar na Autopista (Área de Pedágio)

Ator Principal: Proprietário
Interessados e Interesses:
Proprietário: deseja entrar e sair da autopista passando por áreas de pedágios de maneira rápida e segura.
Empresa: deseja que o Sistema identifique o gizmo do Proprietário e as informações do local onde o veículo entrou na autopista, para que seja possível efetuar a cobrança na saída da autopista.
Pré-Condições: O Proprietário possui um gizmo instalado em seu veículo. Nunca há simultaneamente dois veículos (na mesma pista) dentro da área de pedágio. Os equipamentos eletro-mecânicos do Sistema de pedágio funcionam perfeitamente.
Garantia de Sucesso (Pós-Condições): A cancela é aberta, a luz verde do semáforo é ligada, o veículo entra na autopista e os dados são registrados no Sistema. Após a entrada do veículo, a luz vermelha do semáforo é ligada e a cancela é fechada.
Cenário de Sucesso Principal:

1. O Proprietário chega com seu veículo à área de pedágio de uma pista passe livre na entrada da autopista e reduz a velocidade para 40 km/h.
2. O sensor de gizmo identifica o código do gizmo do Proprietário.
3. O Sistema identifica o código da rodovia, código da área de pedágio, o código da cabine e a data/hora.
4. O Sistema verifica se o veículo é autorizado e em menos de 3 segundos, registra os dados de entrada, envia um sinal para desligar a luz vermelha, ligar a luz verde e abrir a cancela.
5. O Proprietário passa pela cancela, deixa a área de pedágio e segue para a autopista.
6. O sensor de saída verifica que o veículo já passou pela cancela e informa ao Sistema.
7. O Sistema envia um sinal para desligar a luz verde, ligar a luz vermelha e fechar a cancela.

Fluxos Alternativos:
2a. O sensor do gizmo, por algum motivo, não identifica o código do gizmo do Proprietário.
    1. A cancela não abre.
    2. O Proprietário retorna ou é atendido manualmente.
4a. Se o gizmo não é autorizado.
    1. A cancela não abre.
    2. O Proprietário retorna ou é atendido manualmente.
4b. O Sistema não abre a cancela em menos de 3 segundos.
    1. A cancela não abre.
    2. O Proprietário freia o veículo e aguarda até que a cancela seja aberta.
2-5a. Se o Proprietário desistir da viagem, faz um retorno por dentro da autopista e sai por uma cabine de saída.

Figura 11.1 – Caso de Uso "Entrar na Autopista" no formato Completo Abstrato

Caso de Uso: Autenticar Usuário

Ator Principal: Usuário

Interessados e Interesses:
- Usuário: deseja fazer sua autenticação no Sistema pela Interface Web.
- Empresa: deseja impedir o acesso ao sistema de pessoas não autorizadas e que a autenticação do Usuário no sistema seja feita de maneira rápida e segura.

Pré-Condições: O Usuário possui um nome de acesso e senha.

Garantia de Sucesso (Pós-Condições): Usuário é autenticado no Sistema

Cenário de Sucesso Principal:
11. O Usuário digita no navegador web a URL do sistema.
12. O servidor web recupera a tela de autenticação do Sistema e apresenta ao Usuário.
13. O Usuário digita seu nome de acesso e senha.
14. O Sistema confirma autenticação.

Fluxos Alternativos:

4a. O Sistema informa ao Usuário que o nome de acesso ou a senha está incorreta.
1. O Proprietário poderá tentar se autenticar no Sistema por mais 3 vezes, caso não tenha sucesso o Sistema apresentará uma tela informando o ocorrido e cancelando a autenticação.

4b. Se for o primeiro acesso do Usuário, o Sistema apresenta uma tela para que seja feita a alteração da senha.

Requisitos especiais:
- A comunicação entre o Sistema na Web e o servidor deve usar criptografia e outras técnicas de segurança para oferecer maior segurança aos Usuários do Sistema.
- O Sistema deve estar sempre disponível e nunca deixar de operar. A tolerância máxima é de 15min por ano com o Sistema fora de operação, contando inclusive manutenções programadas.

Problemas em Aberto:
- Se o Proprietário entrar 3 vezes com a senha incorreta o Sistema vai desabilitar a senha ?
- Se o Proprietário esquecer a senha ele terá opção para enviá-la para o seu e-mail cadastrado ?

Figura 11.2 – Caso de Uso "Autenticar Usuário" no formato Completo Abstrato

11.2 – Modelo Conceitual

O próximo passo é iniciar a elaboração do Modelo Conceitual do Sistema, com base nos casos de uso. Para facilitar este passo, pode-se sublinhar os substantivos do caso de uso, conforme exemplificado na Figura 11.3. Com isso, obtém-se a lista de candidatos a conceitos da Figura 11.4 que, após analisados e refinados seguindo as regras da seção 4.1.1,

resultam na lista de conceitos a incluir no Modelo Conceitual. Note que a lista da Figura 11.4 foi obtida não somente sublinhando os substantivos do caso de uso Entrar na Autopista, mas também os demais casos de uso do ciclo 1, que estão no Apêndice B.

Caso de Uso: Entrar na Autopista (Área de Pedágio)

. . .

Cenário de Sucesso Principal:

1. O Proprietário chega com seu veículo à área de pedágio de uma pista passe livre na entrada da autopista e reduz a velocidade para 40 km/h.
2. O sensor de gizmo identifica o código do gizmo do Proprietário.
3. O Sistema identifica o código da rodovia, código da área de pedágio, o código da cabine e a data/hora.
4. O Sistema verifica se o veículo é autorizado e em menos de 3 segundos, registra os dados de entrada, envia um sinal para desligar a luz vermelha, ligar a luz verde e abrir a cancela.
5. O Proprietário passa pela cancela, deixa a área de pedágio e segue para a autopista.
6. O sensor de saída verifica que o veículo já passou pela cancela e informa ao Sistema.
7. O Sistema envia um sinal para desligar a luz verde, ligar a luz vermelha e fechar a cancela.

. . .

Figura 11.3 – Caso de Uso "Entrar na Autopista" com substantivos grifados

Analista Financeiro	Extrato	placas dos veículos
Atendente	Gizmo	prateleira
Area de Pedágio	Loja	Proprietário
Autopista	Luz vermelha	Registro de Pedágio
autenticação	Luz verde	Relógio
Banco	no de série	Resumo
banco autorizados	nome de acesso	Rodovia
Cabine de Entrada	número da área de pedágio	Segundos
Cancela	número da conta	Semáfaro
Cobrança	Painel	senha
código	Pedágio	Sensor de Saída
código da rodovia	Período	Sensor do Gizmo
código do gizmo	Pista	total a pagar
data / hora		Valor do Pedágio
		Veículo

Figura 11.4 – Lista de candidatos a Conceitos

A lista de candidatos a conceitos deve ser refinada, dando continuidade aos passos da seção 4.1.1, eliminando-se candidatos a conceitos que na verdade podem ser pensados como atributos de outros conceitos. Por exemplo, código, código da rodovia, código do gizmo, data/hora, nome de acesso, etc., são descartados da lista por serem atributos de conceitos que foram aprovados. A lista resultante de conceitos, após eliminados os falsos candidatos, é mostrada na Figura 11.5.

Analista Financeiro Atendente Área de Pedágio Autopista Banco Cabine de Entrada Cancela	Loja Gizmo Painel Pedágio Pista Proprietário Registro de Pedágio	Rodovia Semáfaro Sensor de Saída Sensor do Gizmo Valor do Pedágio Veículo

Figura 11.5 – Lista refinada de candidatos a Conceitos

Seguindo os passos da seção 4.1.1, pode-se tentar encontrar mais algum candidato a conceito por meio dos verbos presentes nos casos de uso que podem ser transformados em substantivos. Deve-se também eliminar possíveis redundâncias. Por exemplo, autopista e rodovia tem o mesmo significado, portanto escolhemos apenas um, rodovia no caso. Cabine de entrada representa fisicamente um elemento presente na área de pedágio, assim como uma cadeira ou uma mesa no sistema de biblioteca, portanto resolvemos eliminá-la, ou seja, não é um conceito que interfere na análise do problema. O mesmo ocorre para pista, que pode ser eliminada sem maiores conseqüências por não interferir na lógica do sistema. Apesar dos requisitos estabelecerem que ao entrar na área de pedágio o código da pista é informado ao sistema, analisando um pouco mais profundamente o glossário, verificamos que a área de pedágio engloba a pista, o sensor de gizmo, o sensor de saída, a cancela, o semáforo e o painel. Portanto, basta saber em que área de pedágio o veículo entrou e saiu para identificar corretamente qual é a localização (quilometragem) da entrada e da saída da autopista, e posteriormente calcular o valor a ser pago.

Loja representa o posto de atendimento no qual o gizmo é comprado, e resolvemos renomeá-la para Posto para ficar de acordo com o glossário. Incluímos mais um conceito, o de Concessionária, que engloba todos os postos de atendimento. Também renomeamos Registro de Pedágio para Registro de Uso, que é mais significativo, indicando que é registrado um uso efetivo da rodovia. Finalmente, a lista de conceitos obtida é mostrada na Figura 11.6.

Analista Financeiro Atendente Área de Pedágio Banco Cancela Concessionária	Posto Gizmo Painel Pedágio Proprietário Registro de Uso	Rodovia Semáfaro Sensor de Saída Sensor do Gizmo Valor do Pedágio Veículo

Figura 11.6 – Lista final de Conceitos

O próximo passo é identificar as associações entre os conceitos. O Modelo Conceitual resultante é mostrado na Figura 11.7. Em um primeiro momento não nos preocupamos em identificar as especializações, o que é feito no próximo refinamento, que resulta no modelo da Figura 11.8. Para melhor denotar que existem dois tipos de área de pedágio, a de entrada e a de saída, criamos uma superclasse APedágio e duas subclasses APEntrada e APSaída. Dessa forma, fica mais claro que um registro de uso da rodovia envolve sempre um carro passando por uma APEntrada e saindo por uma APSaída. Isso também facilita visualizarmos as diferenças entre APEntrada e APSaída.

Usamos herança também para denotar os diversos tipos de usuários envolvidos no sistema, que são: o atendente, o proprietário e o analista financeiro, o que permite melhor diferenciar as responsabilidades de cada um deles e implementar o controle de acesso no sistema, ou seja, controlar os papéis dos usuários que fazem uso do sistema, permitindo que executem apenas as tarefas para as quais tem permissão. Caso não fosse necessário o controle de acesso, esses usuários seriam simplesmente atores e não precisariam ser incluídos no modelo conceitual.

Usamos agregação em Área de Pedágio para denotar que Sensor do Gizmo, Sensor de Saída, Semáforo, Cancela e Painel são objetos fisicamente contidos na Área de Pedágio. Da mesma forma, a Rodovia agrega um ou mais pedágios, que por sua vez agregam uma ou mais áreas de pedágio.

