UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ANÁLISE E PROJETO OO
&
UML 2.0

Cesar Augusto Tacla
Departamento Acadêmico de Informática
http://www.dainf.cefetpr.br/~tacla

SUMÁRIO

I INTRODUÇÃO

Neste capítulo são apresentados os conceitos fundamentais do curso: modelo, UML, análise e projeto orientado a objetos, objeto e classes de objetos.

1 MODELO

Antes de entrar nos detalhes de UML, é preciso ater-se ao conceito de modelo.

> Um **modelo** é uma simplificação da realidade que descreve um sistema de um ponto de vista particular.

Por exemplo, um projeto arquitetônico é feito segundo diversas perspectivas: do arquiteto, projeto arquitetônico em si, do engenheiro eletricista, projeto elétrico, do engenheiro civil, projeto hidráulico e estrutural. Construímos modelos de sistemas complexos para melhor compreendê-los.

Abstrair e refinar incrementalmente são palavras-chaves. Em certos momentos, o projetista deve focalizar na interação entre componentes do sistema sem se preocupar com seus detalhes internos de funcionamento, então ele abstrai estes detalhes. Em outros momentos, é preciso detalhar o comportamento dos componentes. Enfim projetar um sistema significa fazer modelos sob diferentes perspectivas e graus de abstração, representando-os por meio de uma notação precisa, refinando-os sucessivamente até transformá-los em algo próximo da implementação lembrando sempre de verificar se os requisitos são satisfeitos.

A modelagem visual (com auxílio de diagramas) ajuda a manter a consistência entre os artefatos (produtos) ligados ao desenvolvimento de um sistema: requisitos, projeto e implementação. Resumidamente, a modelagem visual pode melhorar a capacidade de uma equipe a gerenciar a complexidade de software.

2 UML

UML significa *Unified Modeling Language* ou Linguagem de Modelagem Unificada de projetos orientados a objetos. Como o próprio nome diz, **UML é uma linguagem** e não um método!

> A UML é uma linguagem padrão de notação de projetos.

Por notação entende-se especificar, visualizar e documentar os elementos de um sistema OO. A UML é importante, pois:

- ◇ serve como linguagem para expressar decisões de projeto que não são óbvias ou que não podem ser deduzidas do código;
- ◇ provê uma semântica que permite capturar as decisões estratégicas e táticas;
- ◇ provê uma forma concreta o suficiente para a compreensão das pessoas e para ser manipulada pelas máquinas.
- ◇ É independente das linguagens de programação e dos métodos de desenvolvimento.

2.1 Breve histórico

Nos anos 90, conhecida como a época da "guerra dos métodos", vários métodos coexistiam com notações muitas vezes conflitantes entre si. Dentre estes, os mais conhecidos eram:

- ◇ OMT (*Object Modelling Technique*) de Rumbaugh;
- ◇ Método de Booch;
- ◇ OOSE (Object Oriented Software Engineering) de Jacobson;

Inicialmente, Rumbaugh (OMT) e Booch fundiram seus métodos (e notações) resultando no Método Unificado em 1995 quando trabalhavam juntos na Rational Software (atualmente uma divisão da IBM). Jacobson juntou-se a eles mais tarde e seu método OOSE foi incorporado à nova metodologia (RUP).

Salienta-se que além do método, eles unificaram a **notação de projeto** e a chamaram UML. Então, UML representa a unificação das notações de Booch, OMT e Jacobson. Também agrega as idéias de inúmeros autores, tais como Harel e seu diagramas de estados, Shlaer-Mellor e o ciclo de vida dos objetos. Em suma, UML é uma tentativa de padronizar os artefatos de análise e projeto: modelos semânticos, sintaxe de notação e diagramas.

Na década de 90, surge uma organização importante no mundo dos objetos a OMG (*Object Management Group)*, uma entidade sem fins lucrativos onde participam empresas e acadêmicos para definirem padrões de tecnologias OO.

- Outubro de **1995**: primeira versão rascunho, versão 0.8 draft.
- Julho de **1996**: revisão devido ao ingresso de Jacobson, versão 0.9 draft.
- Parceiros UML (HP, IBM, Microsoft, Oracle e Rational Software) desenvolveram a versão 1.1 e a propuseram OMG
- A OMG aceita a proposta em novembro de **1997** e assume a responsabilidade de realizar manutenção é revisão da UML
- Em março de **2003**, a OMG lançou a versão 1.5
- Em outubro de **2004**, a OMG lançou versão 2.0 adotada[1]

3 ANÁLISE E PROJETO ORIENTADOS A OBJETOS

Há vários métodos de desenvolvimento de software. Na década de 80 houve preponderância dos métodos estruturados. Atualmente o paradigma OO é mais forte e, por isso, é importante diferenciar análise e projeto estruturado e orientado a objetos.

3.1 Análise e projeto estruturados

Vários autores participam da corrente de análise, projeto e programação **estruturados**:
1979 - Tom DeMarco: Análise estruturada (DEMARCO, 1989)
1982 - Gane e Sarson: Análise estruturada (GANE & SARSON, 1983)
1985 - Ward e Mellor: Análise estruturada para sistemas tempo real (WARD & MELLOR, 1986)
1989 - Yourdon: Análise estruturada moderna (YOURDON, 1990)

Na análise e projeto estruturados, o processo a ser informatizado é visto como um conjunto de funções com dados de entrada, processamento e dados de saída, ou seja, a ênfase esta em funções que agem sobre dados. Estas funções podem ser decompostas em subfunções (decomposição funcional). As principais características são:

- preocupação com a modularidade e coesão;
- desenvolvimento em diferentes níveis de abstração (*top-down*).

Os principais diagramas empregados nas diversas metodologias estruturadas são:

- dicionários de dados, modelagem do fluxo de dados (DFD);
- modelos de dados: diagrama entidade e relacionamento (DER) e modelo entidade-relacionamento (MER).

[1] http://www.omg.org/technology/documents/formal/uml.htm

3.2 Análise e projeto orientados a objetos

Análise, projeto e programação **orientados a objetos**: Coad e Yourdon(1979), Rumbaugh(1991), Grady Booch(1991), Jacobson(1992). Diferentemente da análise e projeto estruturados, na orientação a objetos o processo a ser informatizado é visto como um conjunto de objetos que interagem para realizar as funções. As vantagens do modelo OO são:

- ◇ maior grau de abstração;
- ◇ maior encapsulamento;
- ◇ modelos apoiados em conceitos do mundo real;
- ◇ reutilização (reusabilidade).

Neste curso, não é abordado o ciclo de vida de desenvolvimento de software que são inúmeros: cascata, iterativo, incremental, ágil, extremo e outros. No entanto, as fases clássicas do ciclo de vida são utilizadas (engenharia de requisitos, análise, projeto, implementação, testes, manutenção e operação).

4 OBJETO E CLASSE

Apresenta-se uma breve revisão de objeto e classes de objeto assim como a notação UML de ambos.

4.1 Objeto

É uma abstração que representa uma entidade do mundo real pode ser algo concreto (computador, carro) ou abstrato (transação bancária, histórico, taxa de juros). Um objeto num sistema possui três propriedades: estado, comportamento e identidade.

- ◇ **Estado**: definido pelo conjunto de propriedades do objeto (os atributos) e de suas relações com os outros objetos. É algo que muda com o tempo, por exemplo, um objeto *turma* pode estar no estado aberto ou fechado. Inicia no estado aberto e fecha quando 10 alunos fazem inscrição.
- ◇ **Comportamento**: como um objeto responde às solicitações dos outros e tudo mais o que um objeto é capaz de fazer. É implementado por um conjunto de operações. Ex. objeto turma pode ter operações *acrescentar aluno* ou *suprimir aluno*.
- ◇ **Identidade**: significa que cada objeto é único no sistema. Por exemplo, o objeto turma *Tecno-OO manhã* é diferente do objeto *Tecno-OO tarde*.

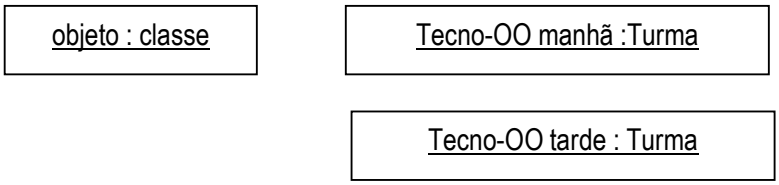


Figura 1. Notação de objeto em UML

4.2 Classe

Uma classe é uma descrição de um conjunto de objetos com propriedades, comportamento, relacionamentos e semântica comuns. Uma classe pode ser vista como um esqueleto/modelo para criar objetos.

Exemplo: classe turma

- ◇ Atributos: sala, horário
- ◇ Operações: obter local, adicionar estudante, obter horário

Dicas

- ◇ Classes devem encerrar uma só abstração do mundo real. Por exemplo, uma classe estudante contendo também o histórico do estudante não é uma boa solução. Melhor é dividi-la em duas classes: estudante e histórico.
- ◇ Utilizar substantivos para nomear as classes.
- ◇ Nas fases iniciais de desenvolvimento, pode-se suprimir os atributos e os métodos deixando somente os compartimentos.

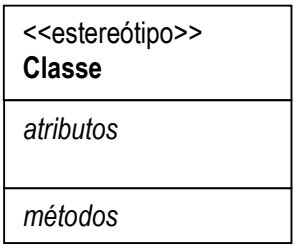


Figura 2. Notação UML para classe.

5 EXERCÍCIOS

1. Um usuário deseja uma calculadora que efetue as quatro operações básicas. As expressões permitidas são binárias envolvendo apenas dois números, por exemplo, *2 + 3.5* ou *3 * 3.2*. Identifique os objetos, seus métodos e atributos.
2. Seguindo a abordagem de orientação a objetos, identificar no enunciado abaixo os objetos e usuários do sistema. Liste os nomes dos objetos, seus atributos e os usuários do sistema. Faça o mesmo para a análise e projeto estruturados identificando as grandes funções e suas decomposições.

UMA LOCADORA de veículos necessita de um sistema para facilitar o atendimento a seus clientes. Os carros são classificados por tipo: popular, luxo e utilitário. As informações que interessam à locadora sobre cada um dos veículos são: placa do carro, tipo e valor diário do aluguel.

Os funcionários da locadora são responsáveis pelo cadastro dos clientes e dos veículos. Eles também fazem as locações e encerram as mesmas. Há clientes especiais e comuns. Os especiais têm direito a uma taxa de desconto e um valor de quilometragem extra nas suas locações. Um cliente é identificado pelo nome, número do cartão de crédito e data de expiração.

II NOÇÃO GERAL DE ANÁLISE E PROJETO OO

O objetivo deste capítulo é apresentar de maneira geral o método de análise e projeto de sistemas orientados a objetos utilizado durante o curso. São descritas as fases principais do método e os diagramas UML mais indicados para cada uma delas. Este método é uma simplificação do RUP (*Rational Unified Process).*

1 VISÃO GERAL

No curso, seguiremos o seguinte método:

1. Análise de **requisitos**: lista de requisitos funcionais e não-funcionais e diagrama de Casos de Uso.
2. Levantamento das **classes** candidatas: montar o diagrama de classes com um levantamento preliminar das classes, com atributos, métodos e relações (quando possível).
3. Estudo da **interação** entre objetos: diagramas de interação
4. **Refinamento** do diagrama de classes
5. Definição do **comportamento** de classes com estado através de máquinas de estados e diagrama de atividades
6. Modelo de implantação
7. Modelo de implementação
8. Codificação

Os passos de 3 a 5 se repetem até que se chegue próximo da implementação. Eventualmente é preciso revisar os requisitos e verificar as implicações das mudanças no projeto.

2 ANÁLISE DE REQUISITOS

Consiste em determinar os serviços que o usuário espera do sistema e as condições (restrições) sob as quais o sistema será desenvolvido e operar. As **necessidades do usuário** podem ser muito variadas, o analista deve ser capaz de retirar os requisitos funcionais e não-funcionais destas necessidades:

- **Funcionais**: lista de serviços que o sistema deve oferecer ao usuário.

◇ **Não funcionais**: propriedades e características desejadas do sistema relativas à capacidade de armazenamento, tempo de resposta, configuração, uso (ex. uso intuitivo), confiabilidade, etc.

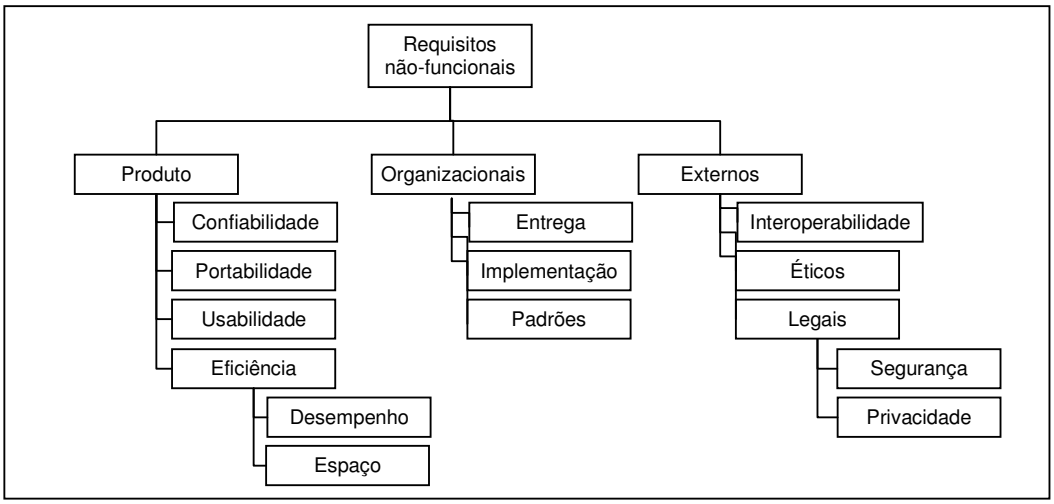


Figura 3: Taxonomia de requisitos não-funcionais (extraída de http://www.csc.liv.ac.uk/~igor/COMP201/files/SE_L4.ppt#289,15,Non-functional requirement types)

2.1 Papel dos Casos de Uso na Análise de Requisitos

Casos de uso representam funcionalidades completas para o usuário e não, funcionalidades internas do sistema. Outro ponto importante é que o diagrama de casos de uso é um artefato de comunicação entre cliente, usuários e desenvolvedores. Por ser extremamente simples e, consequentemente, de fácil compreensão, incentiva a participação do cliente e usuários no processo de desenvolvimento. Também serve como um contrato entre a equipe/empresa desenvolvedora e o cliente.

2.2 Casos de Uso

A coleção de casos de uso representa todos os modos pelos quais o sistema pode ser utilizado pelos atores envolvidos. Um caso de uso é uma seqüência de ações realizadas colaborativamente pelos atores envolvidos e pelo sistema que produz um resultado significativo (com valor) para os atores. Um ator pode ser um usuário ou outro sistema.

Para uma calculadora de linha de comando cujo objetivo é executar expressões aritméticas (ex. -2 + 3*5), o diagrama de casos da figura 4 pode ser considerado adequado.