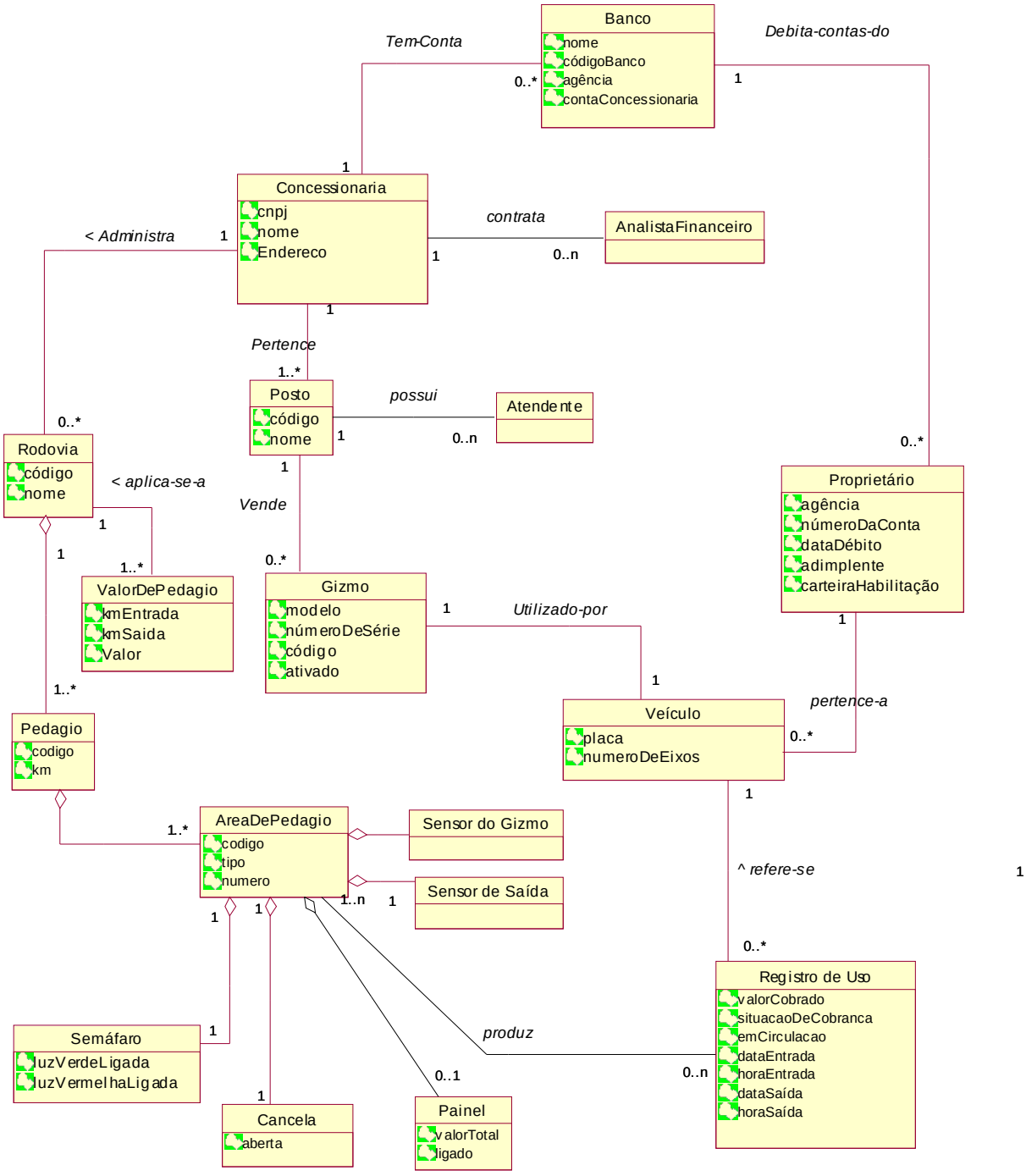


Figura 11.7 – Esboço 1 do Modelo Conceitual

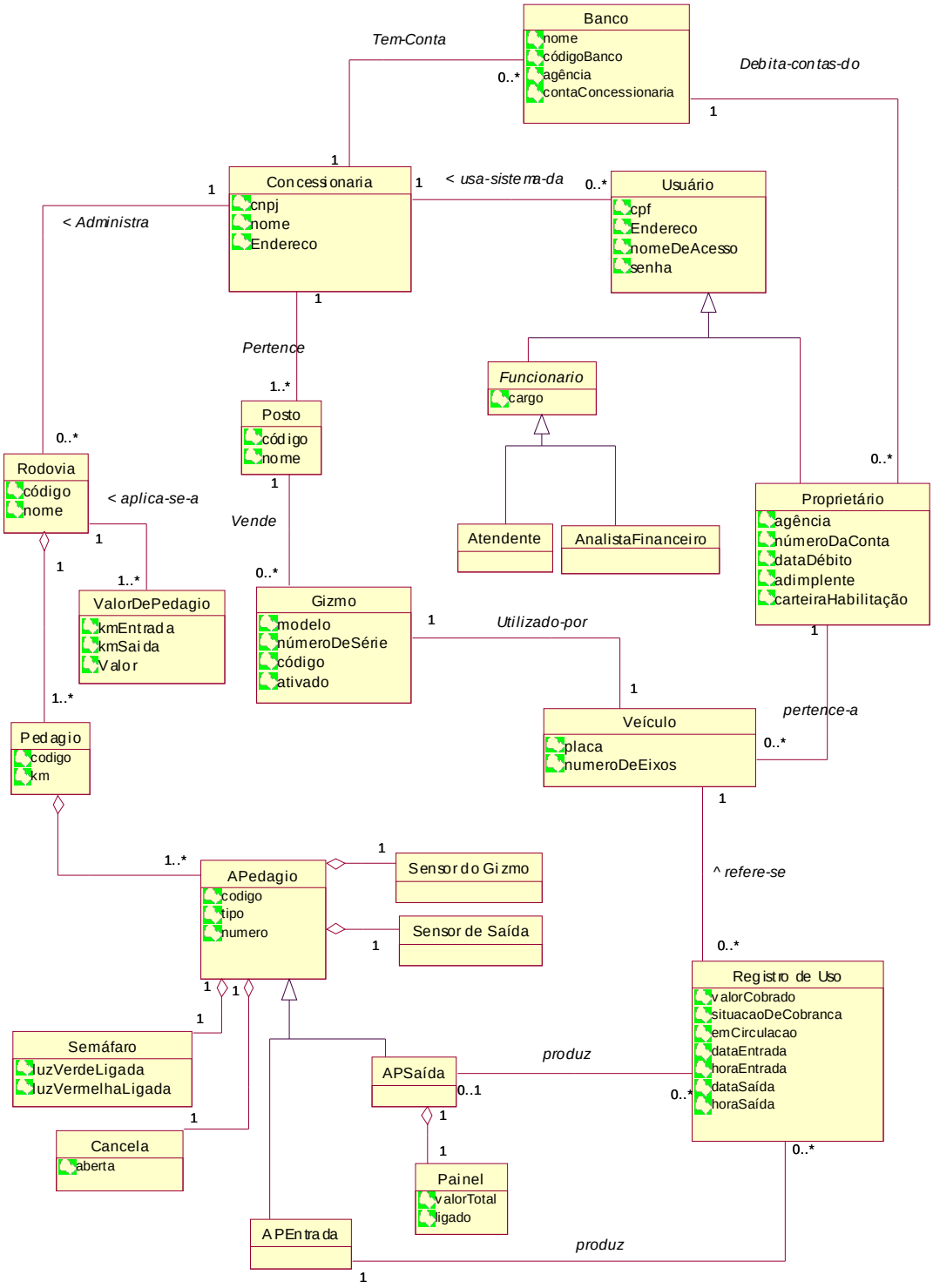


Figura 11.8 – Modelo Conceitual para o Sistema Passe Livre

11.3 – Diagramas de Seqüência do Sistema

Conforme foi visto na seção 5.1, cada caso de uso pode envolver diversas operações do sistema, que são denotadas nos Diagramas de Seqüência do Sistema (DSS). Assim, para cada um dos casos de uso do ciclo I do desenvolvimento do Sistema Passe Livre, deve ser elaborado um DSS correspondente. Ele deve ser feito com base no cenário de sucesso principal do caso de uso, e é uma das últimas atividades de análise OO. O DSS define quais serão as operações do sistema, que posteriormente deverão ser projetadas por meio de diagramas de colaboração. Ainda na fase de análise, pode-se fazer um contrato para cada uma dessas operações, o que é mostrado na seção 11.4.

A Figura 11.9 mostra o DSS para o caso de uso Autenticar Usuário. Como esse caso de uso é bastante simples, era de se esperar que seu DSS também o fosse. Nesse caso de uso há uma única interação entre o ator e o sistema, portanto uma única operação é suficiente para realizar a responsabilidade de autenticar o usuário. A resposta do sistema a essa operação é mostrada com a seta tracejada da Figura 11.9.

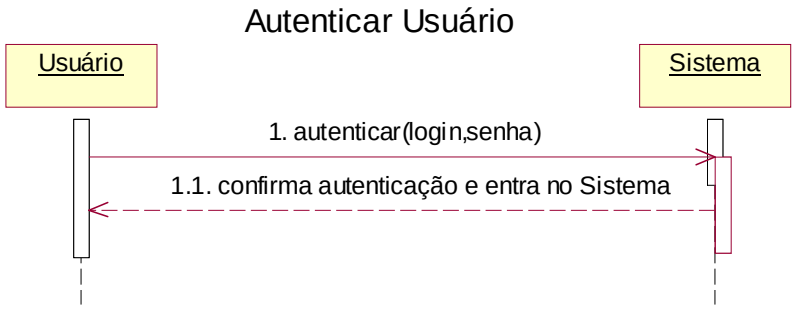


Figura 11.9 – DSS para Autenticar o Usuário

A Figura 11.10 mostra o DSS para um caso de uso mais complexo, o Entrar na Autopista, que é realizado por meio de duas operações, já que os atores (os sensores de gizmo e de saída do veículo) interagem com o sistema duas vezes: quando o gizmo é detectado, o que dispara várias tarefas necessárias para a autorização da entrada do veículo, e quando a saída do veículo é detectada pelo sensor de saída, o que dispara outras tarefas para concluir a entrada do veículo na autopista. Note que, apesar dessa descrição parecer confusa, estamos falando da **entrada do veículo na Autopista**, o que envolve que ele **entre e saia da Área de Pedágio**. Da mesma forma, no caso de uso Sair da Autopista, cujo DSS encontra-se no Apêndice C, o veículo entra e sai da Área de Pedágio.

Cada uma das operações do DSS envolve tarefas “ocultas”, já que no DSS o sistema é representado simplesmente como uma caixa-preta. Posteriormente, na fase de projeto, esta caixa-preta será aberta e saberemos quais os objetos que trocam mensagens para realizar a responsabilidade de registrar a entrada do veículo.

Note que os DSSs apresentados nas figuras 11.9 e 11.10 representam o cenário de sucesso principal. Se quisermos apresentar os cenários de insucesso, devemos criar outros DSSs

para cada um dos fluxos alternativos do caso de uso. Em geral, recomenda-se que sejam criados DSSs apenas para os cenários mais complexos, que possam envolver várias operações.

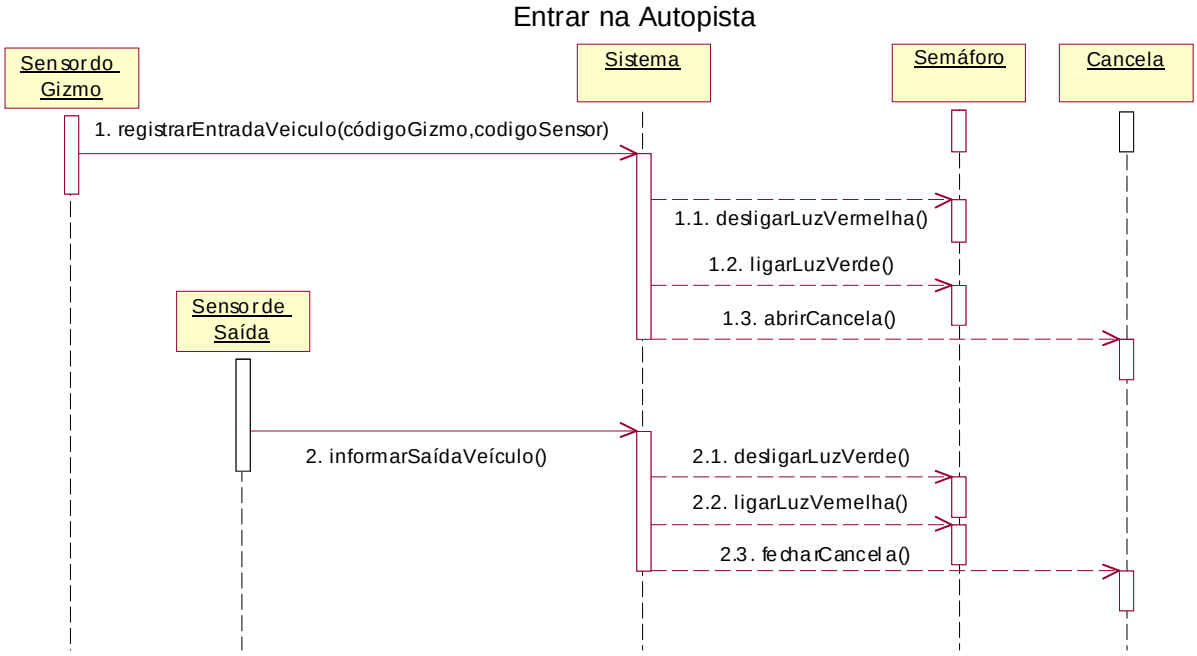


Figura 11.10 – DSS para Entrar na autopista

11.4 – Contratos das Operações

Para encerrar a fase de análise do sistema, cada operação identificada nos DSSs deve possuir um contrato especificando suas responsabilidades, conforme foi visto na Seção 5.2. As Figuras 11.11, 11.12 e 11.13 ilustram alguns contratos para os DSSs das Figuras 11.9 e 11.10. Os demais DSSs e contratos encontram-se no Apêndice C. Note que o contrato da operação *autenticar* (Figura 11.11) não contém pós nem pré-condições, porque trata-se de uma operação que não altera o estado do sistema, isto é, não se criam novos objetos, associações, não se removem objetos ou associações existentes, não se modificam valores de atributos, etc. Portanto, não seria necessário fazer este contrato. Se o sistema possuísse um controle de acesso mais rigoroso, então esta operação poderia mudar o estado do sistema, por exemplo, seria criado um registro contendo a data e hora do acesso, seria registrado se a senha foi digitada corretamente, etc. Já os contratos das operações *registrarEntradaVeiculo* e *informarSaidaVeiculo* são mais complexos, e suas pré e pós-condições informam o que é esperado da operação em termos de mudança no estado do sistema para que o projetista, na fase seguinte, possa projetar a colaboração entre os objetos para cumprir satisfatoriamente o que foi "prometido" no contrato.

Contrato: Autenticar
Operação: autenticar(login, senha)
Referências Cruzadas: Caso de Uso: Autenticar Usuário
Pré-Condições: -
Pós-Condições: -

Figura 11.11 – Contrato para a operação autenticar

Contrato: Registrar Entrada Veículo
Operação: registrarEntradaVeículo(codigoGizmo)
Referências Cruzadas: Caso de Uso: Entrar na Autopista
Pré-Condições: Um veículo na área de pedágio de entrada teve seu Gizmo identificado pelo Sensor de Gizmo. O Gizmo estava habilitado no Sistema e o proprietário era adimplente.
Pós-Condições: - Foi criada uma nova instância reg de Registro_de_Uso.

- reg.emCirculacao tornou-se true
- reg.data_entrada recebeu a data do Sistema.
- reg.hora_entrada recebeu a hora do Sistema.
- reg foi associada a uma instância do Veículo.
- reg foi associada a uma instância da APEntrada.
- Semáforo.luzVerdeLigada recebeu true
- Semáforo.luzVermelhaLigada recebeu false
- Cancela.aberta recebeu true

Figura 11.12 – Contrato para a operação registrarEntradaVeículo

Contrato: Informar Saída Veículo
Operação: informarSaídaVeículo(códigoGizmo)
Referências Cruzadas: Caso de Uso: Entrar da Autopista
Pré-Condições: Um veículo na saída da autopista teve seu Gizmo identificado pelo Sensor de Gizmo.
Pós-Condições:

- cancela.aberta tornou-se true
- semaforo.luzVerdeLigada tornou-se false
- semaforo.luzVermelhaLigada tornou-se true

Figura 11.13 – Contrato para a operação informarSaídaVeículo

11.5 – Diagrama de Estados

Conforme visto na seção 5.3, um diagrama de estados pode ser aplicado a diversos elementos dos modelos OO, entre os quais: conceitos (do modelo conceitual), classes de software, casos de uso e o próprio sistema. Nesta seção, usaremos diagramas de estado para melhor compreender a dinâmica de alguns conceitos do modelo conceitual, a saber: o gizmo, o registro de uso da autopista, o veículo, as áreas de entrada e saída do pedágio (APEntrada e APSaída), a cancela e o semáforo, ilustrados nas Figuras 11.14 a 11.20. Alguns estados podem ser modelados, mesmo estando fora do escopo do sistema, para dar uma visão geral do domínio. Por exemplo, no diagrama de estados do gizmo, o estado

“fabricado” não faz parte dos limites do sistema, pois o gizmo só passa a ser de interesse do sistema a partir do momento que for adquirido pelo posto da concessionária.

Os diagramas de Cancela, Semáforo, APEntrada, APSaída e Veículo funcionam em paralelo e que poderiam ser agrupados em um statechart. Os Statecharts estendem o formalismo convencional das Máquinas de Estados Finitos (MEFs) de forma que, além da noção de estados e transições, definem-se hierarquia de estados, concorrência ou ortogonalidade entre estados e um mecanismo de propagação de eventos (*broadcast*) que possibilita a comunicação entre estados. Embora não abordemos statecharts neste livro, o leitor interessado pode consultar Harel (1987).