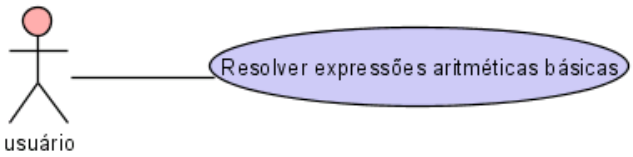


Figura 4. Diagrama de casos de uso para a calculadora.

O diagrama de casos de uso é apenas um panorama visual das funcionalidades do sistema, é necessária uma descrição textual para detalhar os casos de uso. A tabela 1 ilustra esta documentação para o caso de uso resolver expressões aritméticas.

TABELA 1: DOCUMENTAÇÃO PARA O CASO DE USO RESOLVER EXPRESSÕES ARITMÉTICAS BÁSICAS.

Nome do caso de uso	Efetuar expressões aritméticas básicas

Descrição	Permite resolver expressões envolvendo números inteiros e reais e as operações básicas de soma, subtração, multiplicação e divisão sem parênteses.
Ator Envolvido	Usuário
Pré-condições	Sistema deve estar em execução aguardando por uma expressão
Pós-condições	Expressão aritmética resolvida ou abandono da expressão pelo usuário.
Fluxo básico	
Usuário	**Sistema**
	{Solicita expressão}
	Solicita a expressão. (A2)
Fornece a expressão	
	{Valida expressão}
	Avalia se a expressão é sintaticamente correta (A1)
	{Calcula valor}
	Calcula o valor da expressão.
	{Mostra o resultado}
	Informa o resultado da expressão.
	{Fim} Fim do caso de uso.
Fluxos alternativos	
A1: em **{Valida expressão}**	Se o usuário fornecer uma expressão sintaticamente incorreta, informá-lo sobre o erro e retomar o fluxo básico em **{Solicita expressão}**
A2: em **{Solicita expressão}**	O usuário pode decidir encerrar o caso de uso sem fornecer uma expressão. Nesse caso retomar o fluxo básico em **{Fim}**

Portanto, a saída da fase de análise de requisitos é composta por:

- ◇ lista de requisitos funcionais e não-funcionais;
- ◇ diagrama de casos de uso e definições textuais dos casos.

3 ANÁLISE E PROJETO

Análise é a solução conceitual dada ao problema. Marca o início da definição informática, mas sem levar em conta detalhes da implementação tais como a linguagem a ser utilizada e o sistema gerenciador de banco de dados. Preocupa-se principalmente com as classes do domínio do problema e suas relações e também com os casos de uso.

Projeto é a solução informática dada ao problema. A separação entre análise e projeto é tênue, pois o projeto acaba sendo o resultado de sucessivos refinamentos do modelo conceitual de análise.

3.1 Diagramas de Interação

Há vários tipos de diagramas de interação na UML. Exemplifica-se o uso do diagrama de seqüência cuja utilidade é estudar as interações entre os objetos com o objetivo de refinar o diagrama de classes, identificando relações entre classes, seus métodos e atributos. A figura 5 mostra um cenário onde o usuário fornece uma expressão aritmética sintaticamente correta.

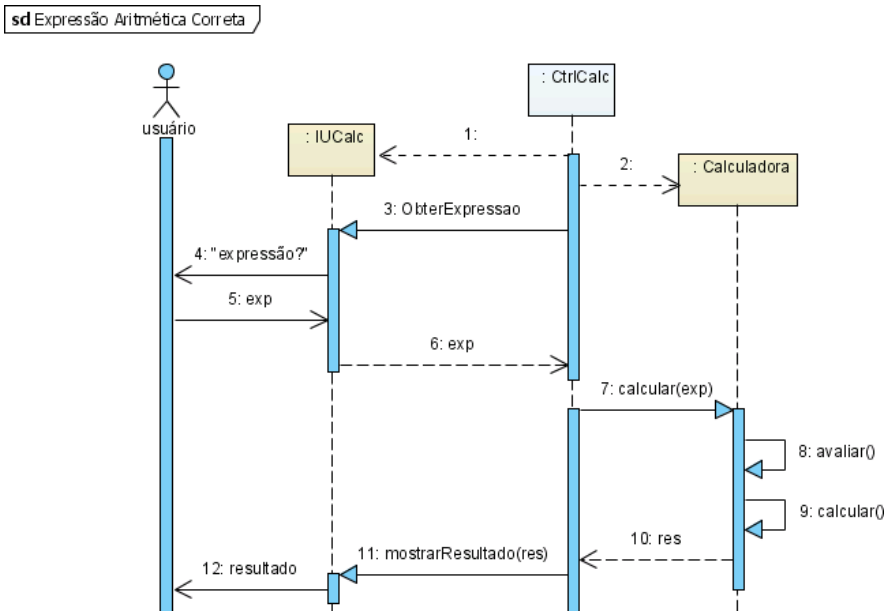


Figura 5. Diagrama de seqüência para a calculadora.

3.2 Refinamento do Diagrama de Classes

A partir das informações do diagrama de seqüência, é possível:

- identificar métodos associados às classes. Por exemplo, a classe IUCalculadora deve ter um método *mostrarResultado(<resultado>)*.
- identificar as relações entre classes. Pelo diagrama de seqüência, fica clara a existência de uma relação de dependência entre a classe de controle e as duas outras como ilustra a figura 6.

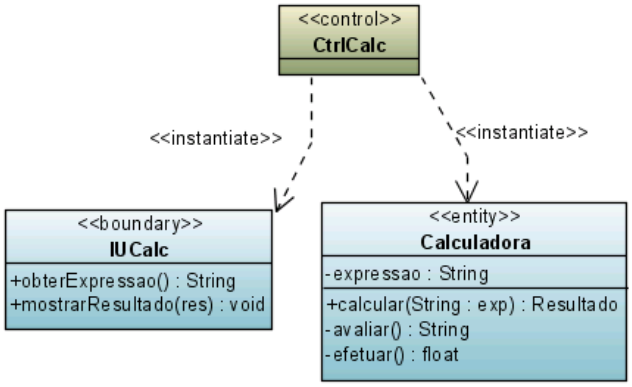


Figura 6. Diagrama de classes.

3.3 Definir o Comportamento das Classes

Nem todas as classes de um sistema possuem mais de um estado. Para as classes mais complexas, podemos especificar seus comportamentos utilizando máquinas de estado. A figura 7 mostra uma possível máquina de estados para a calculadora.

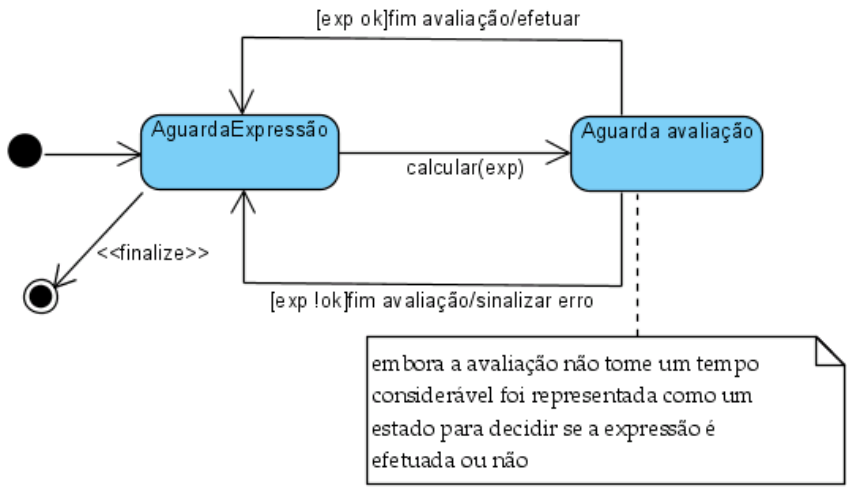


Figura 7. Máquina de estados para calculadora.

Os métodos de uma classe podem ainda ser detalhados por meio de um **diagrama de atividades** como ilustra figura 8.

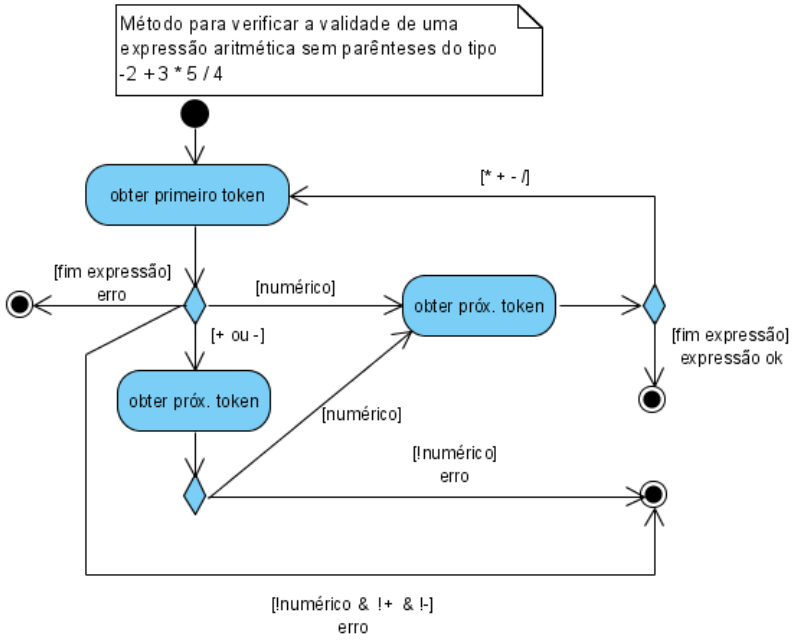


Figura 8: diagrama de atividades - detalhe de um método para verificar a sintaxe de expressão aritmética.

3.4 Implantação

O diagrama de implantação representa as necessidades de hardware e sofware básico (ex. servidores). Para tornar o diagrama mais realista, a figura 9 supõe que a calculadora é um serviço ofertado por um servidor de aplicações Web.

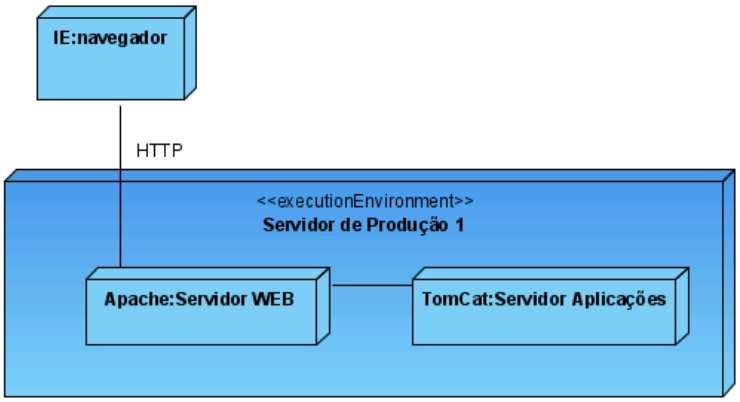


Figura 9: diagrama de implantação (*deployement*).

3.5 Componentes do Sistema

O objetivo é documentar os componentes do sistema (fontes, bibliotecas) e suas relações. A figura 10 ilustra o diagrama de componentes para a calculadora e mostra a dependência entre seus componentes.

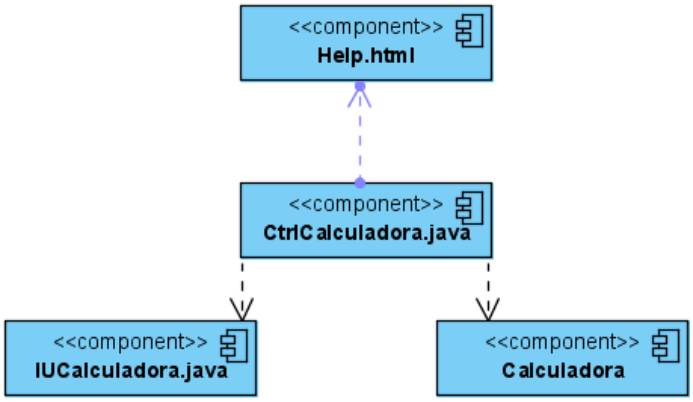


Figura 10: diagrama de componentes para a calculadora.

4 MODELAGEM ESTRUTURAL E COMPORTAMENTAL

Com esta rápida introdução à UML, é possível observar que alguns diagramas são mais indicados para modelar a estrutura do sistema e outros, o comportamento. A figura 11 mostra esta divisão.

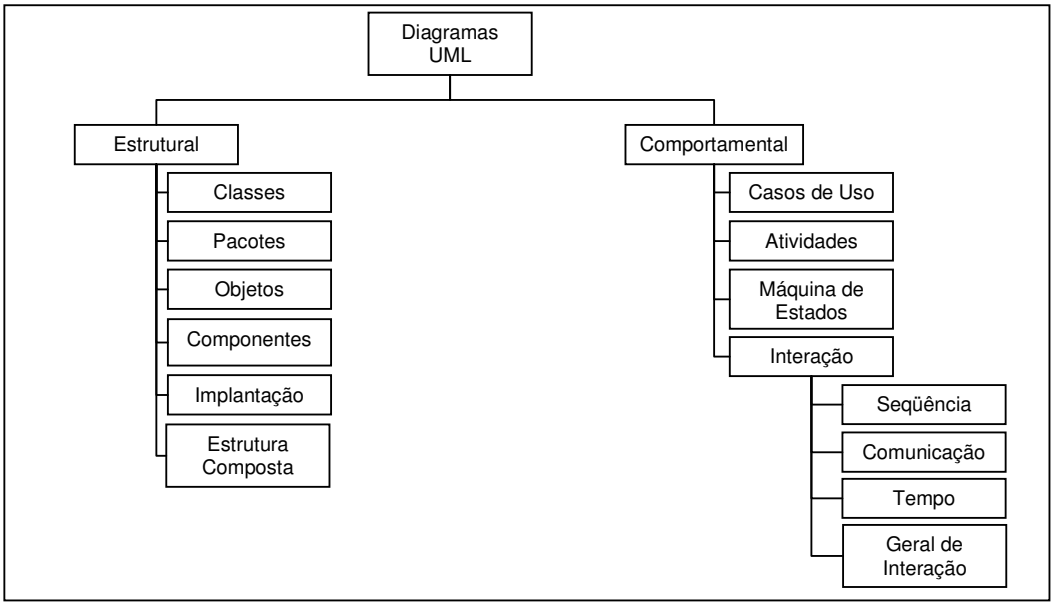


Figura 11. Diagramas estruturais e comportamentais da UML.

Segue uma breve descrição dos diagramas UML ainda não descritos neste documento:

Pacotes: representa uma coleção de classes que juntas formam uma unidade. Também pode servir para agrupar um conjunto de casos de uso com similaridades funcionais. Os pacotes podem apresentar relações, por exemplo, um pacote de classes pode depender de outro para executar suas funções.

Objetos: É um instantâneo da execução do sistema, retrata os objetos instanciados e suas relações em um dado momento.

Componentes: segundo a definição de (OMG, 2007, pg. 146), um componente é um módulo ou parte de um sistema que encapsula seu conteúdo (comportamento e dados). Um componente exibe seu comportamento através de interfaces bem definidas e pode depender de outros componentes.

***Deployement* (Implantação ou Distribuição):** para representar a arquitetura física do sistema, ou seja, para representar as relações entre os componentes (artefatos) e os locais de execução (nodos: máquinas ou sistemas servidores).

Estrutura Composta: "descreve a estrutura interna de uma classe ou componente, detalhando as partes internas que o compõe como estas se comunicam e colaboram entre si" (Guedes, 2004).

Atividades: pode ser utilizado para diversos fins, um deles é a especificação mais detalhada de métodos complexos ou do encadeamento dos casos de uso.

Interação.Comunicação: mostra as interações entre uma coleção de objetos sem a linha do tempo (pode ser obtido do diagrama de seqüência e vice-versa).

Interação.Tempo: mostra o estado de um objeto ao longo do tempo.

Interação.Geral: é a fusão do diagrama de atividades com o de seqüência. Permite fazer referência a diagramas de seqüência e combiná-los com controle de fluxo (ex. pontos de decisão, *forks e joins*).

III MODELO DE CASOS DE USO

O modelo de casos de uso (que é mais do que o diagrama) é o principal resultado da fase de análise de requisitos. Diagramas de casos de uso são utilizados para representar de forma panorâmica os requisitos funcionais de um sistema do ponto de vista do usuário. Cabe salientar que há outras utilizações possíveis para o diagrama de casos de uso, tal como modelagem do negócio, porém, o foco nesta seção está na representação de requisitos funcionais do usuário.

1 DEFINIÇÃO

É um diagrama utilizado na análise de requisitos com objetivos claros:

1. Compreender o problema.
2. Delimitar o sistema (quem está no entorno).
3. Definir as funcionalidades oferecidas ao usuário (não há preocupação com a implementação).

> Diagrama de casos de uso é uma ferramenta de comunicação entre clientes, usuários e desenvolvedores para discutirem e definirem as funcionalidades que devem ser realizadas pelo sistema.

Os elementos básicos de um diagrama de casos de uso são:
- atores,
- casos de uso e
- relações entre os mesmos.

2 ATORES

- Representam **papéis** desempenhados por usuários ou qualquer outra entidade externa ao sistema (ex. hardware, outros sistemas)
- Podem iniciar casos de uso
- Podem prover e/ou receber informações dos casos de uso

Figura 12: Notação UML para ator.

Como encontrar atores de um sistema

- Examinar o problema procurando por pessoas ou sistemas do entorno.
- Quais as pessoas ou departamentos interessados num determinado requisito funcional?
- Quem irá suprir o sistema com informações e quem irá receber informações do sistema?
- Quais os recursos externos utilizados pelo sistema?
- Uma pessoa desempenha diferentes papéis?
- O sistema interage com outros sistemas já existentes?

Como saber se um ator foi bem escolhido?

É um processo iterativo, a primeira tentativa dificilmente será a definitiva. Por exemplo, um aluno calouro é diferente de um veterano – são atores diferentes? SIM, se eles utilizam o sistema de maneiras diferentes e NÃO, caso contrário.

3 CASOS DE USO

A coleção de casos de uso representa todos os modos de execução do sistema do ponto de vista do usuário. Um caso de uso é uma seqüência de ações que produz um resultado significativo (com valor) para um ator.

- Quais são as tarefas de cada ator?
- Que informações o ator obtém do sistema?
  - Quem as fornece?
  - Quem as captura?
- Algum ator precisa ser informado sobre eventos produzidos pelo sistema?
  - Sim => casos de uso de registro e notificação
- Há eventos externos que devem ser notificados ao sistema?
  - Sim => devem haver casos para que os atores possam notificar o sistema

3.1 Descrição

Não há descrição padrão definida pela UML. Em geral o diagrama é bastante informal e sua estruturação (relação entre casos) não deve ser levada ao extremo. É importante ressaltar que o diagrama de casos de uso é uma forma de visualizar os casos e não de descrevê-los detalhadamente. Portanto, o diagrama por si só não é suficiente. Os casos de uso são descritos normalmente pelos seguintes elementos:

- nome
- descrição
- requisitos (*requirements*): são os requisitos funcionais providos pelo caso de uso
- restrições (*constraints*), tais como pré e pós-condições e condições invariantes:
  - Pré-condições: define o que deve ser verdadeiro para que o caso de uso seja iniciado. Por exemplo, num sistema bancário, para o caso de uso *Abrir conta-corrente*, uma pré-condição é apresentar CPF sem restrições (aprovação do pedido)
  - Pós-condições: o que se torna verdadeiro pela execução do caso de uso. No mesmo caso de uso acima, o sistema pode se encontrar em um dos seguintes estados: conta aberta e com um depósito inicial ou conta não-aberta por reprovação do CPF.
  - Invariantes: condições que são verdadeiras durante a execução do caso de uso.
- fluxos de eventos: descrição de interações entre atores e sistema que ocorrem durante a execução do caso de uso.
- outras informações: data, autor, etc.

3.2 Fluxo de Eventos

Um **fluxo de evento** descreve *i)* como o sistema e os atores colaboram para produzir algo de valor aos atores e *ii)* o que pode impedir sua obtenção. Um **fluxo** descreve um caminho entre muitos possíveis visto que um caso de uso pode ser executado de vários modos.

Há fluxos **primários ou básicos** (fluxo normal de eventos) e **alternativos** (o que fazer se…). Para descrevê-los, é possível se inspirar na situação em que uma pessoa explica um caminho à outra. Primeiro, o fluxo básico é explicado, depois, as alternativas.

> Para ir ao churrasco, pegue a BR116 na direção São Paulo. Logo após o clube Santa Mônica, tem um retorno por baixo da pista. Faça o retorno e continue reto (não retorne à BR). Continue nesta estradinha asfaltada por 1 km, no entroncamento pegue a estrada de terra à direita, ande cerca de 500m, você verá um grande eucalipto e uma araucária. A entrada da chácara é entre os dois. Não se esqueça de trazer o pinhão. // ***primário***
>
> Se estiver chovendo muito, os 500m na terra podem ser bem difíceis porque o barro é mole. Neste caso, siga reto no entroncamento (ao invés de virar à direita) e na próxima a direita pegue a rua de paralelepípedos. Ande cerca de 1 km e depois vire na segunda a direita que vai desembocar na frente da chácara. // ***alternativo 1***
>
> Se você for comprar o pinhão no caminho, logo depois de fazer o retorno da BR tem uma venda. Se estiver fechada, um pouco mais a frente, tem um senhor da chácara Pinhais que também vende. Se não encontrar pinhão, não tem problema. // ***alternativo 2***

Fluxos **documentam as responsabilidades**, ou seja, como as responsabilidades especificadas nos casos de uso são divididas entre sistema e atores. No desenrolar do projeto, as responsabilidades atribuídas ao sistema devem ser distribuídas entre os objetos que compõem o sistema.

Nas **fases iniciais de análise** é bom concentrar-se nos fluxos básicos (cerca de 80% do tempo de execução de um sistema é ocupado pelos casos primários) e somente identificar os casos secundários.

3.2.1 Fluxo Básico

Um fluxo básico representa o que ocorre normalmente quando o caso de uso é executado. A descrição do fluxo básico deve conter (Bittner e Spencer, 2003):

- ator e o evento que o mesmo dispara para iniciar o caso;
- a interação normal (sem tratamento de exceções) entre ator e sistema;
- descrição de como o caso termina.

Exemplo: considere um sistema onde o cliente realiza compras on-line num site utilizando um carrinho de compras virtual. O projetista do sistema previu um caso chamado *buscar produtos e fazer pedido* especificado pelo fluxo básico seguinte - extraído de (Bittner e Spencer, 2003):

> **NOME**: Buscar produtos e fazer pedido
>
> **DESCRIÇÃO**: Este caso descreve como um cliente usa o sistema para visualizar e comprar produtos disponíveis. Para encontrar um produto, o cliente pode pesquisar o catálogo por tipo de produto, fabricante ou por palavras-chaves.
>
> **PRÉ-CONDIÇÕES**: o cliente está logado no sistema.
>
> **PÓS-CONDIÇÕES**: o cliente realiza uma compra ou não.
>
> **FLUXO BÁSICO DE EVENTOS**

1. O caso de uso inicia quando o ator *cliente* escolhe a opção de consultar o catálogo de produtos.
2. **{Mostrar catálogo de produtos}**
3. O sistema mostra os produtos oferecidos, ressaltando os produtos cujas categorias constam no perfil do cliente.
4. **{Escolher produto}**

5. O cliente escolhe um produto a ser comprado e define a quantidade desejada.
6. Para cada produto selecionado disponível em estoque, o sistema registra o código do produto e a quantidade solicitada reservando-a no estoque e adiciona-o ao carrinho de compras.
7. **{Produto esgotado}**
8. Os passos 3 e 4 se repetem até que o cliente decida efetuar a compra dos produtos.
9. **{Processar pedido}**
10. O sistema pergunta ao cliente se deseja fornecer informações sobre o pagamento.
11. O sistema utiliza um protocolo seguro para obter as informações de pagamento do cliente.
12. Executar subfluxo ***validar informações de pagamento***
13. **{Informações de pagamento não válidas}**
14. O sistema pergunta ao cliente se deseja fornecer informações sobre o envio das mercadorias.
15. O sistema utiliza um protocolo seguro para obter as informações de envio.
16. Executar subfluxo ***validar informações de envio***.
17. **{Informações de envio não válidas}**
18. Executar subfluxo ***efetuar transação financeira.***
19. O sistema pergunta ao cliente se deseja comprar mais produtos.
20. Se o cliente desejar comprar mais produtos, retomar o caso no ponto **{Mostrar catálogo de produtos}**, se não o caso termina.

No fluxo básico acima, pode-se notar a existência de vários elementos que serão descritos a seguir: subfluxo, pontos de extensão e fluxos alternativos.

3.2.2 Subfluxo

Um fluxo de eventos pode ser decomposto em subfluxos para melhorar a legibilidade. No entanto, é interessante evitar muitas decomposições, pois o fluxo ficará muito fragmentado e seu entendimento dificultado. Um subfluxo deve ser atômico, isto é, ou é executado na sua totalidade ou não é executado. Para referenciar um subfluxo a partir de outro fluxo usar a notação: *executar <nome subfluxo>*. No exemplo *Buscar produtos e fazer pedido*, os subfluxos seguintes são encontrados:

- S1 Validar informações de pagamento;
- S2 Validar informações de envio;
- S3 Efetuar transação financeira.

Estes subfluxos podem ser detalhados da mesma maneira que um fluxo básico, porém deve-se evitar muitas decomposições sob o risco de perder a visão geral do caso de uso.

3.2.3 Pontos de extensão

São pontos precisos num fluxo de eventos que servem para inserir comportamentos adicionais. Pontos de extensão podem ser privados ou públicos. São privados se visíveis somente dentro do caso onde foram definidos ou públicos se visíveis nos casos que **estendem** o caso onde foram definidos.

No exemplo *Buscar produtos e fazer pedido*, os pontos de extensão seguintes são encontrados:

- **{Mostrar catálogo de produtos}**
- **{Escolher produto}**

- **{Produto esgotado}**
- **{Processar pedido}**
- **{Informações de pagamento não válidas}**
- **{Informações de envio não válidas}**

Um ponto de extensão pode definir

- uma localização único dentro do fluxo, por exemplo, {**Mostrar catálogo de produtos}, {Escolher produtos}** e **{Processar pedido}**;
- um conjunto de localizações que representam um certo estado do caso de uso, por exemplo, **{Produto esgotado}** que poderia aparecer em vários pontos do fluxo de eventos;
- uma região entre dois pontos de extensão, por exemplo, **{Escolher produtos}** e **{Processar pedido}**.

3.2.4 Fluxo Alternativo

Um fluxo alternativo apresenta um comportamento opcional, de outra forma, que não é parte do comportamento normal de um caso de uso. Fluxos alternativos são utilizados para representar tratamento de exceções ou um comportamento alternativo complexo que tornaria o fluxo básico muito longo ou de difícil compreensão.

Fluxos alternativos sempre são dependentes da existência de uma condição que ocorre em um ponto de extensão de outro fluxo de eventos. Há três tipos de fluxos alternativos:

1. **Específico**: iniciam num ponto de extensão.
2. **Regional**: podem ocorrer entre dois pontos de extensão.
3. **Geral**: podem ocorrer em qualquer ponto do caso de uso.

No exemplo *Buscar produtos e fazer pedido*, os fluxos alternativos seguintes são encontrados: Tratar produto esgotado (específico) e pesquisar por palavras-chaves (regional).

A2 Tratar produto esgotado

Em **{Produto esgotado}** quando não há a quantidade requisitada do produto em estoque.

1. O sistema informa que o pedido não pode ser completamente satisfeito.
2. *// a descrição deste fluxo continua com oferta de quantidades e produtos alternativos ao cliente*
3. O fluxo de eventos básico é retomado no ponto onde foi interrompido.

A1 Pesquisar por palavras-chaves

Entre **{Mostrar catálogo de produtos} e {Escolher produto}** quando o cliente escolhe realizar uma pesquisa por palavras-chaves.

1. O sistema pergunta ao cliente pelos critérios de busca do produto.
2. O cliente fornece os critérios de busca de produto.
3. *// a descrição deste fluxo continua*
4. O fluxo de eventos básico é retomado em **{Escolher produto}**.

Não há fluxo geral para o exemplo, mas poderia ser definido da seguinte maneira: *em qualquer ponto do caso de uso Buscar produtos e fazer pedido...*

Por que representar um fluxo alternativo separadamente do fluxo básico se é possível representá-lo com um se (if) no fluxo básico?

- Fluxos alternativos são opcionais e descrevem comportamentos que estão fora do comportamento normal esperado.
- Nem todos os fluxos alternativos representam funcionalidades essenciais, muitos deles podem não ser necessários, podem ser muito caros ou não prover funcionalidades importantes o suficiente para dispêndio de esforços de desenvolvimento.
- Fluxos alternativos permitem adicionar funcionalidade ao fluxo básico de maneira incremental ou remover funcionalidade à medida que tempo e dinheiro se esgotam.

Por exemplo, qual a importância de *realizar pesquisas por palavras-chaves* no exemplo em uso? Se for apenas uma das alternativas de busca não inviabiliza a funcionalidade do fluxo básico como um todo. Agora se alguém perguntar qual a importância do fluxo básico *buscar produtos e realizar pedido*, é fácil ver que não pode ser deixado de fora.

3.2.5 Diagrama de atividade

Um diagrama de atividade pode ser empregado para representar os fluxos de eventos de um caso de uso. Sua utilização não suprime a descrição textual, pelo contrário, ele deve ser visto como uma ilustração simplificada da descrição textual. Se todos os detalhes da descrição textual forem colocados no diagrama, este ficará extremamente poluído e perderá sua utilidade: tornar o caso de uso mais compreensível aos leitores.