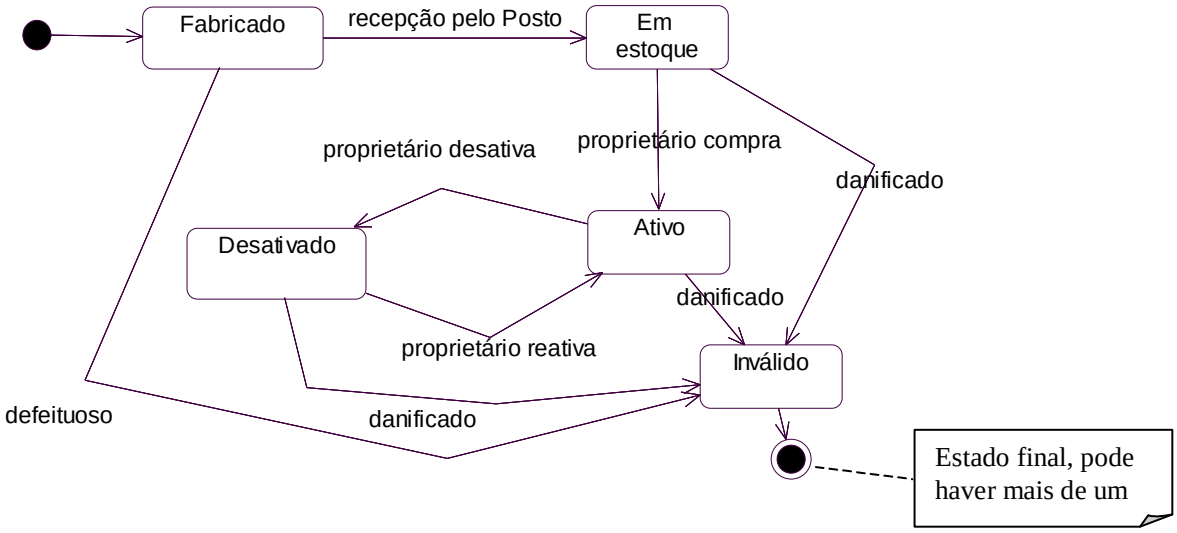


Figura 11.14 –Diagrama de Estados do Gizmo

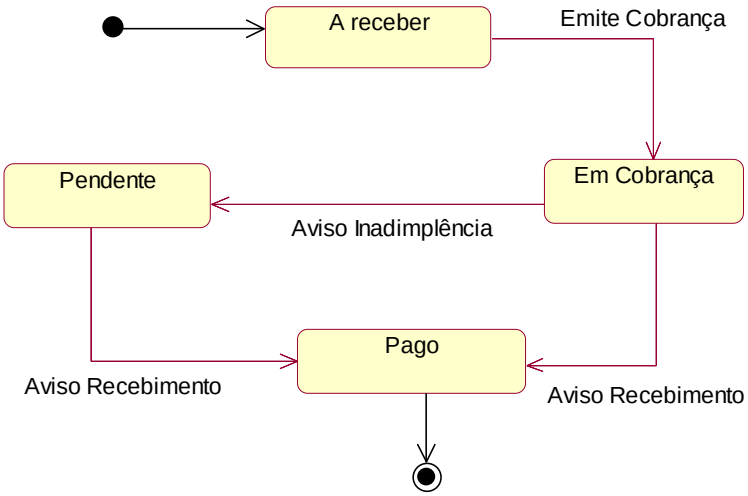


Figura 11.15 –Diagrama de Estados do Registro de Uso

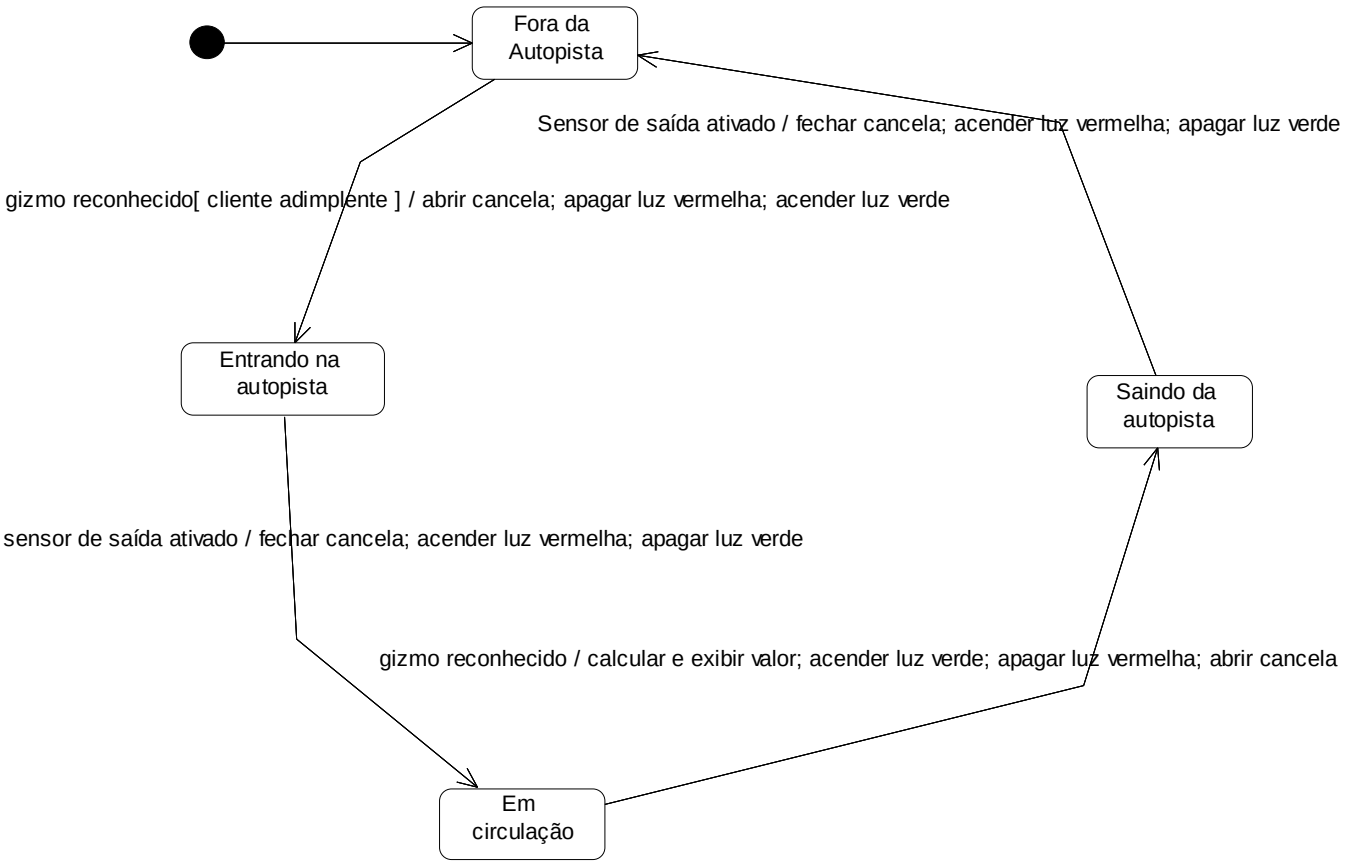


Figura 11.16 –Diagrama de Estados de Veículo

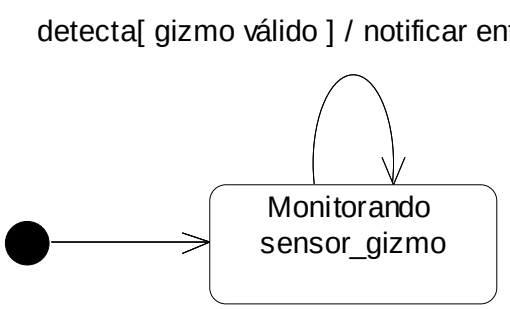


Figura 11.17 –Diagrama de Estados de APEntrada

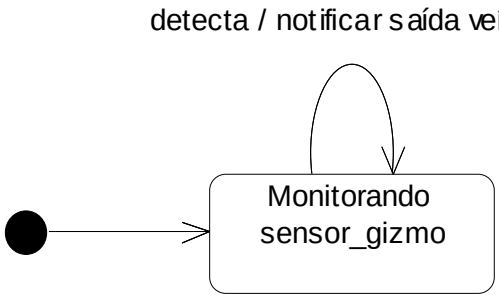


Figura 11.18 –Diagrama de Estados de APSaída

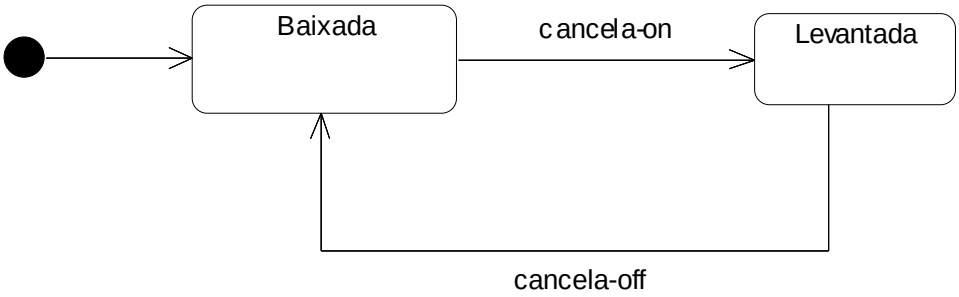


Figura 11.19 –Diagrama de Estados de Cancela

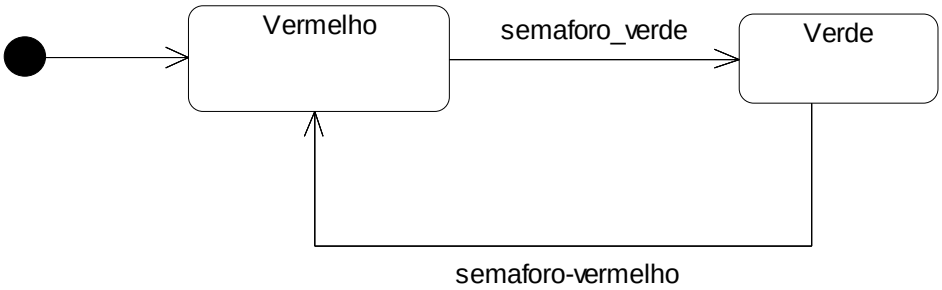


Figura 11.20 –Diagrama de Estados de Semáforo

Capítulo 12 – Ciclo de Desenvolvimento 1 do Sistema Passe Livre – Projeto e Implementação Fase de Requisitos

12.1 – Diagramas de Colaboração

No módulo 11 apresentamos os principais artefatos referentes à fase de elaboração do PU para o ciclo 1 do Sistema Passe Livre. Dentre os artefatos apresentados estavam o modelo conceitual, os casos de uso no formato completo abstrato, o diagrama de seqüência do sistema para cada caso de uso e contrato das operações. Agora, seguindo o PU, vamos iniciar a construção, iniciando pelo detalhamento do projeto das operações. Para isso, precisamos tomar decisões de projeto que envolvem usar padrões de atribuição de responsabilidades – os padrões GRASP – vistos na Seção 6.3.

Vamos iniciar o projeto OO com a operação *autenticar Usuário*, identificada quando construímos o DSS para o caso de uso Autenticar Usuário (Figura 11.9). Os dados de entrada para esta operação são o nome do usuário e sua senha. O sistema deve validar essa entrada e criar um objeto referente ao usuário. Como temos vários tipos de usuário possíveis (atendente, proprietário, analista financeiro), o sistema deve retornar o objeto de uma das classes correspondentes. Vamos incluir também um tipo de usuário adicional, o administrador, que será responsável por gerenciar quaisquer arquivos do sistema e, portanto, precisa ter todo o controle do sistema, ou seja, tem permissão de fazer quaisquer operações no sistema.

Para projetar o diagrama de colaboração da operação *autenticar Usuário*, a primeira decisão é quem será a classe controladora desta operação. Usando o padrão GRASP Controlador, temos duas opções: criar uma classe artificial por caso de uso (por exemplo ControladorDeAutenticarUsuario) ou usar uma classe do domínio que seja relacionada a esta funcionalidade. Optamos pela segunda opção e assim vamos atribuir a operação autenticar Usuário à classe Concessionária. Portanto, consideramos que haverá um objeto da classe Concessionária, que receberá a invocação desta operação e terá que retornar um objeto de uma das classes da classe Usuário: Atendente, Proprietário, Analista Financeiro ou Administrador, dependendo do tipo do usuário que forneceu seu nome e senha. A princípio, podemos simplesmente mostrar um diagrama de colaboração menos detalhado, como o da Figura 12.1, sem nos preocuparmos com o "como" este objeto será retornado. O diagrama proposto simplesmente obtém o usuário a partir da coleção de todos os usuários do sistema, e retorna um booleano que indica se obteve sucesso na autenticão. Voltaremos a esse assunto mais adiante.

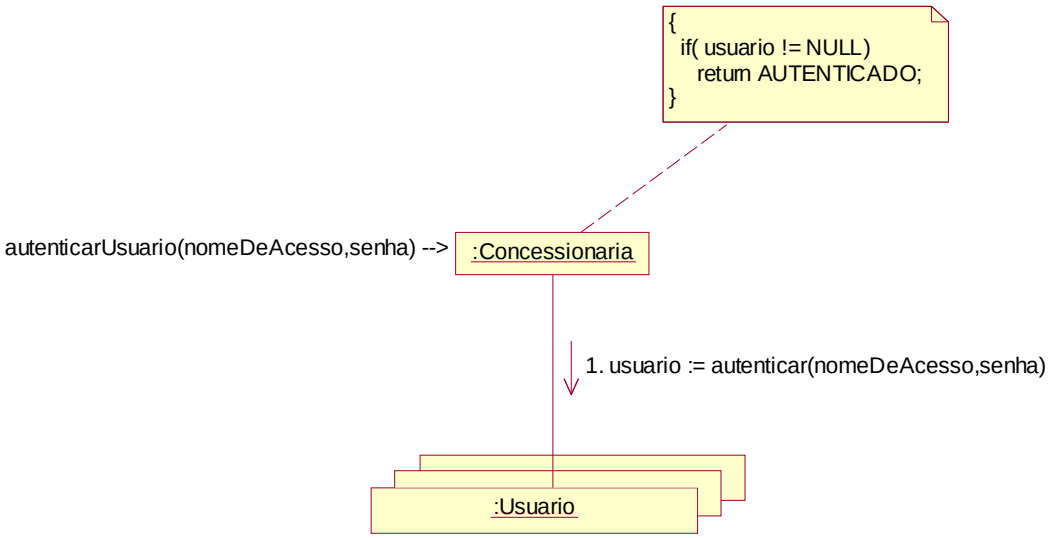


Figura 12.1 – Diagrama de colaboração para autenticar usuário

Para detalhar ainda mais o projeto das colaborações entre os objetos, temos que nos adiantar um pouco no projeto e decidir como será nossa base de dados. Se ela for orientada a objetos, deverá haver um mecanismo pelo qual, dado o nome do usuário, seu identificador seja descoberto e o objeto da subclasse seja recuperado. Se ela for relacional, temos várias possibilidades para tratar o problema de herança.

A primeira alternativa é criar uma única tabela para representar todas as subclasses. Esta tabela conteria todos os atributos da superclasse e das subclasses. Portanto, para recuperar um dado usuário, seja ele de qualquer tipo, basta encontrar o registro correspondente ao seu nome de acesso nessa tabela única. O tipo de usuário poderia ser simplesmente um atributo contido em uma das colunas desta tabela.

A segunda alternativa é criar uma tabela, *Usuário*, para representar as partes comuns a todos os tipos de usuário. Esta tabela conteria colunas correspondentes a cada um dos atributos da superclasse Usuário, além de duas colunas adicionas: uma para identificar unicamente o usuário (*idUsuario*) e outra para definir o tipo de usuário (*tipoUs*). Seriam também criadas tabelas para cada uma das subclasses, com uma chave estrangeira ligando-a à tabela Usuário. Dessa forma, recuperados os dados principais do usuário por meio de seu nome de acesso, é possível recuperar os dados específicos de cada tipo de usuário abrindo a tabela correspondente.

Vamos adotar a segunda alternativa, ou seja, teremos cinco tabelas conforme o esquema da Figura 12.2. Assim, o diagrama de colaboração será como o da Figura 12.3. Recuperado o tipo do usuário na coleção de todos os usuários, pode-se criar o objeto da subclasse apropriada invocando-se a busca ao objeto específico na respectiva coleção.

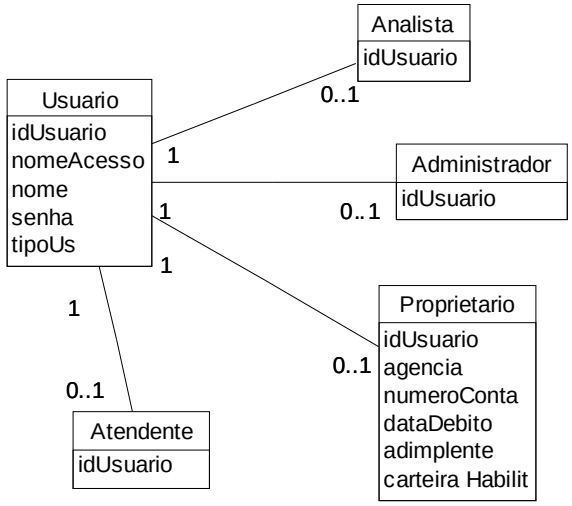


Figura 12.2 – Esquema da base de dados relacional para Usuário

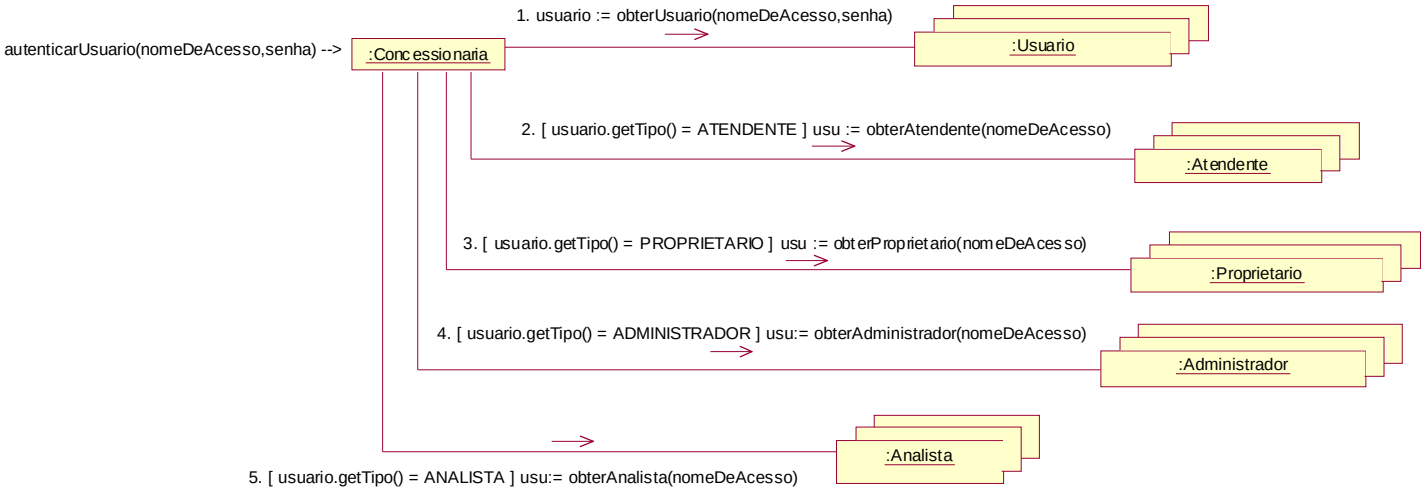


Figura 12.3 – Diagrama de colaboração para autenticar usuário (versão 2)

Vamos agora projetar o diagrama de colaboração para uma operação mais complexa: a *registrarEntradaVeiculo*, que surgiu no DSS para o caso de uso Entrar na Autopista (Figura 11.10). De acordo com o contrato desta operação (Figura 11.12), as seguintes pré-condições devem ser satisfeitas para que ela possa acontecer: a) um veículo na área de pedágio de entrada teve seu Gizmo identificado pelo Sensor de Gizmo; b) o gizmo estava habilitado no Sistema; e c) o proprietário era adimplente. Ainda, de acordo com o contrato, as seguintes pós-condições devem vigorar após sua invocação: a) foi criada uma nova instância reg de Registro_de_Uso; b) reg.emCirculacao tornou-se true; c) reg.data_entrada recebeu a data do Sistema; d) reg.hora_entrada recebeu a hora do Sistema; e) reg foi associada a uma instância do Veículo; f) reg foi associada a uma instância da APEntrada; g)

Semáforo.luzVerdeLigada recebeu true; h) Semáforo.luzVermelhaLigada recebeu false; e i) Cancela.aberta recebeu true.