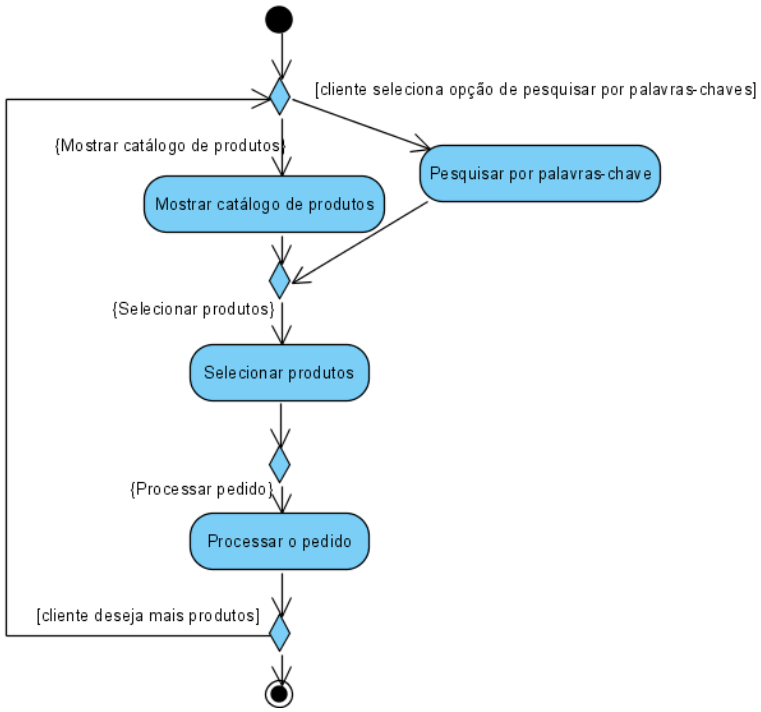


Figura 13: Exemplo de diagrama de atividades.

3.2.6 Cenários

Cenários são instâncias de execução dos casos de uso. Os fluxos alternativos representam as possibilidades de execução de um caso de uso. No exemplo *buscar produtos e realizar pedido*, o fluxo alternativo *pesquisar produtos por palavras-chaves* é uma alternativa à simples visualização do catálogo

de produtos, logo há pelo menos dois caminhos possíveis de execução. Um cenário representa um desses caminhos (figura 14).

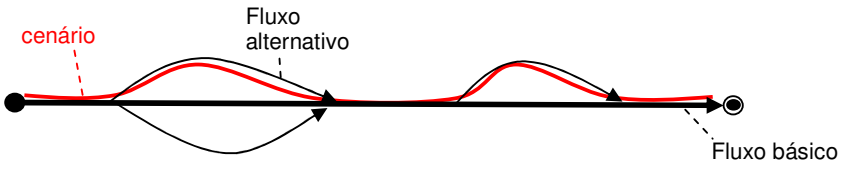


Figura 14: Representação esquemática de um cenário.

Cenários são importantes para definir casos de teste e para desenvolvedores pensarem sobre como o sistema será utilizado. Podem ser documentada adicionando-se informação às descrições dos casos de uso ou como parte da descrição dos testes. Não há necessidade de descrevê-los detalhadamente, basta nomeá-los e descrever o caminho a ser percorrido (por exemplo, fluxo básico, fluxo alternativo a1, fluxo básico).

3.2.7 Realizações de Casos de Uso

Um caso de uso pode ser *realizado* (projetado e implementado) de diferentes modos. Em UML há uma representação para realização de caso de uso como ilustra a figura 15. O intuito dessa representação é fazer uma ponte entre as descrições do sistema utilizadas pelas pessoas envolvidas na sua construção, mas que não participam do desenvolvimento em si, e as descrições do sistema utilizadas pela equipe de desenvolvimento.

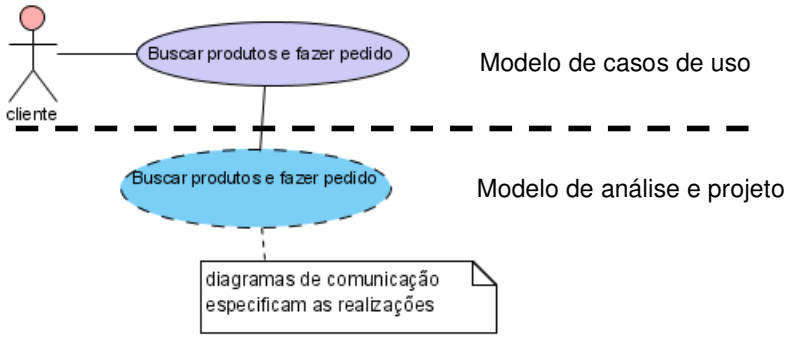


Figura 15: Realização de um caso de uso.

Diagramas de interação podem ser associados às realizações de casos de uso para especificar o fluxo de informações entre objetos que concretizam o caso. Porém, a representação de realização de caso não é muito utilizada.

4 RELAÇÕES

Há vários tipos de relações possíveis num diagrama de casos de uso, porém é importante salientar que as relações:

1. não representam a ordem de execução dos casos;
2. devem melhorar a compreensão do que o sistema deve fazer (e não como projetá-lo).

Em seguida, apresentam-se as relações mais comuns.

4.1 Associação

Associação é o tipo mais comum de relação. Pode ser utilizada entre dois atores ou entre um ator e um caso de uso. São representadas por uma linha cheia, com ou sem direção.

Ator x Ator

Relações associativas podem conectar atores para representar comunicação entre atores. A relação pode receber um nome que identifica o conteúdo da mensagem, documento ou objeto que trafega entre os atores. A figura 16 mostra uma associação entre o ator *usuário de biblioteca* que passa o *livro* ao atendente que realiza o empréstimo ou a devolução. Como não há flechas, assume-se que o atendente devolve algo ao usuário da biblioteca, provavelmente um comprovante não representado no diagrama. Não é recomendável colocar este tipo de relação no diagrama de casos de uso.

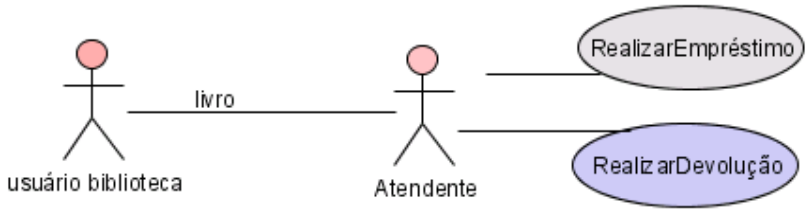


Figura 16: Exemplo de associação entre atores.

Ator x Caso

Há vários usos para associações entre atores e casos de uso:

1. indica quem inicia a comunicação, o ator ou o caso de uso (sistema);
2. indica o fluxo de informações, ou seja, quem fornece informações a quem.

Para documentar a escolha, pode-se atribuir um nome à associação. Na figura 16, há uma associação entre o *atendente* e o caso de uso *realizar empréstimo*. Observar que é bidirecional, portanto o atendente inicia a execução do caso, fornece e recebe informações do mesmo.

Associações unidirecionais deixam os diagramas mais claros, embora não sejam obrigatórias. Por exemplo, a figura 17 ilustra um mesmo diagrama que representa os requisitos de um sistema de telefonia com relações bidirecionais e unidirecionais. No diagrama superior, pode-se deduzir que o emissor inicia a chamada telefônica e, no inferior, esta informação está explícita.

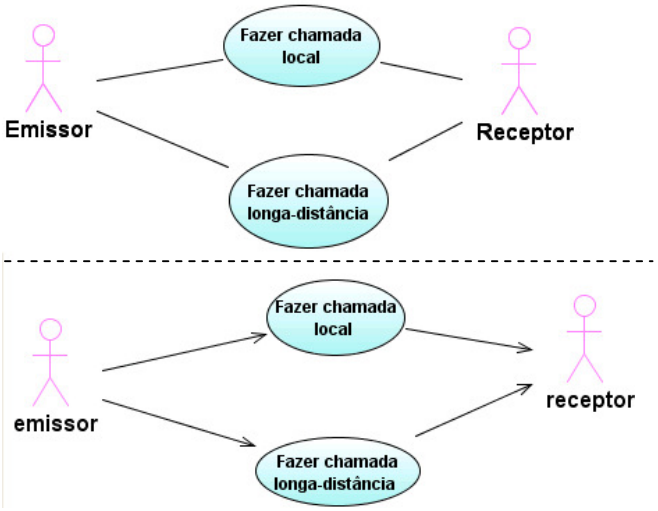


Figura 17: Associações bidirecionais e unidirecionais.

4.2 Inclusão

A relação de inclusão é utilizada entre dois casos, quando um deles inclui o outro na sua execução (subcaso). Um subcaso representa parte de um fluxo de eventos básico, isto é, um subfluxo que foi separado e representa um conjunto de ações atômico.

Ressalta-se que a existência de um subfluxo na descrição textual de um caso **não implica** sua representação no diagrama de casos de uso. A relação de inclusão (<<include>>) deve ser utilizada somente quando dois ou mais casos de uso apresentam partes idênticas nos seus fluxos de evento, caso contrário (se somente um caso de uso utilizar o subfluxo) não deve ser representado. Isto requer que os fluxos de evento sejam escritos antes de se colocar relações de inclusão no diagrama.

A figura 18 mostra um caso *efetuar matrícula* que possui subfluxos *escolher disciplinas*, *alocar alunos às turmas* e *emitir boleto*. Este último é compartilhado com o caso *Efetuar inscrição curso opcional* e, portanto, deve ser representado no diagrama. Um erro comum que adiciona complexidade ao diagrama é incluir os subfluxos *escolher disciplinas e alocar alunos às turmas* no diagrama.

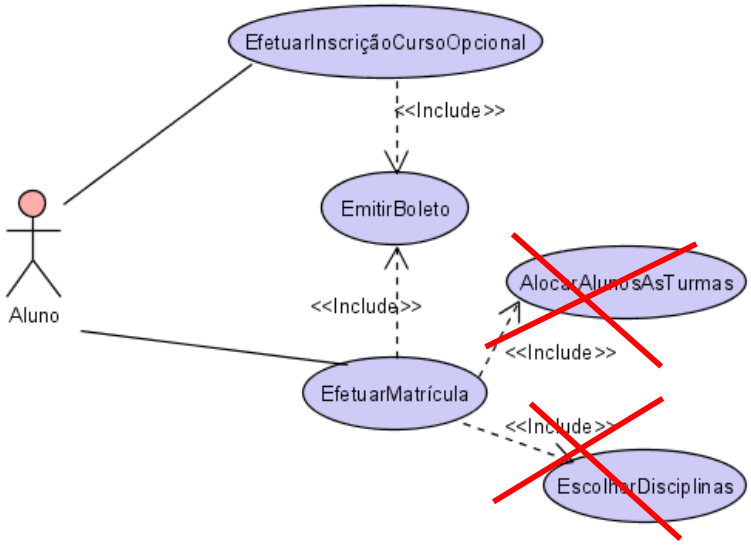


Figura 18: exemplo de inclusão de casos.

É importante ressaltar que:

- um caso de uso **nunca** deve ser incluído apenas por um caso, ou seja, não utilizar <<*include*>> para decompor o diagrama em partes;
- um caso de uso que é incluído por vários outros **não tem conhecimento** sobre quem o inclui, portanto, podem ser incluídos por qualquer caso sem sofrer modificações;
- não utilizar a relação de inclusão para representar **opções de menu**, pois o caso que faz a inclusão seria um simples despachante, todo o comportamento estaria fragmentado nos casos incluídos.

Enfim, inclusão deve ser utilizada para administrar comportamentos comuns e não para estruturar funcionalmente o diagrama.

4.3 Extensão

Um caso pode estender outro quando se deseja inserir um comportamento opcional ou excepcional disparado por alguma condição (ex. um alarme ou condição especial de algum objeto). Situações que podem levar ao uso da relação de extensão (Bittner e Spencer, 2003):

- Descrições **opcionais ao comportamento** normal do sistema: por exemplo, módulos que podem ser comprados do desenvolvedor ou de terceiros.
- Descrições de tratamento de **erros e exceções** complexos: estes tratamentos podem ser extremamente longos e ofuscar o fluxo básico.
- **Customização**: fluxos alternativos que especificam como diferentes clientes tratam certas situações no dentro do mesmo caso de uso base.
- Administração de **escopo** e de **release**: comportamentos que serão incluídos futuramente.

É importante ressaltar que:

- Um caso de uso de extensão não requer modificações no caso base (aquele que é estendido). O comportamento básico do caso base permanece intacto.
- Um caso de uso que estende um caso base conhece este último (não é muito comum um caso de uso estender mais de um caso base).
- Uma extensão nasce como um fluxo alternativo, mas nem todo fluxo alternativo vira uma extensão.
- Casos de uso que estendem assumem o controle no ponto de extensão e quando terminam devolvem o controle no mesmo ponto.

Aqui cabe uma distinção entre fluxos alternativos e casos de uso que estendem outros. Fluxos alternativos são parte do caso de uso base e têm acesso ao seu estado, pré-condições, outros fluxos existentes e pontos de extensão além daquele onde se inserem. Casos de uso que estendem conhecem apenas o ponto de extensão onde se inserem no caso estendido. Para saber se um fluxo alternativo é candidato a ser uma extensão, deve responder positivamente à questão: o sistema pode ser entregue sem a extensão?

Na figura 19, a emissão de histórico escolar é estendida pelo caso imprimir comprovante de término quando o aluno solicitante for formado. Observa-se que é um comportamento opcional que pode não ser oferecido sem prejuízo ao comportamento básico *emitir histórico escolar.*

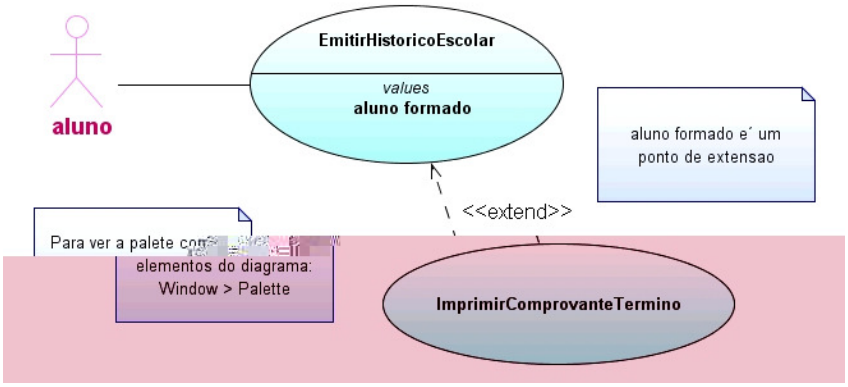


Figura 19: exemplo de relação de extensão entre casos de uso.

Nota-se na o ponto de extensão público denominado **{aluno formado}** onde o comportamento opcional *imprimir comprovante término* é inserido. É provável que existam outros pontos de extensão privados definidos nos fluxos de *emitir histórico escolar*, porém, no diagrama só os usados pelas extensões são listados. A figura 20 ilustra o diagrama de casos de uso para o exemplo *buscar produto e fazer pedido.*

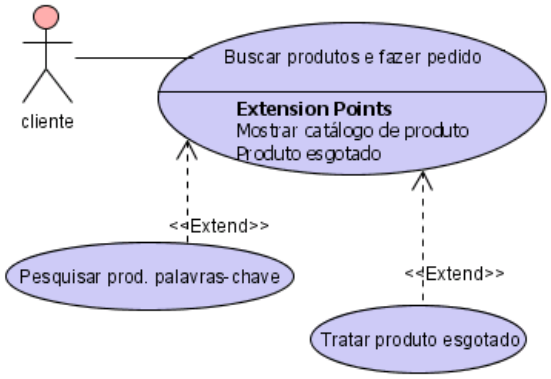


Figura 20: pontos de extensão para o caso *buscar produtos e fazer pedido.*

4.4 Generalização/Especialização

A relação de generalização/especialização pode ocorrer entre casos de uso ou entre atores.

Caso x Caso

Generalização permite especificar comportamentos genéricos que podem ser especializados para atenderem necessidades específicas. Normalmente é utilizado quando se quer descrever **famílias de sistemas**.

Por exemplo, uma empresa que desenvolve software pare terminais bancários de auto-atendimento quer expandir seus negócios para outras áreas, tais como pagamento direto em bombas de gasolina.

NOME: Realizar transação **(caso abstrato)**
DESCRIÇÃO: Permite ao usuário comprar mercadorias de um terminal automático sendo que o valor das mercadorias descontado de uma conta bancária.

PRÉ-CONDIÇÕES: (e) o cliente possui um cartão bancário; a conexão com o banco está ativa; o terminal deve ter mercadoria.

PÓS-CONDIÇÕES: (ou) O terminal retornou o cartão bancário ao cliente, entregou a mercadoria ao cliente e debitou o valor de sua conta. O terminal retornou o cartão bancário ao cliente, não entregou nenhuma mercadoria e nenhum valor foi debitado da sua conta.

FLUXO BÁSICO DE EVENTOS

1. O ator cliente insere o cartão bancário no terminal.
2. O sistema lê as informações da conta do cliente no cartão bancário
3. O sistema solicita ao cliente a senha
4. O cliente fornece a senha
5. O sistema verifica se a senha fornecida pelo cliente é idêntica à lida do cartão bancário
6. O sistema contata com o Sistema Bancário para verificar se as informações da conta do cliente são válidas
7. O sistema solicita o valor da transação
8. O sistema contata o Sistema Bancário para verificar se o cliente tem saldo para cobrir a solicitação.

{Cliente realiza a transação}

9. O sistema registra o valor da transação.
10. O sistema comunica ao Sistema Bancário que a transação foi efetuada.
11. O sistema grava no log os dados da transação: data, hora, valor e conta.
12. Término do caso de uso.

NOME: Sacar **(caso concreto)**

DESCRIÇÃO: Especializa o caso de uso realizar transação para permitir ao cliente retirar dinheiro de um terminal de auto-atendimento bancário.

PRÉ-CONDIÇÕES: (e) o cliente possui um cartão bancário; a conexão com o banco está ativa; o terminal deve ter dinheiro.

PÓS-CONDIÇÕES: (ou) O terminal retornou o cartão bancário ao cliente, entregou o dinheiro ao cliente e debitou o valor de sua conta. O terminal retornou o cartão bancário ao cliente, não entregou dinheiro e nenhum valor foi debitado da sua conta.

FLUXO BÁSICO DE EVENTOS

Em {Cliente realiza transação}

1. O sistema verifica se tem dinheiro suficiente em relação ao montante solicitado pelo cliente.
2. O sistema entrega o montante solicitado.
3. O sistema solicita que retire o dinheiro do terminal.
4. O cliente pega o dinheiro.

5. Retoma-se o caso de uso abstrato em **{Cliente realiza transação}**

NOME: Abastecer veículo **(caso concreto)**

DESCRIÇÃO: Especializa o caso de uso *realizar transação* para permitir ao cliente obter combustível de uma bomba debitando o valor de sua conta.

PRÉ-CONDIÇÕES: (e) o cliente possui um cartão bancário; a conexão com o banco está ativa; a bomba tem combustível.

PÓS-CONDIÇÕES: (ou) O terminal retornou o cartão bancário ao cliente, liberou o combustível ao cliente e debitou o valor de sua conta. O terminal retornou o cartão bancário ao cliente, não liberou combustível e nenhum valor foi debitado da sua conta.

FLUXO BÁSICO DE EVENTOS

Em {Cliente realiza transação}

1. O sistema solicita ao cliente para tirar o bico de abastecimento e libera o fornecimento de combustível.
2. O cliente enche o tanque até atingir o valor informado ou até que o tanque esteja cheio.
3. O cliente recoloca o bico na bomba.
4. Retoma-se o caso de uso abstrato em **{Cliente realiza transação}**

O diagrama de casos de uso correspondente aos casos acima é ilustrado na figura 21.

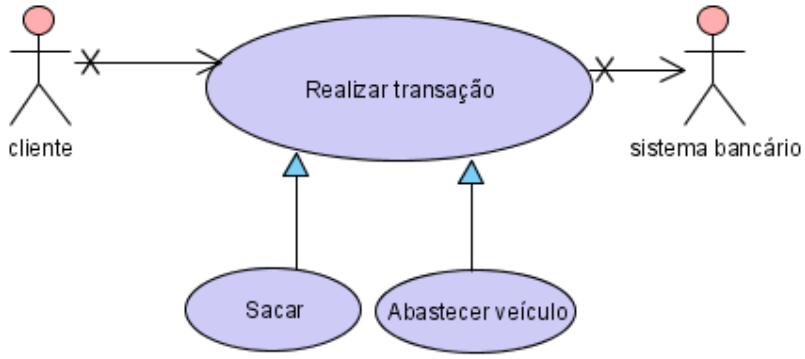


Figura 21: generalização/especialização de casos de uso.

Ressalta-se que nesta situação, são executados os casos de usos especializados. Eles apenas reusam partes do caso geral. A tabela 3 mostra as diferenças entre especialização e extensão de casos de uso.

TABELA 3: COMPARATIVO ENTRE ESPECIALIZAÇÃO E EXTENSÃO DE CASOS DE USO.

Especialização	**Extensão**
O caso de uso especializado é executado	O caso de uso base é executado
O caso de uso base não precisa ser completo e com sentido. Há várias lacunas preenchidas somente nas especializações.	O caso de uso base deve ser completo e com sentido.
O comportamento de uma execução depende unicamente do caso específico.	O comportamento de uma execução depende do caso de uso base e de todas as

	extensões que são executadas.

Ator x Ator

Especialização de atores representa que um conjunto deles possui responsabilidades ou características em comum. Algumas dicas para evitar modelagens desnecessárias:

- ◇ Não utilizar atores para representar permissões de acesso.
- ◇ Não utilizar atores para representar organogramas (hierarquias) de cargos de uma empresa.
- ◇ Utilizar atores somente para definir papéis em relação ao sistema.

Por exemplo, se num sistema de matrículas de uma universidade há casos de uso especiais para alunos de ciências exatas e para alunos de humanas, então é sinal que estes alunos são especializações de um ator genérico aluno. A figura 22 ilustra a notação UML para este caso. Observar que alunos de ciências exatas e de humanas herdam todas as associações do ator aluno.

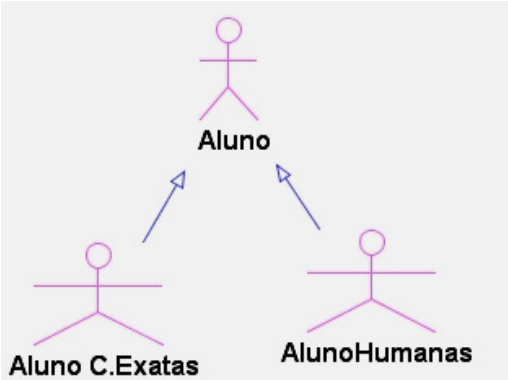


Figura 22: Especialização de atores.

5 MODELAGEM

5.1 Dicas

Casos de uso auxiliares

- ◇ Casos de uso auxiliares são frequentemente esquecidos, pois não são essenciais à funcionalidade do sistema. Porém, esquecê-los completamente pode conduzir a um sistema difícil de ser utilizado.
- ◇ Lembrar de colocar casos de uso para executar, manter e configurar o sistema, tais como: lançar e parar o sistema, incluir novos usuários, fazer *backup* das informações, incluir novos relatórios e realizar configurações.

Decomposição funcional

- ◇ Não é necessário detalhar em excesso os casos de uso. Muitos detalhes levam a decomposição dos casos em funções. O objetivo é compreender o sistema através de cenários de utilização.
- ◇ Casos de uso não são feitos para analisar (no sentido de decompor) os requisitos em requisitos menores. É um processo de **síntese** ou elaboração (e não de análise) no qual o problema não é

totalmente conhecido. Quebrá-lo em partes menores (análise) dificulta a obtenção de uma visão geral.

- ◇ Em equipes com forte competência em análise estruturada, há tendência em encontrar funções ao invés de casos de uso. Por exemplo, *fazer pedido, revisar pedido, cancelar pedido e atender pedido* podem parecer bons casos de uso. No fundo, todas estas *funções* estão relacionadas ao caso de uso *realizar pedido*.
- ◇ Decomposição funcional pode levar a um número intratável de *casos,* mesmo para pequenos sistemas, e à perda de foco no que realmente é importante no sistema (o que traz valor aos atores).
- ◇ Casos de uso não chamam outros casos de uso ou se comunicam com outros casos.

Estrutura e detalhamento

- ◇ Não estruturar demais o diagrama de casos de uso, isto é, não inclua relações entre casos de uso a não ser que sejam extremamente necessárias. O uso em excesso destas relações podem fragmentar o diagrama, diminuindo a compreensão global.
- ◇ O modelo deve ser o mais simples e direto possível.
- ◇ Não descrever o que ocorre fora do sistema. Por exemplo, interação entre atores pode ser importante para o negócio, mas se o sistema não facilita esta interação, deixe-a fora. Se for necessário esclarecer estes pontos faça um modelo do negócio e não um modelo de casos de uso.
- ◇ Não fazer casos tais como *incluir, consultar, alterar e excluir* (ICAE). Casos de uso que descreve comportamentos ICAE não adicionam muito valor à descrição da funcionalidade do sistema. Para estes tipos de comportamentos, os requisitos são bastante claros e não se deve perder tempo especificando-os. A maior parte destes comportamentos tem um padrão *mostrar campos, usuário entra com os dados, sistema valida, usuário corrige erros,...* A validação, a parte mais importante dos comportamentos ICAE, pode ser capturada no domínio do modelo (por meio de relações, cardinalidade e restrições) ou textualmente no glossário de dados.

Modelo de casos de uso é mais que um diagrama

- ◇ O diagrama de casos de uso é uma visão panorâmica dos casos de uso e, portanto, apenas um dos componentes do modelo de casos de uso. A descrição textual dos mesmos é a parte mais importante do modelo. São elementos do modelo de casos de uso: glossário, modelo do domínio e diagramas de atividades.
- ◇ Um glossário e um modelo do domínio podem evitar o excesso de detalhes nas descrições dos casos de uso. Por exemplo, ao invés de descrever a validação da entrada de dados, os campos podem ser definidos no glossário com os respectivos valores possíveis.
- ◇ Um modelo do domínio ajuda a entender as relações existentes entre as entidades do domínio e as restrições sobre as relações.

5.2 Passos

Para elaborar um modelo de casos de uso, os seguintes passos podem ser seguidos:

- ◇ Recapitular a visão do sistema (estudo de viabilidade) aos envolvidos.
- ◇ Elaborar se necessário um diagrama de contexto para definir os limites do sistema (o que está dentro e o que está fora).
- ◇ Identificar os atores do sistema.
- ◇ Identificar os casos de uso: descrevê-los e rascunhar o fluxo de eventos. Não se preocupe com fluxos alternativos. Não gaste mais do que 10 minutos para descrever cada caso de uso.
- ◇ Esboçar o diagrama de casos de uso mostrando associações entre atores e casos de uso.

- ◇ Verificar os casos de uso:
  - Há casos de uso sem conexão com requisitos funcionais? Caso haja, pode haver casos em excesso ou requisitos que não foram identificados!
  - Há requisitos funcionais sem casos? Alguns casos podem ter sido esquecidos.
  - Todos os requisitos não-funcionais foram tratados (associados aos casos de uso quando são específicos)?
  - Todos os tipos de usuários foram mapeados para um ator ao menos?
- ◇ Descrever os casos detalhadamente
- ◇ Representar os subfluxos *(<<include>>)* e fluxos alternativos (*<<extend>>*) considerados importantes no diagrama
- ◇ Verificá-los:
  - É possível eliminar os casos incluídos ou as extensões e ainda ser capaz de entender o que o sistema faz para as partes interessadas?

6 EXERCÍCIOS

1. Um aluno de uma universidade particular deve *escolher disciplinas do semestre*. Em seguida ele é *alocado às turmas* para então receber uma fatura emitida pelo sistema de faturamento com o valor a ser pago em função do número de turmas em que conseguiu vaga. *Quais são os atores e casos de uso?*

2. A secretaria de uma universidade deve cadastrar turmas, apagá-las e modificá-las e enviá-las aos departamentos acadêmicos? *Quais são os atores e casos de uso?*

3. Construir o diagrama de casos de uso e especificar os fluxos de eventos básico.

> Um cliente deseja um sistema que permite jogar jogo da velha e forca. O sistema é destinado a um usuário e deve armazenar as estatísticas de uma sessão (do lançamento ao término do sistema).
>
> Em uma sessão o usuário pode jogar diversas vezes cada um dos jogos. Ao término de cada jogo, atualizam-se as estatísticas da sessão: número de vezes que jogou velha, número de vitórias absoluto e percentual e o mesmo para forca. O usuário deseja que o painel de estatísticas esteja sempre visível.

4. Qual a diferença entre as relações de inclusão e extensão entre casos de uso?

5. Como você modelaria a situação abaixo utilizando casos de usos? Faça a descrição completa para um dos casos incluindo descrição, pré-condições, pós-condições, fluxo de eventos básico e alternativos e o diagrama de casos.

> A atividade da biblioteca centra-se principalmente no empréstimo de publicações pelos alunos. O empréstimo é registrado pelos funcionários da biblioteca que também consultam diariamente os empréstimos cujos prazos foram ultrapassados. Os alunos necessitam pesquisar os livros existentes na biblioteca. Caso um livro esteja emprestado é mostrado a data esperada de entrega.

6. Construir o diagrama de casos de uso:

> Um cliente (pessoa física ou jurídica que paga o advogado pra defendê-la ou para processar outra pessoa) procura o advogado. Se o cliente ainda não estiver cadastrado, o advogado deverá registrar seus dados pessoais.
>
> Em seguida, o cliente deve fornecer informações a respeito do processo que deseja que o advogado mova contra alguém ou que o defenda de outra pessoa. Obviamente o processo precisa ser registrado e receberá diversas adições enquanto estiver em andamento. O cliente

deve fornecer também informações sobre a parte contrária (pessoa física ou jurídica que está processando ou sendo processada pelo cliente), que deverá ser registrada, caso ainda não esteja. Observe que uma mesma pessoa física ou jurídica pode ser tanto um cliente como uma parte contrária em processos diferentes obviamente.

Um processo deve tramitar em um determinado tribunal e em uma determinada vara, no entanto um tribunal pode julgar muitos processos e uma vara pode possuir diversos processos tramitando nela. Um tribunal pode possuir diversas varas, porém um processo julgado por um tribunal só pode tramitar em varas pertencentes ao mesmo. O advogado pode achar necessário emitir relatórios de todos os processos em andamento em um tribunal e tramitando em uma vara.