Se analisarmos o DSS para o caso de uso, podemos ver que para garantir as pré-condições b) e c) deveria haver uma outra operação com essa responsabilidade. Como não há, concluímos que o contrato deve ser corrigido, e a própria operação *registrarEntradaVeiculo* deve fazer essas verificações.

Portanto, vamos partir do princípio que o sensor do gizmo identificou o gizmo presente no veículo, assim como vamos supor que o sensor conhece seu próprio código e pode passá-lo como parâmetro para a operação. Portanto, nossa primeira preocupação é verificar a validade do gizmo (se ele existe e está ativado no momento) e a adimplência de seu proprietário. A validade do gizmo pode ser feita pelo próprio objeto Gizmo, que contém um atributo *ativado*. Portanto, se existir um gizmo cujo código seja o detectado pelo sensor, e se seu atributo *ativado* for *true*, o gizmo é válido. Podemos pensar então em um método *habilitado()*, enviado ao objeto gizmo, que retorna um booleano *true* ou *false*. Mas que classe terá a responsabilidade de enviar esta chamada ao gizmo? Temos que pensar em termos de visibilidade: que classe tem visibilidade para a classe Gizmo? Posto está ligada a Gizmo, bem como Veículo está ligada a Gizmo. Mas no momento que o veículo entra na área de pedágio, ainda não sabemos qual é o veículo, portanto não temos o objeto *veiculo* ainda. Podemos considerar que uma instância de Concessionária está sempre presente no sistema em execução. A concessionária tem visibilidade para os postos, que por sua vez tem visibilidade para os gizmos lá adquiridos.

Logo, poderíamos delegar a busca do gizmo para cada um dos postos, até encontrar o gizmo. Porém, essa solução poderia se tornar um gargalo do sistema, já que esse método é executado muitas vezes, portanto precisamos de uma solução otimizada. Uma possível solução seria incluir uma associação com visibilidade por atributo, partindo de Concessionária para Gizmo. Essa é uma decisão de projeto, em que aumentaríamos o acoplamento para melhorar o desempenho do sistema e, por isso, vamos adotá-la. Assim, definimos Concessionária como sendo a classe controladora para a operação *registrarEntradaVeiculo*.

O diagrama de colaboração da Figura 12.4 mostra as interações necessárias para verificar se o gizmo está habilitado. É feita uma consulta na coleção de gizmos para recupear o gizmo cujo código seja o passado como parâmetro para a operação. Se ele for encontrado, seu método *habilitado*() é invocado. De posse dessa informação, podemos prosseguir para verificar a adimplência do proprietário.

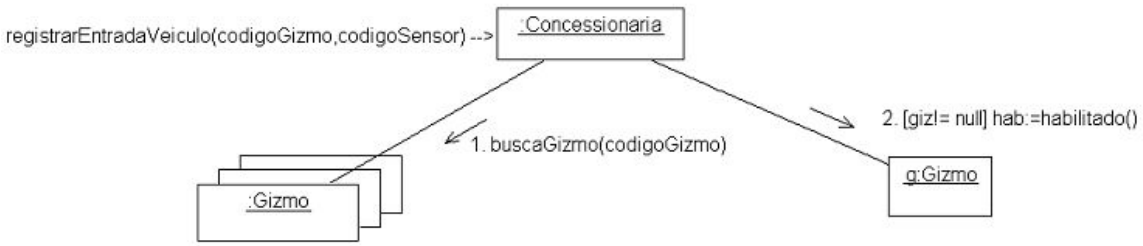


Figura 12.4 – Diagrama de colaboração parcial para registrarEntradaVeiculo (v1)

No diagrama de colaboração da Figura 12.5 são acrescidas as mensagens para verificar se o proprietário é ou não adimplente. Considera-se que o gizmo conhece o veículo ao qual pertence, bem como o veículo sabe quem é seu proprietário. Portanto a mensagem é delegada até chegar ao proprietário que, por possuir um atributo *adimplente*, pode imediatamente responder à mensagem.

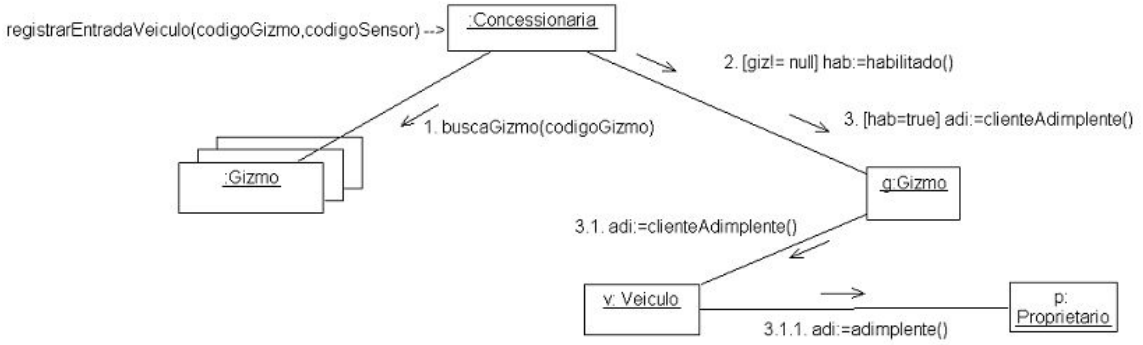


Figura 12.5 – Diagrama de colaboração parcial para registrarEntradaVeiculo (v2)

Agora, é necessário projetar as mensagens para que sejam cumpridas as pós-condições, começando pela a) foi criada uma nova instância *reg* de Registro_de_Uso. Esta condição implica que seja criada uma instância de Registro de Uso para representar o uso da autopista pelo veículo. Como estamos na entrada da autopista, queremos associar o registro de uso a uma APEntrada e ao veículo correspondente. Note que as próximas pós-condições (b, c, d, e, f) têm relação com a criação desse registro de uso. Conforme discutido na seção 6.3.4, pode-se criar a instância e depois invocar métodos para atribuir valores aos atributos e/ou fazer as associações, ou pode-se invocar a criação da instância já passando como parâmetros todos os valores necessários.

Vamos optar por essa segunda alternativa, ou seja, primeiramente vamos obter os objetos e valores necessários, para depois criar a instância. Os valores dos atributos são fixos ou obtidos do próprio sistema, ou seja, o valor do atributo *emCirculacao* é true e os valores dos atributos *data_entrada* e *hora_entrada* são obtidos do próprio sistema. Precisamos então obter o veículo e a área de pedágio de entrada, para cumprirmos as pós-condições ***e*** e ***f***, respectivamente. O objeto veículo é conhecido pelo objeto gizmo, portanto não é necessário obtê-lo. A APEntrada pode ser obtida via autopista e pedágio, conforme ilustrado na Figura 12.6. Com isso, tem-se em mãos todos os parâmetros para que possa ser

invocada a criação do registro de uso. A classe mais apropriada para criar o registro de uso, segundo o padrão Criador, é a Veiculo, já que um veículo deve ter visibilidade para seus registros de uso.

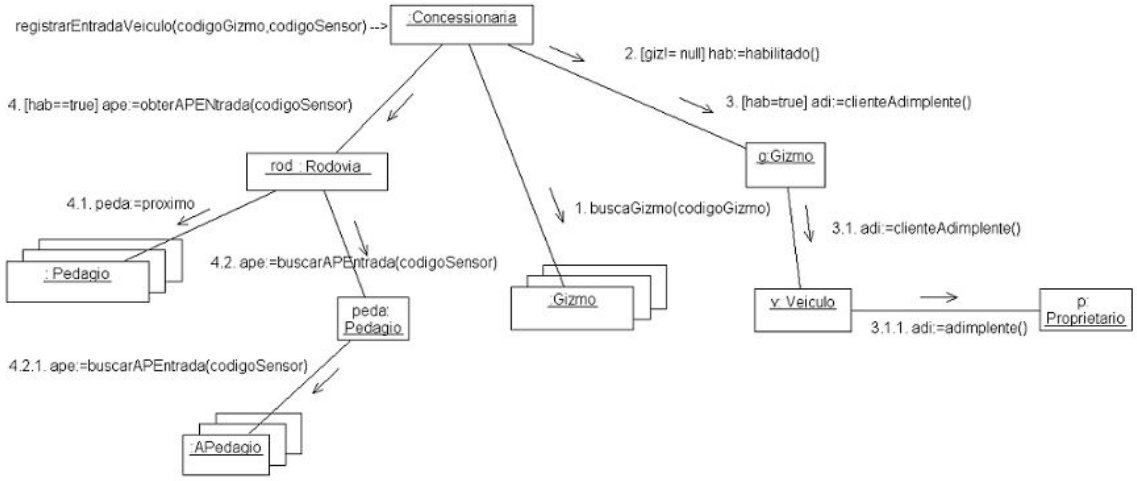


Figura 12.6 – Diagrama de colaboração parcial para registrarEntradaVeiculo (v3)

Finalmente, para cumprir as responsabilidades referentes às pós-condições ***g***, ***h*** e ***i,*** deve-se enviar mensagens à Cancela e ao Semáforo, o que pode ser delegado à classe APEntrada, por estar ligada a essas classes no modelo conceitual. O diagrama da Figura 12.7 mostra o diagrama de colaboração final para a operação *registrarEntradaVeiculo*.

Vale relembrar que, como o projeto de um diagrama de colaboração envolve elementos de criatividade e experiência do projetista, a solução não é única. Ao contrário, muitas outras soluções, similares, piores ou até melhores podem ser propostas para representar o projeto desta operação. O que se deve analisar, basicamente, é a vantagem de outras soluções em termos dos padrões Acoplamento Fraco e Coesão Alta, comentados na Seção 6.3.

O Apêndice D contém diversos outros exemplos de diagramas de colaboração para o ciclo I de desenvolvimento do Sistema Passe Livre.

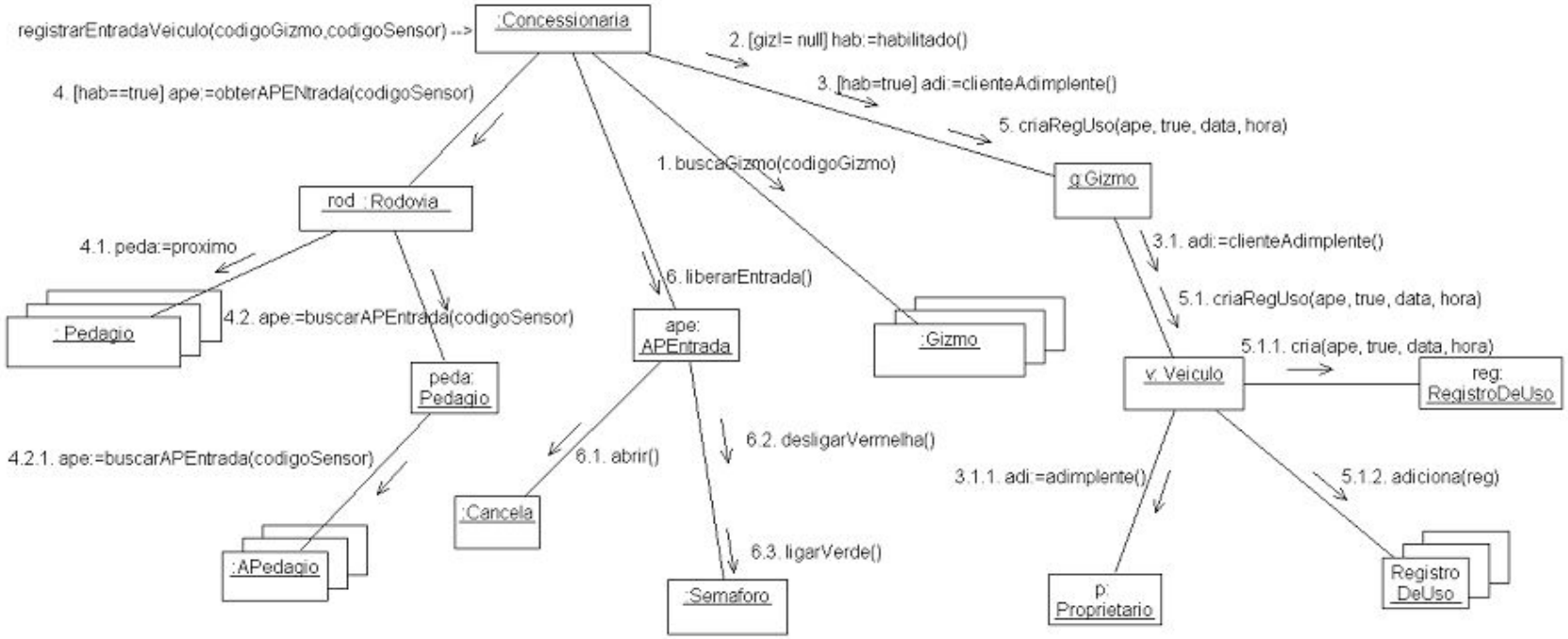


Figura 12.7 – Diagrama de colaboração final para registrarEntradaVeiculo

12.2 – Diagrama de Classes de Projeto

Conforme visto na Seção 7.3, o diagrama de classes de projeto é obtido basicamente a partir do modelo conceitual e dos diagramas de colaboração. Uma classe deve ser criada para cada um dos conceitos do Modelo Conceitual para os quais houverem mensagens sendo recebidas em pelo menos um diagrama de colaboração. As associações entre as classes podem ser derivadas do modelo conceitual acrescentando, se necessário, associações percebidas durante o projeto dos diagramas de colaboração. A visibilidade entre classes deve ser determinada, seguindo os conceitos abordados na seção 7.2.

As mensagens devem ser incluídas nas respectivas classes que a recebem. De acordo com o que já foi dito na seção 7.3, deve-se evitar incluir mensagens de atribuição e recuperação de valor de atributo, para não poluir demasiadamente o modelo. Além disso, não devem ser incluídas as mensagens enviadas a coleções de objetos, pois essas mensagens farão parte da infra-estrutura da linguagem de programação. A Figura 12.8 ilustra parte do diagrama de classes do Sistema Passe Livre, considerando apenas os diagramas de colaboração das Figuras 12.3 e 12.7. O diagrama completo está no Apêndice D.