Cada processo possui no mínimo uma audiência, cada audiência relativa a um determinado processo deve conter sua data e a recomendação do tribunal. Para fins de histórico do processo, cada audiência deve ser registrada.

Um processo pode gerar custas (cópias, viagens, etc.). Cada custa deve ser armazenada de forma a ser cobrada da forma contrária caso o processo seja ganho.

Este sistema deve estar integrado a um sistema de contas a pagar receber, cada custa gera uma conta a pagar. Caso o processo seja ganho, ele gerará uma ou mais contas a receber, dependendo da negociação com a parte contrária.

IV ANÁLISE DOS CASOS DE USO

Este capítulo inicia com uma breve apresentação do padrão de projeto MVC e do padrão observador que servem de ponto de partida na escolha das classes que fazem parte de um sistema (classes de análise). Em seguida, apresenta-se o levantamento das classes de análise a partir dos casos de uso.

1 ANÁLISE

O levantamento das classes de análise marca o início da construção do modelo da análise (RUP). Ocorre uma mudança na linguagem, enquanto no modelo de casos de uso a descrição do sistema era feita na linguagem do cliente/usuário, na análise emprega-se a linguagem dos desenvolvedores. A tabela 4 ilustra as diferenças entre o modelo de casos de uso e o modelo de análise segundo (Jacobson e co-autores, 1999).

TABELA 4: COMPARAÇÃO ENTRE MODELO DE CASOS DE USO E DE ANÁLISE (JACOBSON E CO-AUTORES, 1999).

Modelo de casos de uso	Modelo de análise
Linguagem do cliente/usuário	Linguagem dos desenvolvedores
Visão externa do sistema	Visão interna do sistema
Estruturado pelos casos de uso	Estruturado por classes estereotipadas e pacotes
Contrato entre cliente e desenvolvedores sobre o que o sistema deve e não deve fazer	Utilizado pelos desenvolvedores para entender qual a forma do sistema, i.e., como deve ser projetado e implementado
Pode haver redundâncias e inconsistências nos requisitos	Não deve haver redundâncias e inconsistências nos requisitos
Captura a funcionalidade do sistema	Rascunha como realizar a funcionalidade
Define casos de uso que são analisados no modelo de análise	Define realizações de casos de uso, cada uma representando a análise de um caso de uso

Por que fazer análise?

Uma pergunta recorrente é por que fazer análise ao invés de partir diretamente ao projeto e implementação? Há várias respostas que se complementam:

- Antes de projetar e implementar é preciso **especificar** os casos de uso num grau de **detalhamento** e de formalismo que não interessa aos usuários, só aos desenvolvedores.
- Análise permite obter uma **visão geral** do sistema de difícil obtenção se todos os detalhes de projeto e implementação fossem nela inseridos. Análise pode envolver grandes porções do sistema sem muitos custos se comparado ao projeto e implementação. Resultados da análise permitem planejar melhor o projeto/implementação dividindo-se o sistema em partes menores que podem ser desenvolvidos incrementalmente de maneira seqüencial ou concorrente (projetado e implementado por equipes diferentes).
- Se o sistema em desenvolvimento utiliza um **sistema legado** este último pode sofrer reengenharia para se obter o modelo de análise para que os desenvolvedores o entendam sem entrar nos detalhes tecnológicos. A reengenharia completa é um processo caro, custoso e demorado que pode não ser necessário se o sistema legado não necessita ser modificado e utiliza tecnologias obsoletas.
- Um modelo de análise pode fornecer uma **visão conceitual** do sistema independente das tecnologias empregadas. Por exemplo, um cliente pode querer que duas fornecedoras de software façam uma proposta baseando-se em uma especificação. Outro caso seria a necessidade de implementar parte de um sistema em plataformas ou linguagens diferentes devido à infra-estrutura existente em filiais distintas (por exemplo, plataformas diferentes). Ainda, sistemas

críticos (controle de aeronaves, linhas de trem) podem necessitar de programas que realizam cálculos empregando diferentes métodos para que decisões críticas só sejam tomadas quando os dois métodos produzem resultados equivalentes.

Análise ou não

Em relação aos modelos de análise, pode-se dizer que há três abordagens possíveis:

- ◇ Mantê-lo atualizado durante todo o ciclo de vida do sistema.
- ◇ Considerá-lo como um resultado intermediário e, uma vez que o projeto esteja feito, atualizar o modelo de projeto.
- ◇ Fazer análise e projeto integrados (recomendável apenas para sistemas extremamente simples), há duas possibilidades:
  - requisitos são analisados durante o levantamento ou captura: exige mais formalismo no modelo de casos de uso;
  - requisitos são analisados durante o projeto: se os requisitos forem simples e/ou bem conhecidos.

Realização dos casos de uso

Segundo (Jacobson e co-autores, 1999, pg. 221), se o modelo de análise não é atualizado durante o ciclo de vida do software, ou seja, foi criado apenas para possibilitar a elaboração de um bom projeto, não há realização de casos de uso na análise. Neste caso, a realização do caso de uso ocorrerá somente no projeto. As realizações de casos de uso de projeto podem ser diretamente conectadas ao modelo de caso de uso por uma relação de dependência *<<trace>>* (ao invés de estarem ligadas à realização do caso de uso de análise), conforme ilustra a Figura 24.

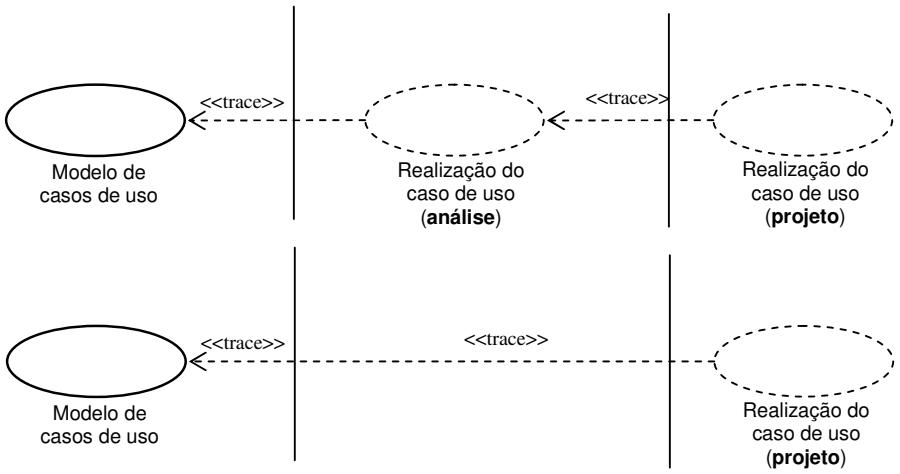


Figura 24: realização dos casos de uso - análise e projeto ou somente no projeto.

Método de Análise

De acordo com o RUP, a análise de casos de uso é composta por:

- ◇ Para cada caso de uso considerado em uma iteração
  - Criar uma realização de caso de uso
  - Complementar a descrição do caso de uso
  - Levantar as classes de análise examinando o comportamento do caso de uso
  - Distribuir o comportamento do caso de uso entre as classes de análise
- ◇ Para cada classe de análise
  - Descrever as responsabilidades da classe

- Definir atributos
- Definir relações entre classes

2 PADRÃO MVC

MVC (*Model, View, and Controller)* é um padrão de arquitetura de software. No contexto da engenharia de software, um padrão é uma solução "comprovada" a um problema recorrente. Segundo (Gamma e co-autores, 1995), padrões especificam **abstrações** que estão acima do nível de classes, objetos isolados ou de componentes. As vantagens ao se utilizar padrões de projeto são:

◇ ter um vocabulário comum para a discussão de problemas e soluções de projeto (Gamma et al. 1995);
◇ facilitar documentação e manutenção da arquitetura do software (Buschmann et al., 1996);
◇ auxiliar o projeto de uma arquitetura com determinadas propriedades (Buschmann et al., 1996).

A idéia do padrão MVC é desacoplar ao máximo a apresentação da aplicação (*view*) da lógica do negócio (*business model*). Muitos sistemas tornam-se complexos, pois misturam código da apresentação com o código da lógica do negócio. Qualquer mudança requerida pelo usuário na forma de apresentação das informações exige também mudança na lógica do negócio. A camada de apresentação envolve interfaces e também relatórios.

Atualmente, é freqüente a utilização de diversas interfaces para um mesmo negócio em função do tipo de usuário ou do tipo de dispositivo de saída. Se para cada tipo de usuário/tipo de dispositivo de saída fizermos uma lógica de negócio diferente, estaremos replicando código desnecessariamente. Por exemplo, uma aplicação que mostra o extrato bancário de uma conta corrente em diversos tipos de dispositivo tais como PC, caixa automático, Palm e celular. A lógica do negócio é sempre a mesma, juntar as transações executadas em uma conta em um período, mudando apenas a forma de mostrar os dados.

O padrão MVC propõe a divisão da aplicação em três partes (ou camadas):

◇ **Modelo do negócio** (*model)*: contém os dados do negócio e as regras do negócio que ditam o acesso e a modificação dos dados. De forma mais prática, encapsula os dados e os comportamentos do negócio e salva os mesmos **sem se preocupar** em como serão mostrados.

◇ **Visão** (*view*): é responsável pela interação com o usuário e por apresentar as diversas visões que dos dados do negócio. Não se preocupa em como os dados foram obtidos, apenas em apresentá-los.
◇ **Controle** (*controller*): comanda o fluxo de obtenção, encaminhamento e apresentação das informações fazendo a intermediação entre as camadas de visão e de modelo.

A Figura 25 ilustra o modelo MVC. Há duas formas básicas, na primeira (a) os eventos/respostas da interface são tratados diretamente pela camada de visão e na segunda, pelo controle que então seleciona a visão apropriada. O controle interpreta os eventos e informações fornecidas pelo usuário e chama as ações que poderão mudar o estado do modelo do negócio. Em seguida, as alterações do modelo são retratadas pela camada da visão podendo ter o controle como intermediário ou não. Normalmente, **há um controlador para cada caso de uso do modelo do negócio**.

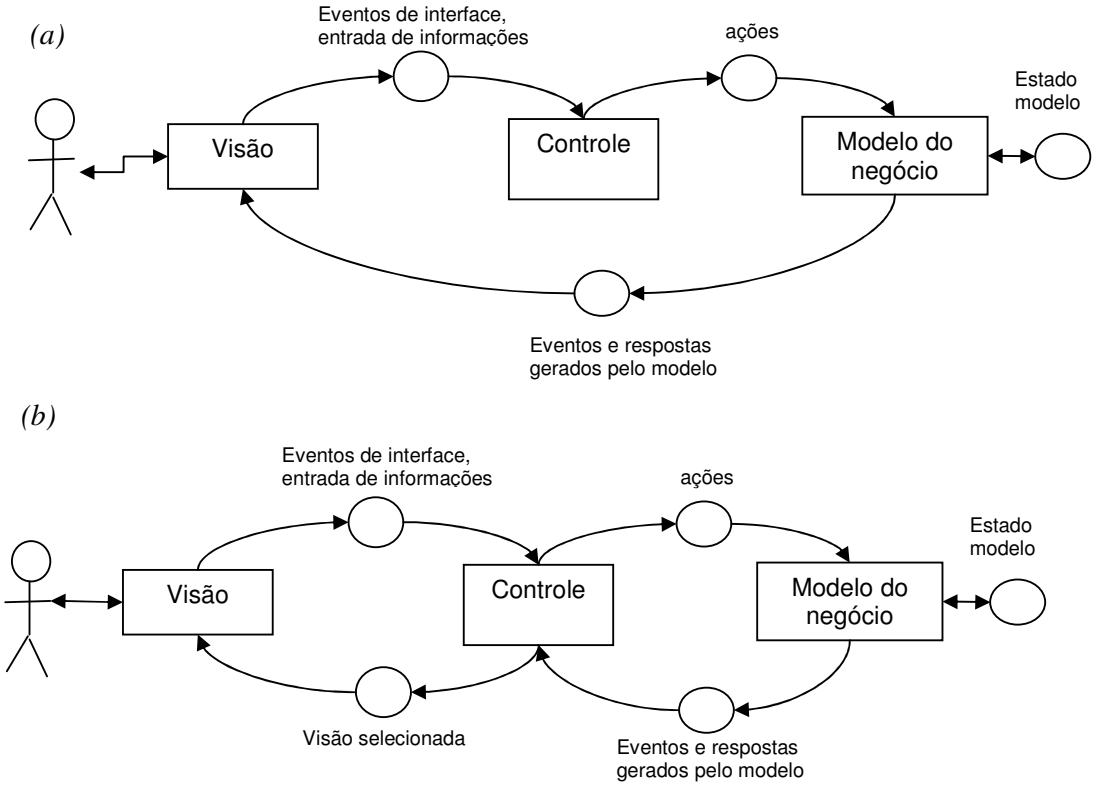


Figura 25. Modelo MVC nas duas formas possíveis.

O modelo MVC é bastante utilizado em aplicações WEB. Um serviço pode ser chamado a partir de diferentes clientes tais como celulares, PCs e Palms. A lógica do negócio, no entanto, permanece a mesma independente do cliente. Normalmente, nas aplicações WEB o servidor escolhe a visão mais apropriada como mostra a Figura 26. Observar que os servidores não são necessariamente máquinas distintas. Existe um *framework* chamado Struts que diminui o esforço de desenvolver uma aplicação Web segundo o padrão MVC. *Framework* é uma coleção de interfaces e classes para auxiliar o desenvolvimento e manutenção de aplicações.

‘

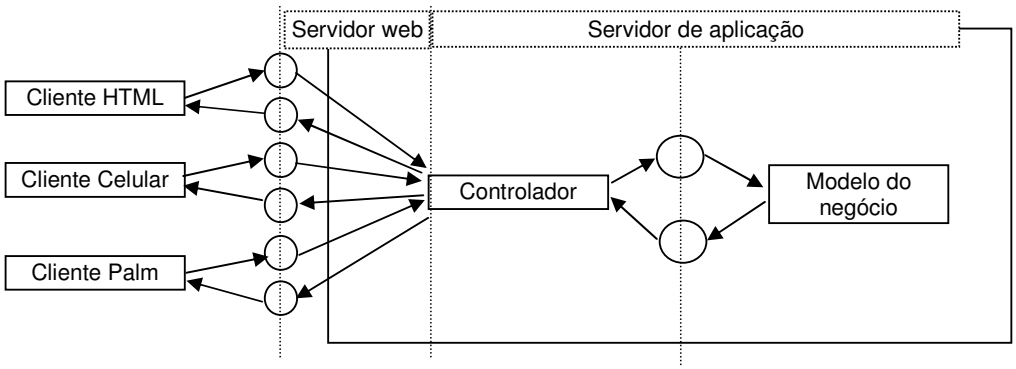


Figura 26. Padrão MVC numa aplicação WEB.

3 PADRÃO OBSERVADOR

O objetivo do padrão observador (Gamma e co-autores, 1995, pg. 294) é reduzir o acoplamento entre classes. Podemos utilizar este padrão em conjunto com o MVC para desacoplar as classes da camada de visão das do modelo. Este é o desacoplamento mais importante, pois separar as classes de controle das de visão nem sempre é uma tarefa evidente.

Suponha que os dados do modelo devem ser vistos de várias maneiras (por meio de diferentes interfaces), mas de maneira sincronizada, ou seja, cada mudança nos dados do modelo deve ser refletida igualmente em todas as interfaces. Neste caso o padrão observador é útil, pois as classes do modelo não necessitam saber quantas e quais classes da camada de visão dependem delas.

Segundo (Gamma et al., 1995), o padrão observador é útil nas situações seguintes:

- ◇ Quando uma modificação em um objeto implica modificações em outros e não se sabe quantos objetos precisam ser modificados.
- ◇ Quando um objeto deve ser capaz de notificar outros objetos, mas sem pressupostos sobre os objetos a serem notificados.

4 CLASSES DE ANÁLISE

Em um diagrama de classes são definidas as classes de objetos, suas operações e atributos e as relações entre classes. A dificuldade é que não há método ou receita para escolher as classes de um sistema. É uma tarefa que depende em grande parte da experiência do desenvolvedor.

Nas fases iniciais do projeto, as classes são chamadas de **classes candidatas ou de análise**, pois há grande probabilidade que mudem ao longo do projeto. Com base no padrão de projeto MVC, uma primeira aproximação das classes necessárias ao sistema pode ser feita. Antes de explicar como fazê-la, apresenta-se a notação de classes e alguns estereótipos de classes.

4.1 Notação UML para Classes

Uma classe é representada por um retângulo dividido em três compartimentos como ilustra a figura 27. Os compartimentos de atributos e métodos são opcionais (figura 28). Um estereótipo define um tipo para a classe.

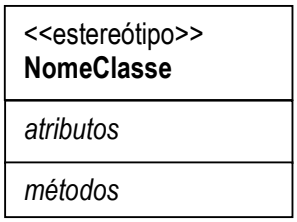


Figura 27: notação para classe.

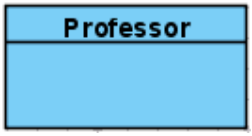


Figura 28: notação de classe sem atributos e métodos.

4.1.1 Atributos

A sintaxe para declaração de um atributo em UML segue o formato:

[<*visibilidade*>]<*nome*>[<*multiplicidade*>]:[<*tipo*>][=<*valor* inicial>]

- ◇ <*visibilidade*>:
  - + público, todas as classes têm acesso;
  - **-** privada, somente métodos definidos na classe podem vê-lo;
  - # protegido, métodos definidos na classe e nas classes derivadas podem vê-lo;
  - ~ pacote, visível dentro das classes de um pacote.
- ◇ <*nome*>: nome do atributo
- ◇ <*multiplicidade*>: por exemplo, valores[5] ou matriz[4, 6]
- ◇ <*tipo*>: pode-se utilizar os tipos da linguagem de implementação. Por exemplo, char, float ou int.
- ◇ <*valor inicial*>: valor inicial para o atributo que respeite o tipo de dado.

Exemplos
- nome: String
- sexo: char='f'
+ código: int=20

4.1.2 Métodos

Sintaxe para declaração de um método em UML:

[<*visibilidade*>]<*nome*>(<*lista argumentos*>):[<*retorno*>]

- ◇ <*visibilidade*>:
  - + público, todas as classes têm acesso;
  - **-** privada, somente métodos definidos na classe podem vê-lo;
  - # protegido, métodos definidos na classe e nas classes derivadas podem vê-lo;
  - ~ pacote, visível dentro das classes de um pacote.
- ◇ <*nome*>: nome do método
- ◇ <*lista de argumentos*>: **(**<nome_argumento>**:**<tipo>**, ..., **<nome_argumento>**:**<tipo>**)**. Por exemplo, (nome:String, idade: int)
- ◇ <*retorno*>: tipo do dado retornado. Pode-se utilizar os tipos da linguagem de implementação. Por exemplo, char, float ou int.

Exemplos
- calcularIdadeEmMeses(Data dataNasc): int
+moverPara(x:int, y:int):void

4.1.3 Categorias de responsabilidades: estereótipos

De maneira geral, um estereótipo deixa clara a função de um componente em um diagrama. É possível definir seus próprios estereótipos o que permite estender a UML. No diagrama de classes há três estereótipos bastante utilizados que designam a responsabilidade das classes:

- ◇ <<*entidade*>> ou <<*entity*>>
- ◇ <<*controle*>> ou <<*control*>>
- ◇ <<*fronteira*>> ou <<*boundary*>>

Estes estereótipos são oriundo do trabalho de Ivar Jacobson (1992) sobre Análise de Robustez. Basicamente, a análise de robustez permite determinar preliminarmente as classes de análise baseando-se nas descrições dos casos de uso.

Classes entidade

É utilizado em classes que armazenam informações do domínio a longo-prazo. Objetos destas classes são normalmente persistentes, mas não é uma regra geral. Frequentemente estas classes se encontram na camada do modelo, por isso não dependem (ou ao menos não deveriam depender) do contexto onde o sistema está inserido. Pode tanto refletir uma entidade do mundo real ou uma entidade necessária ao funcionamento interno do sistema. Classes com este estereótipo podem ser representadas pelo ícone alternativo[2] da figura 29.

Figura 29: ícone alternativo para classes <<entidade>>.

Classes de fronteira

É utilizado em classes que realizam a interação entre sistema e ator (humano ou não). São classes que dependem do contexto onde o sistema está inserido, dado que o contexto é dado justamente pelas interações com o mundo externo. Classes de fronteira representam, por exemplo, protocolos de interação com outros sistemas, interfaces com usuários, interação entre sistema e dispositivos. Classes com este estereótipo podem ser representadas pelo ícone da figura 30.
Segundo (Jacobson e co-autores, 2002), se uma classe <<fronteira>> já foi designada para um ator humano, o desenvolvedor deve tentar reutilizá-la para minimizar o número de janelas que o ator utiliza.

Figura 30: ícone alternativo para classes <<fronteira>>.

Estereótipo de controle

É utilizado em classes que encapsulam o controle/lógica de um caso de uso, ou seja, aquelas que comandam a execução de um caso de uso fazendo a ligação das classes de fronteira com as de entidade. Normalmente são dependentes da aplicação. Classes com este estereótipo podem ser representadas pelo ícone da figura 31.

Em alguns casos, a lógica do caso de uso pode ser muito complexa e exigir mais de uma classe de controle. Em outros, a lógica é comanda pelo ator e neste caso, a classe de controle e de fronteira podem ser unificadas como fronteira.

Figura 31: ícone alternativo para classes <<controle>>.

[2] Robustness Analysis Icon

4.2 Linhas Mestras

Para identificar as classes candidatas seguindo o padrão de projeto MVC, executar os passos seguintes:

- ◇ Definir uma classe de fronteira para cada par ator-caso de uso
- ◇ Definir uma classe de controle para cada caso de uso
- ◇ Definir classes de entidade: procurar substantivos nos casos de uso e pontos onde há um conjunto de dados que possuem unidade (objeto).

Estas linhas são extremamente gerais e, portanto, sujeitas a adaptações em função do problema em mãos. Por exemplo, se a fronteira com um ator for extremamente simples pode-se conjugar controle e fronteira. Se o controle para todos os casos de uso for extremamente simples, pode-se colocá-los numa só classe de controle.

5 EXEMPLO

O levantamento de classes de análise é mostrado por meio de um exemplo didático cujo enunciado é o seguinte:

> Um meteorologista necessita de um programa simples que faça conversões de temperaturas em graus Celsius para Fahrenheit. Ele deseja armazenar as 10 últimas conversões realizadas, denominadas no seu todo "histórico", e visualizá-las constantemente numa janela independente daquela destinada a fazer a conversão. Se por acaso o usuário fechar a janela do histórico, ele deve ter algum modo de reabri-la. Uma conversão é formada pela data, hora, valor em Celsius e em Fahrenheit.

A partir do enunciado, elaborou-se o diagrama de casos de uso da figura 53. Notar que não preocupação com a ordem de execução dos casos de uso e com o fluxo de dados (o fato de *consultar histórico* utilizar os mesmos dados manipulados por *atualizar histórico*).

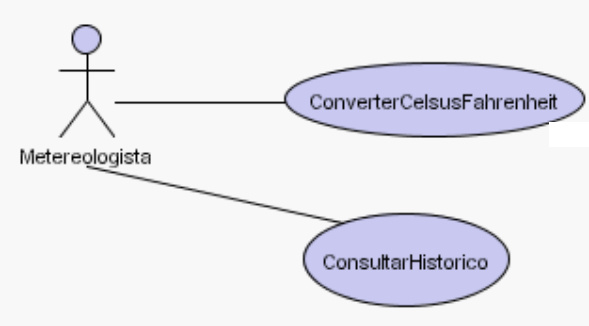


Figura 32: Diagrama de casos de uso para conversor de graus Celsius para Fahrenheit.

TABELA 5: DESCRIÇÃO DO CASO DE USO CONVERTER.

Nome do caso de uso	**Converter Celsius Fahrenheit**
Descrição	Permite ao meteorologista realizar várias conversões de Celsius para Fahrenheit e as armazena em um histórico.
Ator Envolvido	Meteorologista
Pré-condições	Sessão iniciada
Pós-condições	Conversão realizada
Fluxo Básico	
Ator	**Sistema**
Solicitar conversão de uma temperatura em graus Celsius	
	{Obter valor}

	Solicita o valor a ser convertido
Fornece valor a ser convertido	
	Obtém o valor ***{Validade valor entrada}***
	Realizar a conversão
	Executar subfluxo Atualizar Histórico
	Mostrar temperatura em Fahrenheit
	Se o usuário deseja continuar, voltar ao ponto {Obter valor}; se não, o caso de uso termina.
Fluxos alternativos e subfluxos	
A1: em ***{Validade valor entrada}***	Se temperatura não for numérica voltar ao ponto ***{Obter valor}***
S1: Atualizar Histórico	Armazena uma conversão de Celsius para Fahrenheit incluindo o valor em Celsius, o equivalente em Fahrenheit, data e hora da conversão.

TABELA 6: DESCRIÇÃO DO CASO DE USO CONSULTAR HISTÓRICO

Nome do caso de uso	**Consultar Histórico**
Descrição	Permite ao usuário visualizar as dez últimas conversões realizadas.
Ator Envolvido	Meteorologista
Pré-condições	Sessão iniciada
Pós-condições	Histórico mostrado ao meteorologista.
Fluxo básico	
Ator	**Sistema**
Solicitar visualização	
	{nenhuma conversão realizada} Sistema busca as últimas dez conversões realizadas ou as existentes
	Mostra o histórico de conversão
	{fim} O caso de uso termina.
Fluxos alternativos	
A1: em **{nenhuma conversão realizada}**	Se não houver conversões no histórico mostrar mensagem "nenhuma conversão realizada até o momento" e ir para o ponto **{fim}**

Após a elaboração do diagrama de casos de uso, fez-se uma primeira tentativa de identificar classes de análise seguindo as linhas mestras. Na figura 33, as linhas mestras foram seguidas à risca.

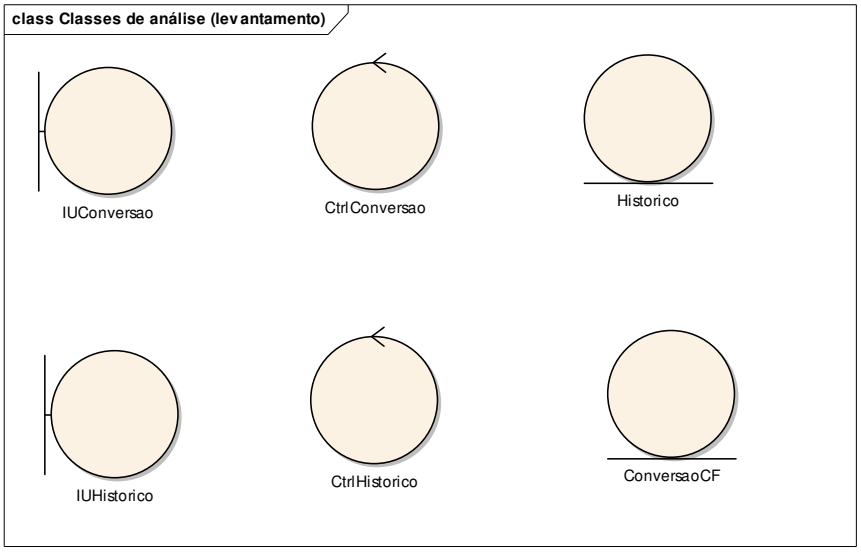


Figura 33: Levantamento das classes de análise.

6 EXERCÍCIOS

1. Considere um caso de uso chamado efetuar operações aritméticas para uma calculadora. Analise as responsabilidades e o comportamento de uma classe de controle para este caso de uso frente a duas implementações para a classe <<entidade>> calculadora cujo comportamento é descrito abaixo:
   a. capaz de efetuar operações binárias, por exemplo, somar(2, 3), subtrair (5, 4), etc.
   b. capaz de avaliar sintaticamente e executar expressões aritméticas, por exemplo, 2 + 3/5.
2. Qual o auxílio trazido pelo padrão observador ao modelo MVC?
3. Fazer o levantamento das classes candidatas para os exercícios do capítulo anterior (pg. 31):
   a. Biblioteca
   b. Jogo da forca e da velha
   c. Escritório de advocacia

V ESTUDO DA INTERAÇÃO ENTRE OBJETOS

O projeto de um sistema pode ser visto como sucessivos refinamentos de divisão de responsabilidades. Nos casos de uso, são identificadas as funcionalidades do sistema separando-as do que é externo ao mesmo. Em seguida, são levantadas as classes de análise, o que representa uma primeira tentativa de identificar quem dentro do sistema fará o quê para que os casos de uso se concretizem. Os diagramas de interação constituem um passo importante nesta divisão, pois detalham como os objetos das classes de análise e, por consequência, quais operações cada um deve realizar. Se um caso de uso possui vários fluxos, pode ser útil criar um diagrama de interação para cada cenário. Os diagramas de interação mais utilizados são o de sequência e o de comunicação (ex-colaboração).

Diagramas de interação são usados tanto na análise quanto no projeto. Na análise as comunicações entre objetos são mais abstratas, não importando muito os argumentos e valores retornados. No projeto, o detalhamento é maior

1 DIAGRAMA DE SEQUÊNCIA

Relembra-se que cada caso de uso representa uma funcionalidade do sistema vista da perspectiva de um ator. Um caso de uso pode apresentar diversas possibilidades de execução visto que há fluxos **primários** (fluxo normal de eventos) e **alternativos** (o que fazer se). Estes fluxos são documentados textualmente ou na forma de uma tabela ator x sistema.

Cenários são utilizados para identificar como uma sociedade de objetos interage para realizar um caso de uso. Cenários **documentam as responsabilidades**, ou seja, como as responsabilidades especificadas nos casos de uso são divididas entre os objetos do sistema. Portanto, um **cenário é uma instância** de um caso de uso: um caminho entre os muitos possíveis. Cenários podem ser representados por **diagramas de seqüência** e pelos **diagramas de colaboração**.

> Um **diagrama de seqüência** representa os atores e objetos envolvidos num cenário e a seqüência de troca de mensagens ao longo do tempo para realizar o cenário.