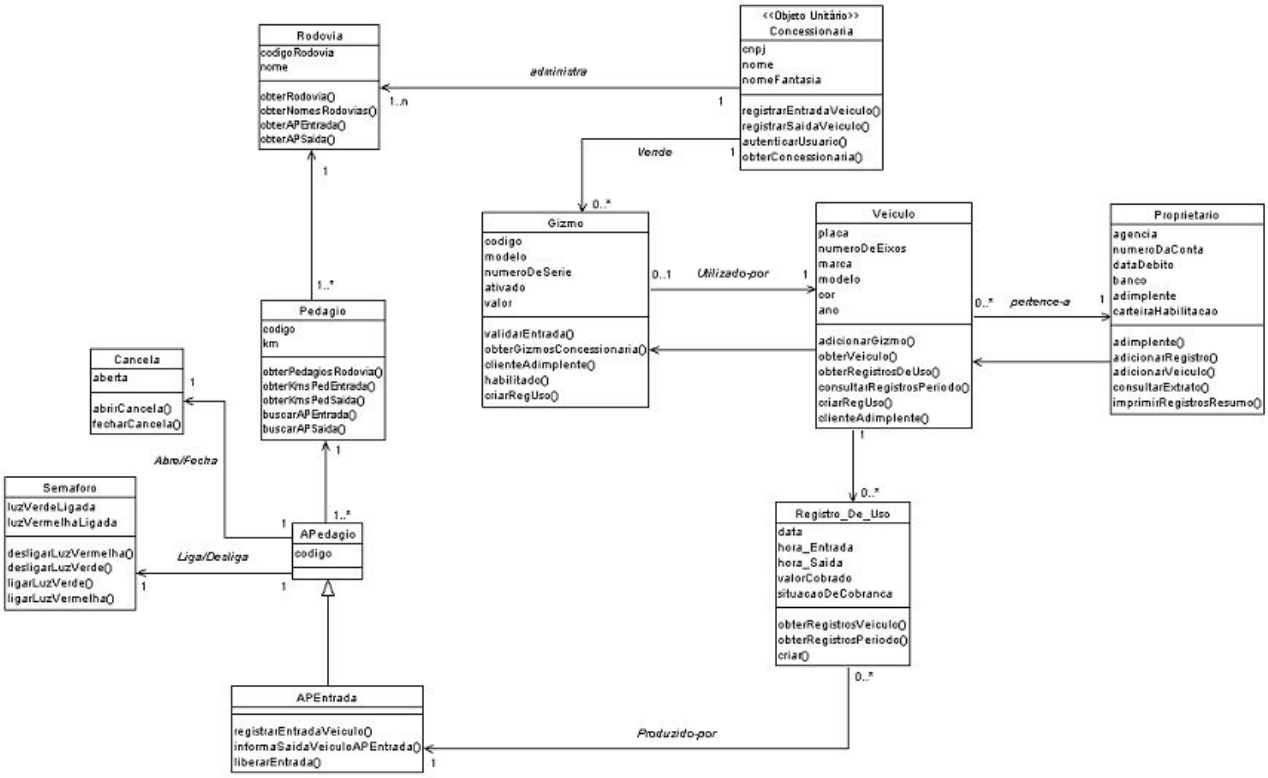


Figura 12.8 –Diagrama de Classes de Projeto (parcial) para o Sistema Passe Livre

12.3 – Exemplos de código-fonte

Os trechos de código a seguir ilustram como o projeto apresentado nas seções anteriores pode ser implementado usando a linguagem de programação Java. O código aqui apresentado é implementado com arquitetura Web, usando frameworks, o banco de dados MySQL e padrões de projeto. Para entender esse código, são necessários mais alguns conceitos sobre padrões, principalmente alguns padrões J2EE, que farão parte de um manual a ser produzido em breve.

Também, no Anexo E, pode-se encontrar outros exemplos de código implementado usando a linguagem Java. P, porém são bastante simples, não fazendo a divisão em três camadas, mas em apenas duas (ou seja, não se isola a camada de persistência, mas somente a camada de interface com o usuário). O Anexo E contém as classes que são necessárias para implementar o caso de uso 'Autenticar Usuário'. E, essas classes podem ser compiladas e executadas em máquinas com a máquina virtual Java. No anexo E também está disponibilizado um guia de como instalar e executar o código.

12.3.1 – O Código

Um dos padrões utilizados nestses códigos tem o nome de DAO ou *Data Access Object*. Ele um padrão J2EE altamente difundido na comunidade Java. Basicamente, o problema que ele resolve é de como permitir acesso ao armazenamento persistente, como um banco de dados ou arquivo XML, de maneira que uma variação nessa fonte de dados não interfira diretamente na camada da lógica de negócios do sistema.

Então Neste padrão, as classes são divididas em BO e TO. Os objetos BO (*Business Object*) representa o objeto que requer acesso à origem de dados para obter e armazenar dados. Os objetos TO (*Transfer Object* ou *Value Object*) representa um objeto de dados utilizado apenas como um transportador de dados (outro padrão J2EE).

Vamos mostrar, a partir de agora, como seria implementada a classe Usuário usando o padrão DAO. Mostraremos também um pouco da implementação da classe Proprietário para ilustrar o uso da herança.

- A classe UsuarioTO

```
package passeLivre.model.to;

import java.io.Serializable;

public class UsuarioTO implements Serializable {

	public final static int PROPRIETARIO = 1;
	public final static int ATENDENTE = 2;
	public final static int ANALISTA = 3;
	public final static int ADMINISTRADOR = 4;
	public final static String USUARIO = "USUARIO";

	int tipo;
	String nomeDeAcesso;
```

```
String senha;
String cpf;
EnderecoTO endereco;
boolean novo;
String nome;


/**
 * @return
 */
public String getCpf() {
	return cpf;
}

/**
 * @return
 */
public EnderecoTO getEndereco() {
	return endereco;
}

/**
 * @return
 */
public String getNomeDeAcesso() {
	return nomeDeAcesso;
}

/**
 * @return
 */
public String getSenha() {
	return senha;
}

/**
 * @return
 */
public int getTipo() {
	return tipo;
}

/**
 * @param string
 */
public void setCpf(String string) {
	cpf = string;
}

/**
 * @param enderecoTO
 */
public void setEndereco(EnderecoTO enderecoTO) {
	endereco = enderecoTO;
}

/**
 * @param string
 */
public void setNomeDeAcesso(String string) {
	nomeDeAcesso = string;
}
```

```
	/**
	 * @param string
	 */
	public void setSenha(String string) {
		senha = string;
	}

	/**
	 * @param i
	 */
	public void setTipo(int i) {
		tipo = i;
	}

	/**
	 * @return
	 */
	public boolean isNovo() {
		return novo;
	}

	/**
	 * @param b
	 */
	public void setNovo(boolean b) {
		novo = b;
	}

	/**
	 * @return
	 */
	public String getNome() {
		return nome;
	}

	/**
	 * @param string
	 */
	public void setNome(String string) {
		nome = string;
	}

}
```

- A classe UsuarioBO

```
package passeLivre.model.bo;

import passeLivre.db.general.DAOFactory;
import passeLivre.db.general.UsuarioDAO;
import passeLivre.model.to.UsuarioTO;

public class UsuarioBO {

	public UsuarioTO obterUsuario( String nomeDeAcesso ) {

		DAOFactory mySQLDAOFactory = DAOFactory.getDAOFactory(
DAOFactory.MYSQL );
		UsuarioDAO mySQLUsuarioDAO = mySQLDAOFactory.getUsuarioDAO();

		return mySQLUsuarioDAO.obterUsuario( nomeDeAcesso );
	}
```

```
	public boolean atualizarUsuario( UsuarioTO usuarioTO ) {

		DAOFactory mySQLDAOFactory = DAOFactory.getDAOFactory(
DAOFactory.MYSQL );
		UsuarioDAO mySQLUsuarioDAO = mySQLDAOFactory.getUsuarioDAO();

		return mySQLUsuarioDAO.atualizarUsuario( usuarioTO );
	}

	public boolean alterarSenha( UsuarioTO usuarioTO, String nomeDeAcesso,
String senha ) {

		if ( usuarioTO.getNomeDeAcesso().equals( nomeDeAcesso ) ) {

			usuarioTO.setSenha( senha );
			usuarioTO.setNovo( false );

			// atualiza usuário na fonte de dados
			return atualizarUsuario( usuarioTO );
		}
		else {

			return false;
		}
	}

	public UsuarioTO autenticarUsuario( String nomeDeAcesso, String senha ) {

		UsuarioTO usuarioTO = obterUsuario( nomeDeAcesso );

		if ( usuarioTO.getSenha().equals( senha ) ) {

			return usuarioTO;
		}
		else {

			return null;
		}
	}

	public boolean verificaNomeDeAcesso( String nomeDeAcesso ) {

		UsuarioTO usuarioTO = obterUsuario( nomeDeAcesso );

		if ( usuarioTO != null ) {

			return true;
		}
		else {

			return false;
		}
	}
}
```

- A classe ProprietarioTO

```
package passeLivre.model.to;

import java.io.Serializable;
import java.util.ArrayList;
```

```
public class ProprietarioTO extends UsuarioTO implements Serializable {

	BancoTO banco;
	ArrayList veiculos;

	String carteiraHabilitacao;
	boolean adimplente;
	int dataDebito;
	String numeroDaConta;
	String agencia;


	/**
	 * @return
	 */
	public boolean isAdimplente() {
		return adimplente;
	}

	/**
	 * @return
	 */
	public String getAgencia() {
		return agencia;
	}

	/**
	 * @return
	 */
	public BancoTO getBanco() {
		return banco;
	}

	/**
	 * @return
	 */
	public String getCarteiraHabilitacao() {
		return carteiraHabilitacao;
	}

	/**
	 * @return
	 */
	public int getDataDebito() {
		return dataDebito;
	}

	/**
	 * @return
	 */
	public String getNumeroDaConta() {
		return numeroDaConta;
	}

	/**
	 * @return
	 */
	public ArrayList getVeiculos() {
		return veiculos;
	}

	/**
```

```
	 * @param b
	 */
	public void setAdimplente(boolean b) {
		adimplente = b;
	}

	/**
	 * @param string
	 */
	public void setAgencia(String string) {
		agencia = string;
	}

	/**
	 * @param bancoTO
	 */
	public void setBanco(BancoTO bancoTO) {
		banco = bancoTO;
	}

	/**
	 * @param string
	 */
	public void setCarteiraHabilitacao(String string) {
		carteiraHabilitacao = string;
	}

	/**
	 * @param i
	 */
	public void setDataDebito(int i) {
		dataDebito = i;
	}

	/**
	 * @param string
	 */
	public void setNumeroDaConta(String string) {
		numeroDaConta = string;
	}

	/**
	 * @param list
	 */
	public void setVeiculos(ArrayList list) {
		veiculos = list;
	}

}
```

- A classe ProprietarioBO

```
package passeLivre.model.bo;

import java.util.ArrayList;

import passeLivre.db.general.DAOFactory;
import passeLivre.db.general.ProprietarioDAO;
import passeLivre.model.to.ProprietarioTO;

public class ProprietarioBO extends UsuarioBO {

	public ArrayList obterVeiculosProprietario( String nomeDeAcesso ) {
```

```
            VeiculoBO veiculoBO = new VeiculoBO();

            // obtém todos os veículos do proprietario
            ArrayList veiculos = veiculoBO.encontrarVeiculos( nomeDeAcesso );

            return veiculos;
      }

      public ProprietarioTO obterProprietario( String nomeDeAcesso ) {

            DAOFactory mySQLDAOFactory = DAOFactory.getDAOFactory(
DAOFactory.MYSQL );
            ProprietarioDAO mySQLProprietarioDAO =
mySQLDAOFactory.getProprietarioDAO();

            // obtém os dados do proprietário
            ProprietarioTO propTO = mySQLProprietarioDAO.obterProprietario(
nomeDeAcesso );

            if( propTO != null ) {

                  EnderecoBO endBO = new EnderecoBO();

                  // obtém o endereço do proprietário
                  propTO.setEndereco( endBO.obterEnderecoUsuario(
propTO.getNomeDeAcesso() ) );
            }

            return propTO;
      }

      public boolean inserirProprietario( ProprietarioTO propTO ) {

            DAOFactory mySQLDAOFactory = DAOFactory.getDAOFactory(
DAOFactory.MYSQL );
            ProprietarioDAO mySQLProprietarioDAO =
mySQLDAOFactory.getProprietarioDAO();

            return mySQLProprietarioDAO.inserirProprietario( propTO );
      }
}
```

O DAOFactory é uma classe abstrata que é herdada e implementada pelas diferentes fábricas de DAO, no caso do sistema implementado existia apenas uma chamada MySQLDAOFactory (poderiam haver outras fábricas, por exemplo, XMLDAOFactory, OracleDAOFactory,etc.). Uma Factory ou fábrica aqui citada se refere a outroao padrão Fábrica Abstrata, visto na seção 9.7. O oObjeto de Nnegócios pode obter uma implementação de factoryda fábrica de DAO concreto, por exemplo como o MySQLDAOFactory, e utilizá-la para obter DAOs concretos. O DAO é quem extrai a implementação de acesso de dados para permitir o acesso transparente àa origem de dados. A Fonte fonte de dados representa uma implementação de uma origem de dados como um banco de dados ou repositório XML.

- A interface UsuarioDAO

```
package passeLivre.db.general;
```

```
import passeLivre.model.to.UsuarioTO;


public interface UsuarioDAO {

	public UsuarioTO obterUsuario( String nomeDeAcesso );
	public boolean atualizarUsuario( UsuarioTO usuarioTO );

}
```

- A classe MySQLUsuarioDAO

```
package passeLivre.db.mysql;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import passeLivre.db.general.UsuarioDAO;
import passeLivre.model.to.UsuarioTO;

public class MySQLUsuarioDAO implements UsuarioDAO {

	// referencia para uma conexão com a base de dados
	private Connection con;

	// instruções que serão executadas na base de dados
	private PreparedStatement sqlObterUsuario;
	private PreparedStatement sqlAlterarSenha;

	public MySQLUsuarioDAO( Connection con )
	{
		this.con = con;

		try {

			// instrução localizar cliente
			sqlObterUsuario = con.prepareStatement(
					"SELECT nomeDeAcesso, senha, tipo, cpf, nome,
novo "+
					"FROM usuario "+
					"WHERE nomeDeAcesso = ? ");

			// instrução localizar cliente
			sqlAlterarSenha = con.prepareStatement(
					"UPDATE usuario SET senha = ? , tipo = ?, " +
					"cpf = ?, novo = ?, nome = ? " +
					"WHERE nomeDeAcesso = ? ");

		}
		// pega exceção e apresenta msg de erro quando necessária
		catch ( SQLException e ) {

			e.printStackTrace();
		}
	}

	public UsuarioTO obterUsuario( String nomeDeAcesso ) {

		UsuarioTO usuarioTO = null;
```

```
		try {
			// seta strings chave
			sqlObterUsuario.setString( 1, nomeDeAcesso );

			// executa consulta na base de dados
			ResultSet resultSet = sqlObterUsuario.executeQuery();

			// se não encontrou nenhum registro, retorna imediatamente
			if ( !resultSet.next() ) {

				return null;
			}

			usuarioTO = new UsuarioTO();

			usuarioTO.setNomeDeAcesso( resultSet.getString( "nomeDeAcesso"
) );
			usuarioTO.setSenha( resultSet.getString( "senha" ) );
			usuarioTO.setTipo( resultSet.getInt( "tipo" ) );
			usuarioTO.setCpf( resultSet.getString( "cpf" ) );
			usuarioTO.setNovo( resultSet.getBoolean( "novo" ) );
			usuarioTO.setNome( resultSet.getString( "nome" ) );

		} catch (SQLException e) {

			e.printStackTrace();

		} finally {

			try {

				con.close();

			} catch (SQLException e) {

				e.printStackTrace();
			}
		}
		return usuarioTO;
	}

	public boolean atualizarUsuario( UsuarioTO usuarioTO ) {

		int nlinha = 0;

		try {
			// seta strings chave
			sqlAlterarSenha.setString( 1, usuarioTO.getSenha() );
			sqlAlterarSenha.setInt( 2, usuarioTO.getTipo() );
			sqlAlterarSenha.setString( 3, usuarioTO.getCpf() );
			sqlAlterarSenha.setBoolean( 4, usuarioTO.isNovo() );
			sqlAlterarSenha.setString( 5, usuarioTO.getNome() );
			sqlAlterarSenha.setString( 6, usuarioTO.getNomeDeAcesso() );

			// executa consulta na base de dados
			nlinha = sqlAlterarSenha.executeUpdate();

		} catch (SQLException e) {

			e.printStackTrace();

		} finally {
```

```
                    try {

                          con.close();

                    } catch (SQLException e) {

                          e.printStackTrace();
                    }
              }

              // se não alterou nenhum registro, retorna imediatamente
              if ( nlinha == 0 ) {

                    return false;
              }
              else {

                    return true;
              }
        }
}
```

- A classe DAOFactory

```
package passeLivre.db.general;

import passeLivre.db.mysql.MySQLDAOFactory;

  // Classe abstrata fábrica de DAO
  public abstract class DAOFactory {

  // Lista de tipos de DAO suportado pela fábrica
  public static final int MYSQL = 1;

  // Esse são os métodos que cada DAOFactory deve implementar.
  public abstract ProprietarioDAO getProprietarioDAO();
  public abstract AnalistaDAO getAnalistaDAO();
  public abstract AtendenteDAO getAtendenteDAO();
  public abstract AdministradorDAO getAdministradorDAO();
  public abstract VeiculoDAO getVeiculoDAO();
  public abstract RegistroDeUsoDAO getRegistroDeUsoDAO();
  public abstract GizmoDAO getGizmoDAO();
  public abstract APedagioDAO getAPedagioDAO();
  public abstract PedagioDAO getPedagioDAO();
  public abstract RodoviaDAO getRodoviaDAO();
  public abstract EnderecoDAO getEnderecoDAO();
  public abstract ConcessionariaDAO getConcessionariaDAO();
  public abstract BancoDAO getBancoDAO();
  public abstract UsuarioDAO getUsuarioDAO();

  public static DAOFactory getDAOFactory( int whichFactory ) {

    switch ( whichFactory ) {

      case MYSQL: return new MySQLDAOFactory();

      default : return null;
    }
  }
}
```

- A classe MySQLDAOFactory

```
package passeLivre.db.mysql;

// Implementação da Fabrica DAO concreta da base MYSQL
import java.sql.Connection;

import passeLivre.db.general.APedagioDAO;
import passeLivre.db.general.AdministradorDAO;
import passeLivre.db.general.AnalistaDAO;
import passeLivre.db.general.AtendenteDAO;
import passeLivre.db.general.BancoDAO;
import passeLivre.db.general.ConcessionariaDAO;
import passeLivre.db.general.ConnectionPool;
import passeLivre.db.general.DAOFactory;
import passeLivre.db.general.EnderecoDAO;
import passeLivre.db.general.GizmoDAO;
import passeLivre.db.general.PedagioDAO;
import passeLivre.db.general.ProprietarioDAO;
import passeLivre.db.general.RegistroDeUsoDAO;
import passeLivre.db.general.RodoviaDAO;
import passeLivre.db.general.UsuarioDAO;
import passeLivre.db.general.VeiculoDAO;

public class MySQLDAOFactory extends DAOFactory {

	// método para criar uma conexão com a base de dados MySQL
	public static Connection createConnection() {

		Connection conexao = null;

		// pega uma conexao da Fabrica do pool
		conexao = ConnectionPool.getInstance().getConnection();

		return conexao;
	}

	public ProprietarioDAO getProprietarioDAO() {
		 // MySQLProprietarioDAO implementa ProprietarioDAO
		return new MySQLProprietarioDAO( createConnection() );
	}

	public AnalistaDAO getAnalistaDAO() {
		// MySQLAnalistaDAO implementa AnalistaDAO
		return new MySQLAnalistaDAO( createConnection() );
	}

	public AtendenteDAO getAtendenteDAO() {
		return new MySQLAtendenteDAO( createConnection() );
	}

	public AdministradorDAO getAdministradorDAO() {
		return new MySQLAdministradorDAO( createConnection() );
	}

	public VeiculoDAO getVeiculoDAO() {
		return new MySQLVeiculoDAO( createConnection() );
	}

	public RegistroDeUsoDAO getRegistroDeUsoDAO() {
		return new MySQLRegistroDeUsoDAO( createConnection() );
	}
```

```
	public GizmoDAO getGizmoDAO() {
		return new MySQLGizmoDAO( createConnection() );
	}

	public APedagioDAO getAPedagioDAO() {
		return new MySQLAPedagioDAO( createConnection() );
	}

	public PedagioDAO getPedagioDAO() {
		return new MySQLPedagioDAO( createConnection() );
	}

	public RodoviaDAO getRodoviaDAO() {
		return new MySQLRodoviaDAO( createConnection() );
	}

	public EnderecoDAO getEnderecoDAO() {
		return new MySQLEnderecoDAO( createConnection() );
	}

	public ConcessionariaDAO getConcessionariaDAO() {
		return new MySQLConcessionariaDAO( createConnection() );
	}

	public BancoDAO getBancoDAO() {
		return new MySQLBancoDAO( createConnection() );
	}

	public UsuarioDAO getUsuarioDAO() {
		return new MySQLUsuarioDAO( createConnection() );
	}
}
```

Para facilitar a conexão com a base de dados, um pool de conexões é criado com a classe ConnectionPool.

- A classe Connection Pool

```
package passeLivre.db.general;

import java.sql.Connection;
import java.sql.SQLException;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

// final -> evita que seja feita uma herança
public final class ConnectionPool {

	private static final String DATA_SOURCE_MYSQL =
"java:comp/env/jdbc/passeLivre";

	private DataSource dataSource;		// pool de conexão com a base de
dados
	private static ConnectionPool mySelf;  // referência para uma única
instância dessa classe

	  // construtor privado
	private ConnectionPool( DataSource dataSource ) {
```

```
            this.dataSource = dataSource;

      }

      public static synchronized ConnectionPool getInstance() {

            try {
               // verifica se ainda não foi criada uma única instância
                  if( mySelf == null ) {
                        // pega o contexto da aplicação
                        Context contexto = new InitialContext();
                        // pega o pool de conexões com a base
                        DataSource dataSource = ( DataSource )contexto.lookup(
DATA_SOURCE_MYSQL );
                  // cria a única instância dessa classe
                        mySelf = new ConnectionPool( dataSource );
                  }

            } catch( NamingException e ) {
                  System.err.println( e.getMessage() );
            }

            return mySelf;
      }

      public Connection getConnection() {
            Connection con = null;
            try {
                  con = dataSource.getConnection();
            } catch ( SQLException e ) {
                  System.err.println( e.getMessage() );
            }

            return con;
      }
}
```

As vantagens que se podem conseguir utilizando esses padrões são várias. As mais relevantes são: transparência, redução na complexidade dos códigos nos objetos de negócio e centralização de acesso aos dados em uma camada separada o que torna a aplicação mais fácil de manter e gerenciar. Além disso, uma pessoa interessada em estudar e estender o código exemplo poderia facilmente acrescentar uma nova fonte de dados sem se preocupar em fazer alterações em todas as camadas do sistema.

Capítulo 13 – Ciclo de Desenvolvimento 2 do Sistema Passe Livre

13.1 – Casos de Uso no formato completo abstrato

Terminado o ciclo 1, partimos então para a análise do ciclo 2, em que foram alocados os seguintes casos de uso: 1 – Receber Aviso de Débito, 2 – Desativar Gizmo, 3 – Reativar Gizmo, 4 – Efetuar Pagamento Pendente, 5 – Consultar Proprietários Inadimplentes, 6 – Alterar dados do proprietário e 7 – Alterar senha proprietário.

Embora esses casos de uso tenham ficado para o segundo ciclo de desenvolvimento, eles são igualmente importantes para o correto funcionamento do sistema, embora sejam menos complexos de implementar. A Figura 13.1 ilustra o caso de uso Receber Aviso de Débito. Outros exemplos de casos de uso no formato completo abstrato estão nas Figuras 13.2 e 13.3.

Caso de Uso: Receber Aviso de Débito

Ator Principal: Analista Financeiro
Interessados e Interesses:
Proprietário: deseja informar ao sistema que sua conta foi debitada do valor dos pedágios do período e que portanto ele está em dia com suas obrigações.
Empresa: deseja conferir quais proprietários estão em dia e quais não tinham saldo suficiente e que portanto devem regularizar sua situação por meio de pagamento direto em um posto da concessionária.
Pré-Condições: O Analista Financeiro é identificado e autenticado.
Garantia de Sucesso (Pós-Condições): Os clientes adimplentes tem sua situação renovada e os clientes inadimplentes recebem uma comunicação para que paguem seus débitos.
Cenário de Sucesso Principal:

8. O Analista Financeiro informa o dia do vencimento ao qual se refere o aviso de débito.
9. O Sistema exibe os bancos que possuem débitos a efetuar em tal vencimento.
10. Para cada banco, o Analista Financeiro informa a identificação dos proprietários e as respectivas situações do débito (sucesso ou fracasso).
11. Se a situação é de sucesso, o Sistema atualiza o registro do proprietário com a data do débito. Caso contrário, o Sistema emite um aviso ao proprietário inadimplente, informando os endereços autorizados a receber o pagamento.

Fluxos Alternativos:
1a. O dia informado é inválido ou não corresponde a um dia de fechamento. Informar novamente.
1b. Já foram registrados todos os avisos de débito para o dia informado. Cancelar.

Figura 13.1 – Caso de Uso "Receber Aviso de Débito" no formato Completo Abstrato

Caso de Uso: Desativar Gizmo

Ator Principal: Proprietário
Interessados e Interesses:
- Proprietário: deseja desativar um ou mais gizmos que estão sendo utilizados pelos seus veículos. Deseja que isso seja feito por uma interface simples e intuitiva.
- Empresa: deseja que o Sistema seja informado quando um Proprietário desativar um ou mais gizmos. Deseja que isso seja feito de maneira rápida e que as informações trocadas com o Sistema seja feita de maneira consistente.

Pré-Condições: O Proprietário é identificado e autenticado no Sistema.
Garantia de Sucesso (Pós-Condições): Os gizmos dos veículos selecionados pelo Proprietário adimplente são desativados no Sistema.
Cenário de Sucesso Principal:
15. O Proprietário consulta os veículos cadastrados.
16. O Sistema apresenta uma lista dos veículos do Proprietário.
17. O Proprietário seleciona os veículos que terão os gizmos desativados.
18. O Proprietário seleciona opção desativar gizmos.
19. O Sistema apresenta uma mensagem ao Proprietário informando que os gizmos desativados só poderão ser reativados mediante o pagamento de uma taxa, na presença de um Atendente em uma loja autorizada.
20. O Proprietário confirma a operação.
21. O Sistema desativa os gizmos dos veículos selecionados.

Fluxos Alternativos:
2a. O Sistema informa ao Proprietário que não existe nenhum veículo cadastrado.
1. O Proprietário desiste da operação desativar gizmo.
3a. O Proprietário verifica que não existe nenhum veículo com gizmo ativado.
1. O Proprietário desiste da operação desativar gizmo.
4a. O Sistema informa ao Proprietário que ele possui contas pendentes e que deve regularizar suas contas para desativar o uso dos gizmos.
1. O Proprietário desiste da operação desativar gizmo.
6a. O Proprietário cancela operação.

Requisitos especiais:
- A comunicação entre o Sistema na Web e o servidor deve usar criptografia e outras técnicas de segurança para oferecer maior segurança aos usuários do Sistema.
- O Sistema deve estar sempre disponível e nunca deixar de operar. A tolerância máxima é de 15min por ano com o Sistema fora de operação, contando inclusive manutenções programadas.

Figura 13.2 – Caso de Uso "Desativar Gizmo" no formato Completo Abstrato

Caso de Uso: Consultar Proprietários Inadimplentes

Ator Principal: Analista Financeiro
Interessados e Interesses:
- Analista Financeiro: deseja consultar os dados dos Proprietários inadimplentes.
- Empresa: deseja que os dados dos Proprietários inadimplentes possam ser consultados com facilidade.

Pré-Condições: O Analista Financeiro é identificado e autenticado.

Garantia de Sucesso (Pós-Condições): Os dados referentes aos Proprietários inadimplentes são apresentados ao Analista Financeiro.
Cenário de Sucesso Principal:

1. O Analista Financeiro seleciona a opção consultar Proprietários inadimplentes.
2. O Sistema apresenta ao Analista Financeiro uma lista com os nomes de acesso, nomes, sobrenomes e telefones de todos os Proprietários que se encontram em situação irregular. Alem disso apresenta o número total de Proprietários inadimplentes e o valor total da dívida.
3. O Analista Financeiro seleciona um Proprietário para que possa visualizar os valores devidos por mês e ano, assim como o valor total da dívida.

Fluxos Alternativos:

2a. Nenhum Proprietário está em situação irregular.

1. O Sistema informa ao Analista Financeiro que todos os Proprietários estão em situação regular.

Figura 13.3 – Caso de Uso "Consultar Proprietários Inadimplentes" no formato Completo Abstrato

13.2 – Modelo Conceitual

Optamos, no ciclo 1, por já desenvolver o Modelo Conceitual de todo o sistema. No entanto, devemos re-visitar os casos de uso do ciclo 2 para verificar se surge algum novo conceito e respectivas associações. Como os casos de uso do ciclo 2 referem-se a conceitos já modelados no ciclo 1, consideramos então que o Modelo Conceitual da Figura 11.7 continua sendo válido para o ciclo 2.

13.3 – Diagramas de Seqüência do Sistema e Contratos das Operações

Os DSSs para os casos de uso do ciclo 2 devem ser construídos da mesma forma que os do ciclo 1. Algumas operações podem já ter sido definidas anteriormente e poderão ser reusadas. Por exemplo, ao definir o DSS para Reativar Gizmo pode-se reusar operações do DSS para Desativar Gizmo, tal como a operação para exibir todos os veículos de um determinado proprietário.

Devem ser feitos os contratos para as operações identificadas nos DSSs, a menos para operações comuns já definidas anteriormente. Deixaremos como exercício para o leitor o desenvolvimento dos DSSs e respectivos contratos.

13.4 – Discussão sobre o PU

Ao invés de prosseguirmos com o detalhamento de outros casos de uso e seus DSSs/ contratos, preferimos aprofundar um pouco mais sobre o Processo Unificado e suas fases. Pudemos perceber, durante o decorrer do desenvolvimento do Sistema Passe Livre, que existe uma transição bastante suave entre as diversas fases do PU. Começando pela

concepção, que resulta em artefatos (como por exemplo o diagrama de casos de uso) que delimitam o sistema e já fornecem uma boa idéia de sua complexidade e escopo.

Na elaboração, após escolher os casos de uso a serem atacados no ciclo atual, esses casos de uso são mais detalhados e produz-se o modelo conceitual, os DSSs e contratos, que fornecem os detalhes necessários para prosseguir com o projeto. Na construção, são projetadas as interações entre os objetos para realizar cada operação dos DSSs, com base nos artefatos das fases anteriores. O diagrama de classes é desenvolvimento e o sistema é implementado usando uma linguagem de programação orientada a objetos. Esta fase ocorre de maneira iterativa e incremental, de forma que um incremento do sistema é produzido, testado e pode ser submetido à fase posterior, a transição. Finalmente, na transição o incremento é implantado no cliente, submetido a testes e colocado em operação.

O ciclo de repete, mas em um segundo ciclo a fase de concepção é abreviada, tratando apenas de ajustar escopo, custos e cronograma, além de tratar os riscos que possam ter se manifestado com a conclusão do ciclo anterior. Se necessário, a arquitetura do sistema pode ser revista ou modificada, embora isso deva ser evitado, pois pode provocar um impacto muito grande nos custos e cronogramas. Lembre-se que o PU é centrado na arquitetura, que deve ser o alicerce para a construção do sistema, de forma que mudanças na arquitetura podem por em risco todo o projeto.

Já as demais fases (elaboração, construção e transição) no segundo ciclo em diante são similares à do primeiro ciclo, ou seja, escolhem-se outros casos de uso para serem atacados, e o processo é o mesmo, a menos dos refinamentos que podem ser feitos nos artefatos produzidos em ciclos anteriores.

Capítulo 14 – Outras questões de projeto

No capítulo 12 apresentamos alguns artefatos referentes à fase de construção do PU para o ciclo 1 do Sistema Passe Livre. Dentre os artefatos apresentados estavam os diagramas de colaboração para algumas das operações pertencentes aos DSSs e o diagrama de classes do Sistema. Apresentamos também o código fonte de algumas das classes, com duas versões: uma simplificada, sem utilizar a arquitetura em três camadas, e outra mais robusta.

Neste capítulo, ao invés de mostrar outros artefatos similares aos já apresentados, vamos aprofundar um conceito muito importante durante a fase de projeto: o problema da persistência de dados.

Não se trata de deixarmos para resolver esse problema somente no ciclo 2 de desenvolvimento de um projeto, mas para a finalidade deste livro achamos que seria mais difícil para o leitor entender esse conceito antes de tentar implementar um sistema OO de verdade. Acreditamos que o problema é mais valorizado após pelo menos uma tentativa de implementar o sistema, pois o aluno enfrentará de perto as questões que serão abordadas a seguir.

14.1 – O problema da persistência

Retomando o exemplo do nosso sistema de Biblioteca, suponha que todas as instâncias de Livro residam em algum mecanismo de armazenamento persistente e precisem ser trazidas para a memória durante o uso do sistema de Biblioteca. Objetos Persistentes são aqueles que necessitam de armazenamento, tais como um banco de dados relacional ou orientado a objetos, um arquivo simples, XML, ou qualquer outros tipo de armazenamento.

Se escolhermos um banco de dados orientado a objetos, não precisaremos de serviços de persistência adicionais, pois basta informar ao sistema que o objeto é persistente e sua gravação na base de dados será automaticamente realizada pelo sistema.

Por outro lado, se escolhermos um banco de dados relacional, teremos representações diferentes devido ao desencontro entre as representações de dados orientadas a registros e as orientadas a objetos. Com isso, serão necessários serviços para resolver esse problema. O mesmo ocorre se escolhermos outros meios de armazenamento, tais como arquivos comuns, banco de dados hierárquicos, etc.

Como esse problema tem sido enfrentado por muitos projetistas desde o surgimento da OO, várias soluções têm sido propostas, entre elas a criação de um framework para persistência, que é um conjunto de classes reutilizáveis que fornece serviços para objetos persistentes. As tarefas típicas de um framework de persistência (FP) são: traduzir objetos para registros, salvar objetos no banco de dados, traduzir registros para objetos. As funções principais do FP são: armazenar e recuperar os objetos em um mecanismo de

armazenamento persistente e efetuar confirmação (commit) e retrocesso (rollback) de transações do banco de dados. As qualidades esperadas de um FP são: ser extensível, de forma que novas funcionalidades possam ser acrescidas com um mínimo de mudanças no código; ser fácil de usar, já que praticamente todas as aplicações OO precisam fazer persistência de maneira fácil e segura; e ser transparente, para que o código da aplicação principal tenha o mínimo possível de interação com o código do FP em si.

Uma possível solução para implementar um FP seria criar uma classe *ObjetoPersistente*, com atributos e métodos para apoiar a persistência. Todos os objetos persistentes deveriam herdar dessa classe, direta ou indiretamente. O problema é o forte acoplamento causado por essa solução, já que nem sempre as classes do sistema poderiam herdar da classe *ObjetoPersistente*, ou essa herança poderia não parecer natural em muitos casos. Entretanto, mesmo com esses problemas, essa solução tem sido empregada em diversos FPs existentes.

14.2 – Problemas para implementar um FP

Nesta seção listamos alguns problemas-chave que devem ser resolvidos quando pensamos em implementar um FP. Esses problemas podem ser resolvidos de diversas maneiras e, na seção seguinte, apresentamos padrões específicos para resolver cada um deles.

Um FP deve oferecer alguma forma de mapeamento entre cada classe e seu armazenamento persistente. Por exemplo, cada classe poderia ter uma tabela correspondente, sendo que os atributos da classe corresponderiam aos campos do registro da tabela. O FP deve tratar o problema da identidade dos objetos, já que deve haver alguma forma de recuperar o objeto a partir da base dados. Isso pode ser resolvido atribuindo um identificador único a cada objeto, para evitar duplicatas indesejadas do mesmo objeto.

Um outro conceito importante ao pensar em um FP é o de materialização e desmaterialização dos objetos. A materialização é a transformação de uma representação de dados não orientada a objetos em um objeto. No caso mais simples, em que uma classe é mapeada para uma tabela de uma base de dados relacional, a materialização é a recuperação de um registro de uma tabela e atribuição desse registro a um objeto da classe correspondente. Desmaterialização é o processo inverso, ou seja, o objeto que está materializado é persistido na base de dados.

Uma decisão importante referente à materialização é sobre quando fazê-la, isto é, vamos recuperar o objeto somente quando ele for referenciado ou vamos recuperar todos os objetos no início da execução do sistema, de forma que já estejam disponíveis quando precisarmos dele. Chamamos de materialização sob demanda (*lazy materialization*, em inglês) quando nem todos os objetos são materializados de uma vez, mas somente quando necessário ou requisitado. Isso pode ser implementado usando Referências Inteligentes (*smart references*) – ou padrão Proxy Virtual.

Outra decisão diz respeito a objetos complexos, que são aqueles não compostos apenas de atributos simples, mas de atributos que fazem referências a outros objetos ou coleções de objetos, que por sua vez podem também fazer referência a outros objetos. Como representar esses objetos e como materializá-los? A materialização sob demanda pode ajudar, mas pode prejudicar o desempenho do sistema. Voltaremos a esse assunto na próxima seção.

Uma forma elegante de otimizar a persistência é manter o estado de transação de um objeto: objetos modificados ficam "sujos" (dirty) para determinar se precisam ser salvos de volta no seu armazenamento persistente. Dessa forma, evitamos o acesso desnecessário à base de dados. Podemos também pensar em manter o controle sobre operações em transações, por exemplo, efetivar a transação (commit) e desfazer a transação (rollback), para minimizar perda de dados.

14.3 – Uso de padrões para Persistência

Na seção anterior vimos que, ao implementar um FP, deparamo-nos com várias questões de projeto. Os padrões das tabelas 14.1 a 14.6 nos ajudam a resolver esses problemas.

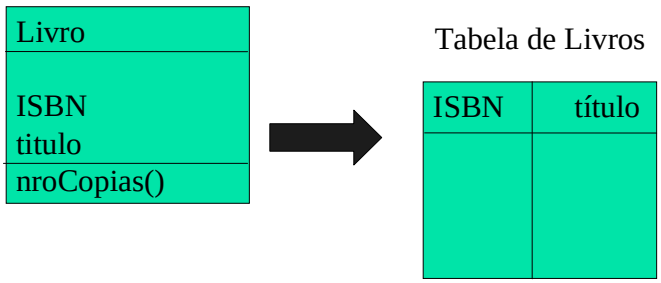
Tabela 14.1 – Padrão "Representar Objetos como Tabelas"

Nome: Representar Objetos como Tabelas

Problema: Como mapear um objeto para um arquivo ou um esquema de banco de dados relacional?

Solução: Defina uma tabela para cada classe de objetos persistentes. Os atributos de objetos que contém tipos de dados primitivos são mapeados para colunas.

Exemplo:



Conseqüências: Para objetos apenas com atributos simples, o mapeamento é direto. Porém, objetos complexos possuem atributos que referenciam outros objetos (às vezes complexos por sua vez).

Tabela 14.2 – Padrão "Identificador do Objeto"

Nome: Identificador do Objeto

Problema: Como relacionar objetos com registros e garantir que a materialização de um objeto não resulte em objetos duplicados?

Solução: Atribuir um identificador de objeto (OID) para cada registro de objeto.

Exemplo: usar o Identificador Universal Único de 16 bytes, o qual garante que cada valor seja único para qualquer data e hora. Existe uma API do Windows que fornece uma função para geração automática desse identificador.

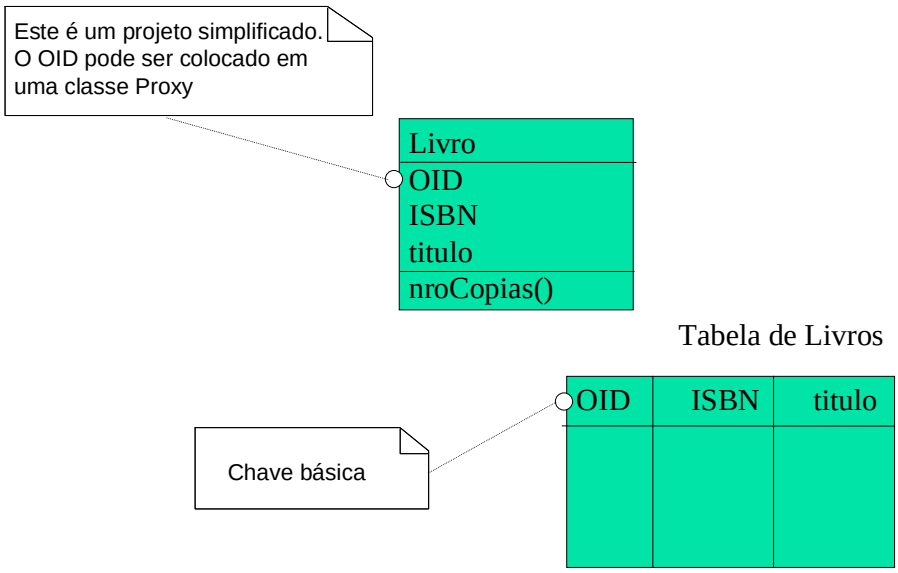


Tabela 14.3 – Padrão "Intermediário ("Database Broker")"

Nome: Intermediário ("Database Broker")

Problema: como evitar o alto acoplamento de uma classe de objetos persistentes ao conhecimento do mecanismo de persistência? Como evitar a alta coesão de acrescentar a uma classe responsabilidades não relacionadas à sua função principal?

Solução: Criar uma classe Intermediária "Broker", responsável pela materialização, desmaterialização e memorização prévia (*cache*) dos objetos.

Exemplo:

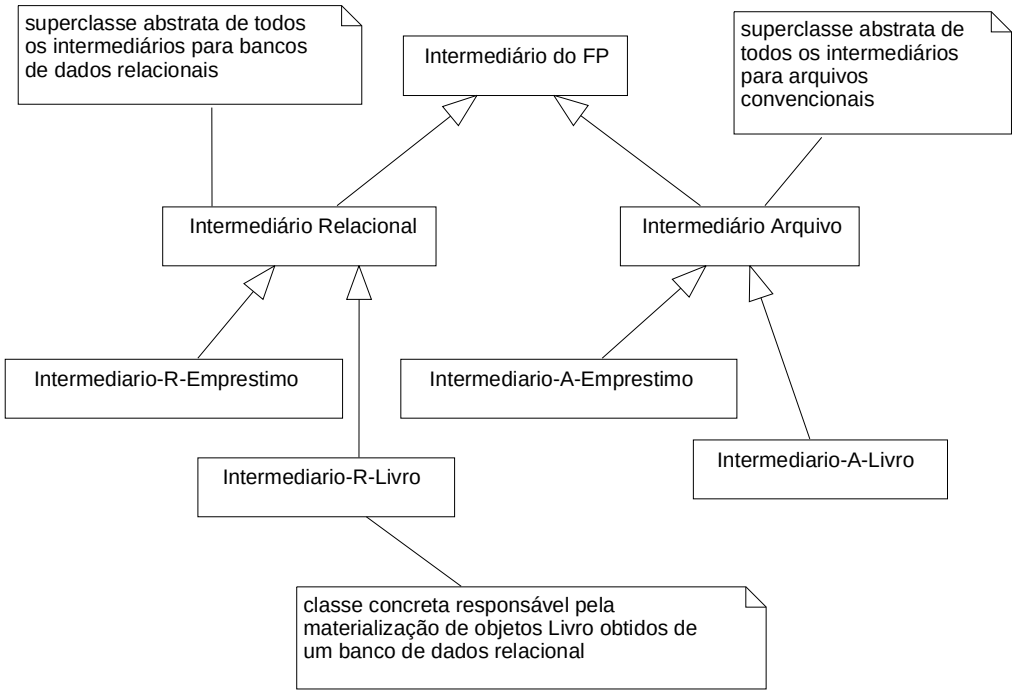


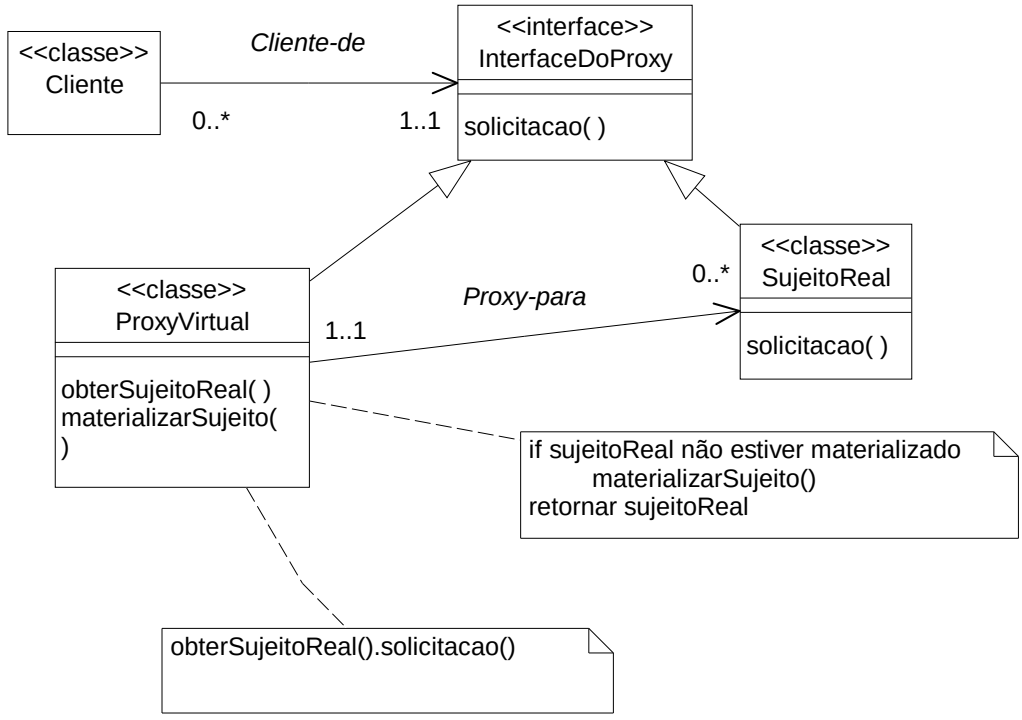
Tabela 14.4 – Padrão Procurador Virtual "Virtual Proxy"

Nome: Procurador Virtual (*Virtual Proxy*)

Problema: como evitar que objetos sejam materializados desnecessariamente, consumindo tempo e espaço em memória?

Solução: Realizar a materialização sob demanda, por meio do padrão GoF [Gamma 1995] Procurador Virtual. Um Procurador virtual é uma referência inteligente para o objeto real. Ele materializa o objeto real quando referenciado pela primeira vez, portanto implementa a materialização sob demanda. É considerado uma referência inteligente porque é tratado pelo cliente como se fosse o próprio objeto.

Exemplo:



Outro problema que precisa ser solucionado, havendo inclusive padrões para isso mas que não serão mostrados aqui, é o da representação de objetos complexos, que possuem conexões com outros objetos, e não simplesmente atributos primitivos simples. Considerando bancos de dados relacionais, se houver uma associação do tipo um-para-um, basta colocar uma chave estrangeira que seja um OID em uma ou em ambas as tabelas ou criar uma tabela associativa que registra os OIDs de cada objeto no relacionamento.

Já nas situações em que houver uma associação um-para-muitos ou muitos-para-muitos, a solução é criar uma tabela associativa que registra os OIDs de cada objeto do relacionamento.
Retornemos agora ao assunto da materialização, seja total ou sob demanda. Considere o exemplo da Biblioteca, em que temos duas associações: Emprestimo – LinhaDoEmprestimo e LinhaDoEmprestimo – CopiaDoLivro. O que significa materializar Emprestimo? Emprestimo é materializado? Ou Emprestimo, suas linhas de empréstimo e suas respectivas cópias do livro são materializadas?

No caso de hierarquias com muitos níveis, é possível que dezenas ou centenas de objetos relacionados precisem também ser materializados. Isso geralmente causa lentidão e ineficiência (espaço gasto) no sistema. Uma boa solução é adiar a materialização de objetos de acordo com as formas de acesso e requisitos de desempenho. O caso extremo é aquele em que 100% de materialização é feita sob demanda. Nesse caso, garante-se que o acesso à base de dados é mínimo, mas acaba sendo muito ineficiente se houverem poucos objetos sendo materializados. Uma solução considerada como um meio termo é a materialização até 1 ou 2 níveis da hierarquia. Se for usado um Intermediário diferente para cada objeto persistente, é possível decidir esse nível individualmente, de acordo com cada aplicação específica.

As Figuras 14.1 a 14.8 ilustram o processo de materialização sob demanda. Na Figura 14.1 relembramos a visibilidade entre as classes, determinada durante o projeto dos diagramas de colaboração, por exemplo o da Figura 14.2.

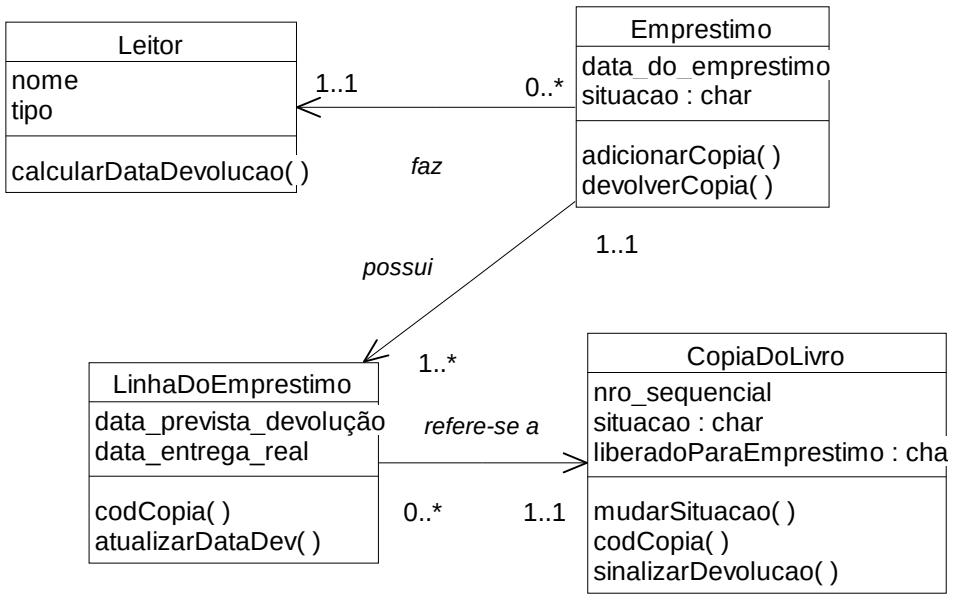


Figura 14.1 – Associações no modelo Conceitual

Veja na Figura 14.2 que, quando a mensagem 2 é invocada, não é necessário materializar o objeto CopiaDoLivro, pois precisamos apenas de seu código. Já quando a mensagem 4 é invocada, precisamos modificar a situação do livro, por isso precisamos ter acesso ao

objeto real. Usando um Proxy para CopiaDoLivro, como na Figura 14.3, a linha do empréstimo referenciará o proxy e não o objeto real. Assim, o objeto CopiaDoLivro só será materializado, por exemplo, quando o método sinalizaDevolucao() for invocado, pois ele precisará chamar o método mudarSituacao(), que precisará do SujeitoReal para poder modificar a situação do livro.

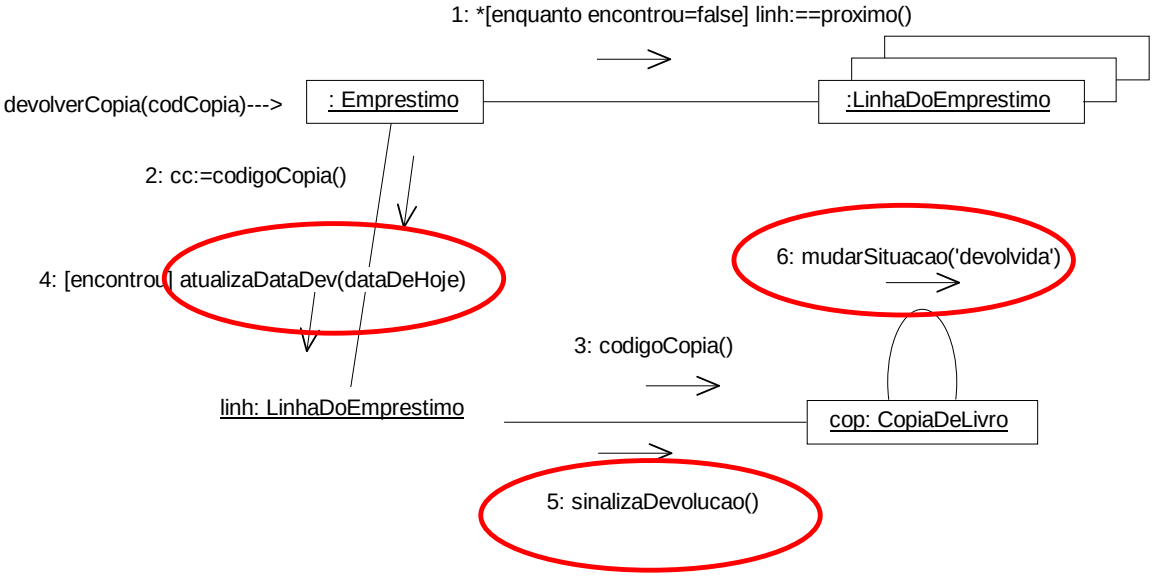
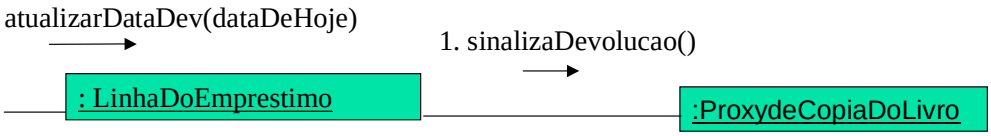


Figura 14.2 – Diagrama de colaboração da operação



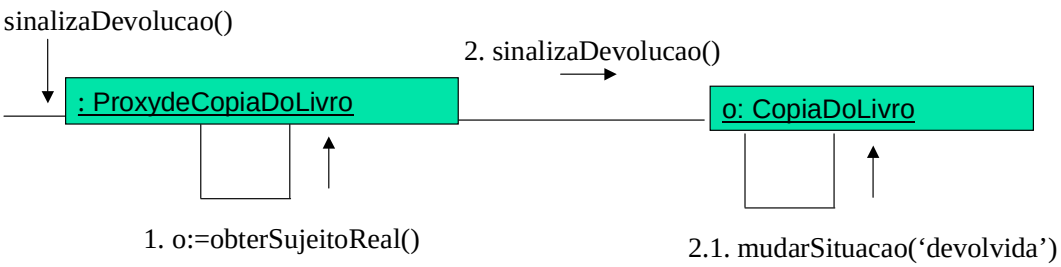


Figura 14.3 – Materialização de Empréstimo (passo 3)

Na Figura 14.4 vemos como a materialização é realizada usando o padrão Intermediário (visto acima). No momento que o proxy precisa materializar o objeto, ele invoca o

intermediário, que no caso é específico para materializar objetos de uma base de dados relacional.

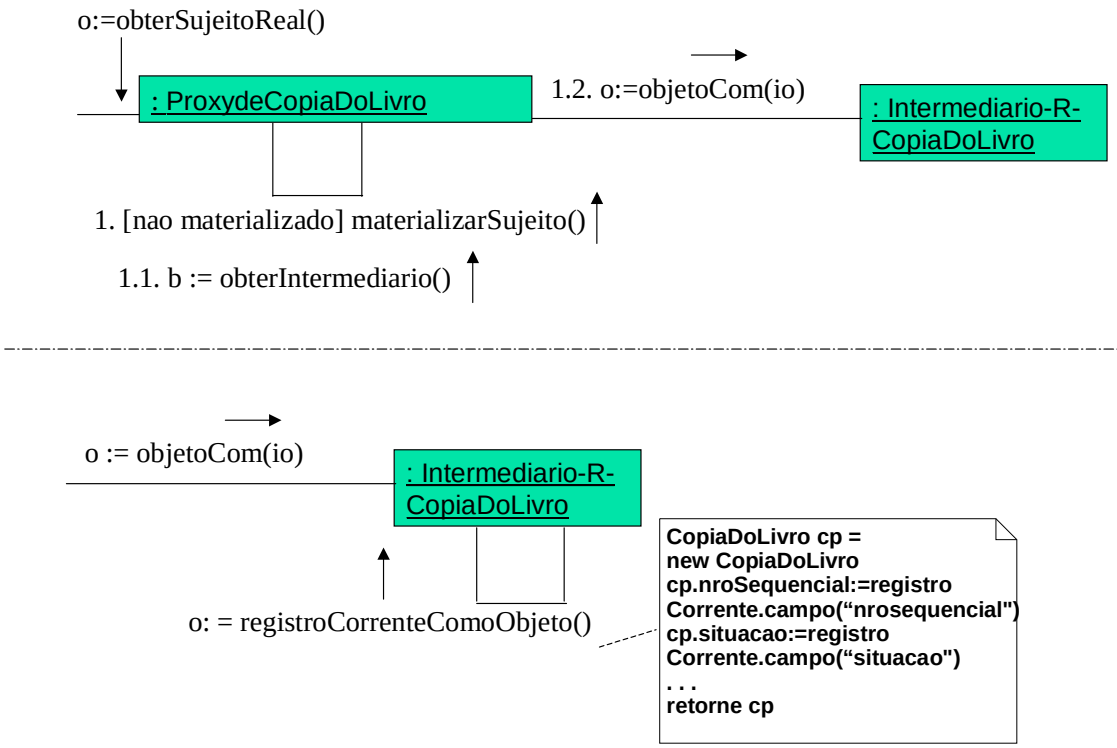


Figura 14.4 – Materialização de Empréstimo (passo 3)

As figuras 14.5 a 14.7 ilustram tabelas relacionais referentes ao diagrama de classes do Sistema de Biblioteca. Dado um identificador de um empréstimo (OID), as respectivas linhas de empréstimo podem ser recuperadas, cada uma das quais possui um identificador da cópia do livro emprestado (Figura 14.5). Então, com base no identificador da linha do empréstimo, pode-se descobrir qual é a cópia do livro (Figura 14.6). Finalmente, tendo o identificador da cópia do livro, pode-se ter acesso à cópia do livro para realizar a mudança do atributo *situação*.

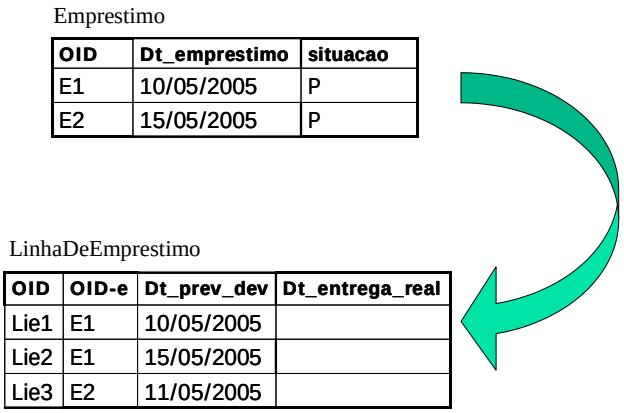


Figura 14.5 – Materialização de Empréstimo (passo 1)

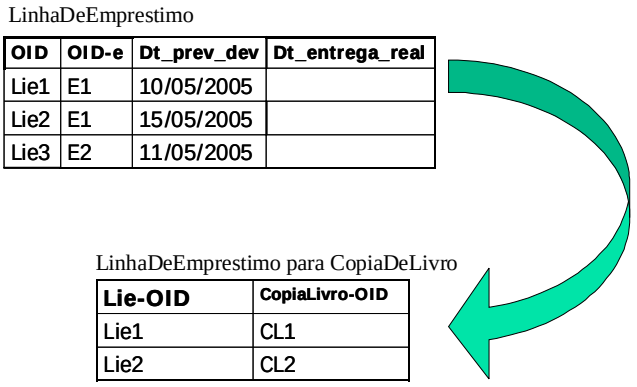


Figura 14.6 – Materialização de Empréstimo (passo 2)

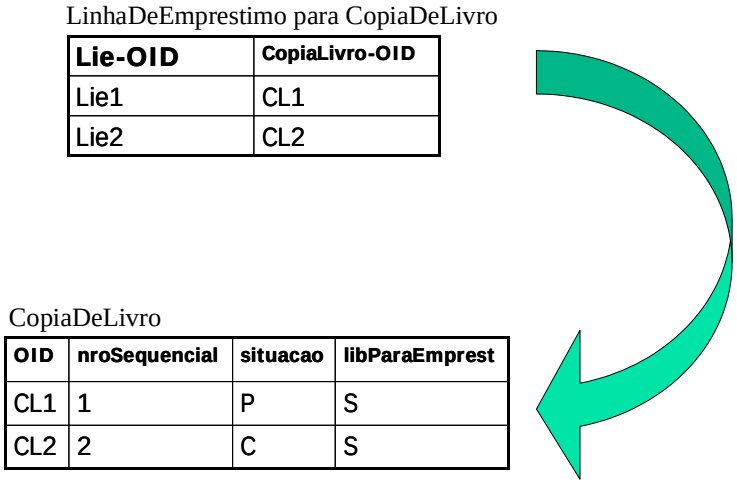


Figura 14.7 – Materialização de Empréstimo (passo 3)

Referências

[Alexander 77] Christopher Alexander et. al., A Pattern Language, Oxford University Press, New York, 1977.

[Alexander 79] Christopher Alexander, The Timeless Way of Building, Oxford University Press, New York, 1979.

[Appleton 97] Appleton, Brad. Patterns and Software: Essential Concepts and Terminology, disponível na WWW na URL: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

[Bauer, 2004] Bauer, Christian; King, Gavin. Hibernate in Action, Manning Publications.

[Beck 87] Beck, Kent; Cunningham, Ward. Using Pattern Languages for Object-Oriented Programs, Technical Report nº CR-87-43, 1987, disponível na WWW na URL: http://c2.com/doc/oopsla87.html

[Booch, 1995] Object Solutions : Managing the Object-Oriented Project (Addison-Wesley Object Technology Series by Grady Booch , Pearson Education; 1st edition (October 12, 1995)

[Buschmann 96] Buschmann, F. et at. A System of Patterns, Wiley, 1996.

[Coad 92] Coad, Peter. Object-Oriented Patterns. Communications of the ACM, V. 35, nº9, p. 152-159, setembro 1992.

[Coad 95] Coad, P.; North, D.; Mayfield, M. Object Models: Strategies, Patterns and Applications, Yourdon Press, 1995.

[Coleman et al, 1994] Coleman, Derek et al. Object Oriented Development - the Fusion Method, Prentice Hall, 1994.

[Coplien 92] Coplien, J.O. Advanced C++ Programming Styles and Idioms. Reading-MA, Addison-Wesley, 1992.

[Coplien 95] Coplien, J.; Schmidt, D. (eds.) Pattern Languages of Program Design, Reading-MA, Addison-Wesley, 1995.

[Deitel, 2002] Deitel, Harvey M; Deitel Paul J. JAVA – como programar, Editora Bookman, 4a Edição.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Reading-MA, Addison-Wesley, 1995.

[Goodwill, 2002] Goodwill,J. Martering Jakarta Struts, Wiley Publishing, Inc.

[Krutchen, 2000] Krutchen, Philippe. The Rational Unified Process, An Introduction, Second Edition, Addison Wesley, 2000.

[Larman, 2004] Larman, Craig. Utilizando UML e Padrões, 2a edição, Bookman, 2004.

[Rational, 2000] RATIONAL, C. Unified Modeling Language. Disponível na URL: http://www.rational.com/uml/references, 2000.

[Rumbaugh, 1990] Object-Oriented Modeling and Design by James R Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, William Premerlani, Prentice Hall; 1st edition (October 1, 1990)

[Waslawick, 2004] Waslawick, Raul. Análise e Projeto de sistemas de Informação Orientados a Objetos, Campus, 2004.

[Yourdon, 1990] Youdon, Edward. Análise estruturada moderna. Ed. Campus, 1990.