Um diagrama de seqüência permite identificar os métodos e atributos de cada classe assim como as responsabilidades de cada classe na realização de um caso de uso. Os elementos básicos de um diagrama se seqüência são

- ◇ Atores
- ◇ Objetos
- ◇ Linha do tempo (uma para cada objeto e ator)
- ◇ Comunicação entre ator e objeto ou entre objetos
- ◇ Interpretação das mensagens: por exemplo, evento do sistema operacional ou de uma interface, envio de mensagem ou chamada de método

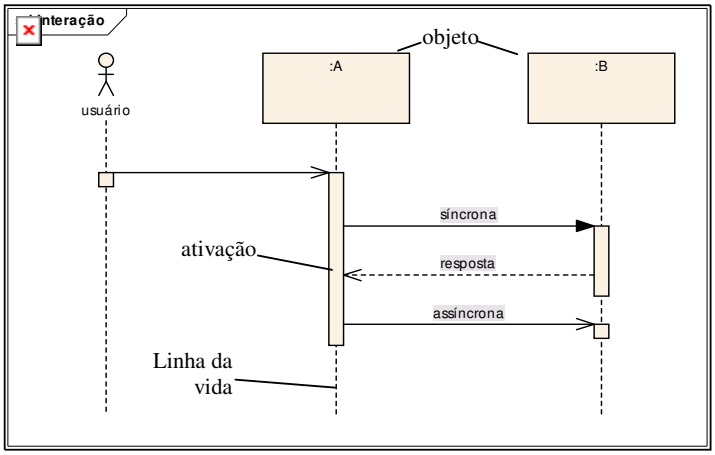


Figura 34: exemplo de diagrama de seqüência.

1.1 Tipos de mensagem

Há vários tipos de mensagem que podem ser utilizados num diagrama se seqüência, sendo os mais comuns os seguintes (figura 34):

- ◇ **Simples**: quando o tipo de mensagem é irrelevante ou ainda não foi definido.
- ◇ **Síncrona**: quando enviada, o emissor fica **bloqueado** aguardando a resposta.
- ◇ **Assíncrona:** o emissor **não bloqueia** até que o receptor enviar resposta para continuar seu processamento.
- ◇ **Retorno ou resposta**: resposta à mensagem síncrona; pode ser omitida

Uma mensagem é definida sintaticamente por:

expressão-seqüência recorrência: v := mensagem

Expressão-seqüência: é um número seqüencial de envio das mensagens. Por exemplo,

1: msg1
2: msg2
representando que a mensagem msg1 precede a msg2.

Pode-se utilizar níveis:

1.1: msg1
1.2: msg2
representando que msg1 e msg2 foram enviadas após recebimento de uma mensagem 1.

Para representar mensagens concorrentes, pode-se utilizar letras:

1.1a: msg1
1.1b: msg2
significando que msg1 e msg2 são enviadas concorrentemente.

Recorrência: indica um envio condicional ou uma repetição. Zero ou mais mensagens são enviadas dependendo da condição envolvida. As opções são:

[cláusula-iteração***]** *ou*

[*guarda***]**

Não há sintaxe rígida definida pela UML, a cláusula de interação e a condição de guarda podem ser especificadas em pseudocódigo ou na linguagem alvo. Exemplos:

[x > y] *msg*: mensagem *msg* é enviada somente se x > y

*[i:=1..10] *msg*: mensagem *msg* é enviada 10 vezes.

*[x > 0] *msg*: enquanto x for maior que zero enviar mensagem *msg.*

Receber valor de retorno: é possível receber um valor de retorno e atribuí-lo a uma variável.

x:=obter()

1.2 Linha da Vida

São linhas verticais que representam o tempo de vida de um objeto. Um **X** na linha denota o fim da existência do objeto. Objetos no topo do diagrama são considerados criados desde o início.

1.3 Ativação

Uma ativação (um retângulo sobre a linha da vida) representa o período durante o qual um objeto está em execução **diretamente** ou **indiretamente**. Direta, se estiver executando um método ou indireta, se chamou um método e está bloqueado aguardando o retorno. A ativação mostra a d**uração** das operações nos objetos e o **fluxo de controle** entre o objeto invocador e o invocado.

1.4 Alt

Equivale ao if-else.

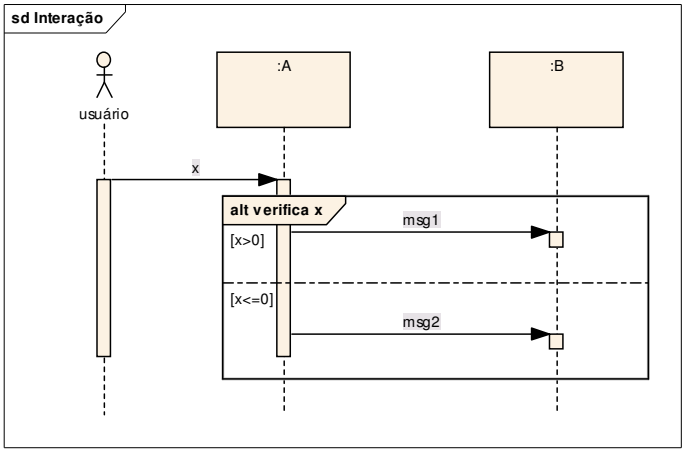


Figura 35: exemplo de alt no diagrama de seqüência.

O mesmo comportamento pode ser representado através de uma condição de guarda *in-line* como ilustra a figura 36.

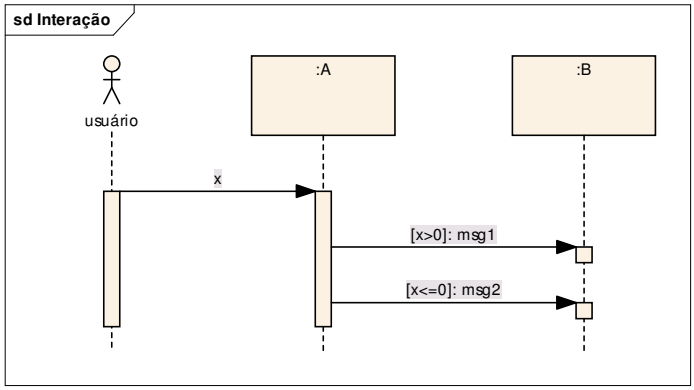


Figura 36: exemplo de condição de guarda no diagrama de seqüência.

1.5 Opt

Opcional: equivale ao if (sem else).

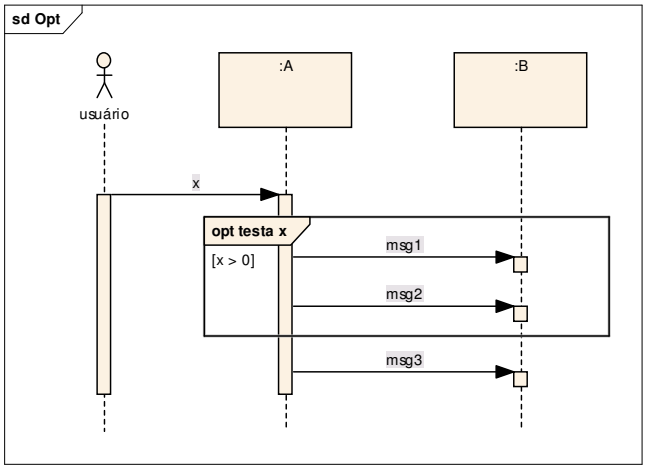


Figura 37: exemplo de opt.

1.6 Loop

Permite representar laços.

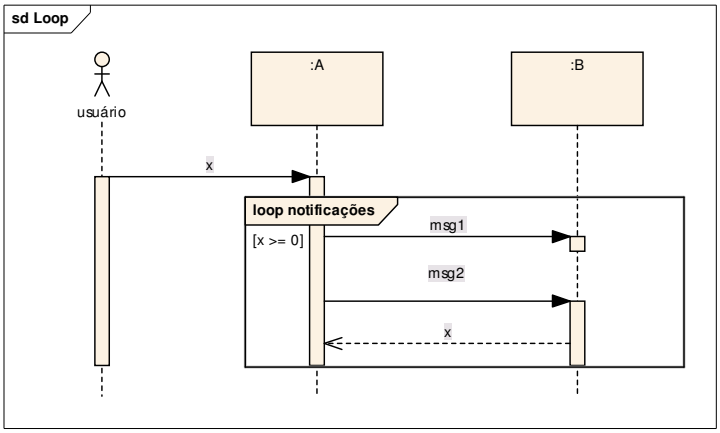


Figura 38: exemplo de loop em diagrama de seqüência.

Repetições também podem ser representados *inline*. Na Figura 39 está representado que a mensagem *msg1* é enviada várias vezes (*) enquanto *x for maior ou igual a zero*. Ao fazer a chamada *msg1*, o objeto da classe A recebe como resposta um inteiro e o armazena na variável *x*.

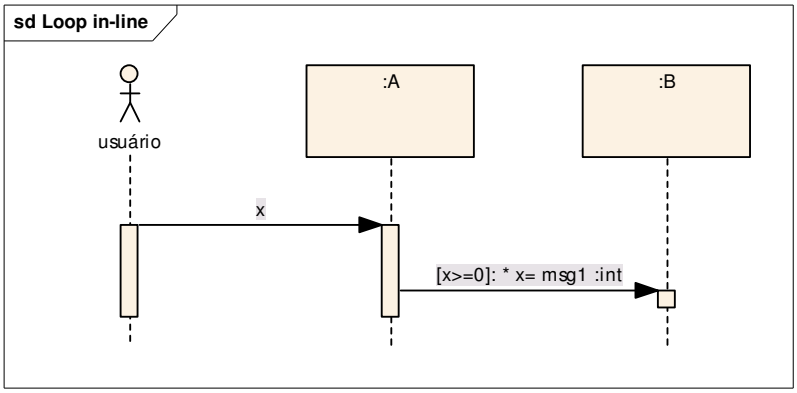


Figura 39: exemplo de loop *inline*.

A sintaxe especificada no UML Superstructure (7/2/2005) para loop é definida como:

*'**loop**['(' <minint> [',' <maxint>] ')']*
<*minint*> ::= natural não negativo
<*maxint*> ::= natural não-negativo maior ou igual <*minint*> | ' *' significa infinito.
Se somente <*minint*> for definido, significa que <*minint*> = <*maxint*> = <*inteiro*>.
Se somente **loop** for definido, então representa um laço com limite superior infinito e limite inferior igual a zero.

Exemplo:
loop[(0, 10)] define um laço com 11 repetições

1.7 Ref

Permite fazer referências a outros diagramas. Observar no diagrama referenciado que o resultado retornado foi representado como um objeto. Para fazer referência ao diagrama em questão, basta colocar um fragmento ref *inline*. O fragmento ref é substituído por uma réplica do diagrama referenciado onde os argumentos substituem os parâmetros.

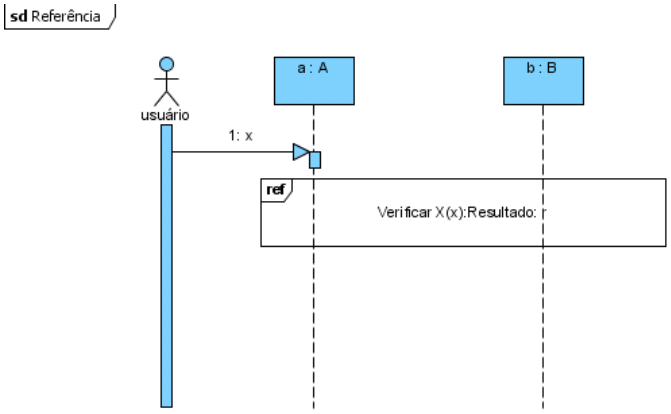


Figura 40: exemplo de referência a outro diagrama (ver figura 41)

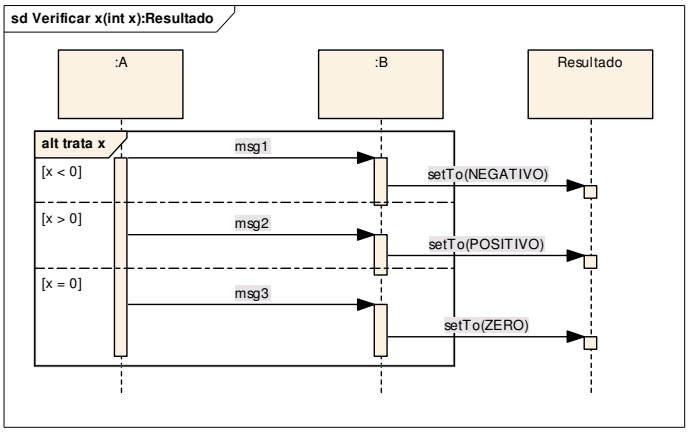


Figura 41: exemplo de diagrama referenciado.

1.8 Criar e destruir

É possível mostrar a criação e a destruição de objetos num diagrama de seqüência. Objetos que são criados e destruídos são chamados de transientes.

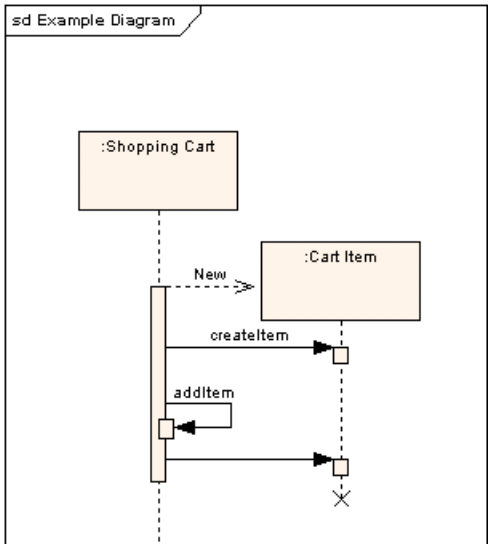


Figura 42: exemplo de objeto transiente (*CartItem*).

1.9 Linhas Mestras

Um diagrama de seqüência deve ter certa distância do código fonte, não é preciso representar todos os detalhes do cenário. Algumas dicas:

- ◇ mantê-lo simples;
- ◇ condições de guarda: se forem poucas devem ser colocadas em um só diagrama, caso contrário fazer um novo diagrama só para mostrá-las;
- ◇ os diagramas podem ser ligados entre si (ref).

Em algumas ferramentas é possível **gerar automaticamente** a primeira versão do **diagrama de seqüência**. Por exemplo, no Rational Rose, se o fluxo de evento foi definido, basta clicar com o botão da direita em cima do caso de uso e escolher *generate sequence diagram.*

2 DIAGRAMA DE COMUNICAÇÃO

Diagrama de comunicação (antes da versão 2.0 era chamado colaboração) é outra forma de representar um cenário: contém as mesmas informações que o diagrama de seqüência, mas não considera a dimensão temporal. Alguns autores recomendam utilizá-lo para analisar a colaboração das classes de realização de um caso de uso (Jacobson e co-autores, 1999), pois facilita a identificação das classes que colaboram de forma mais próxima e, por conseqüência, a identificação de pacotes de análise.

Visual Paradigm: Para gerar automaticamente este diagrama a partir do de seqüência clicar com o botão da direita no pano de fundo do diagrama de seqüência e escolher to *collaboration diagram*.

No diagrama de comunicação, a ordem de envio das mensagens é dada por números seqüenciais (é possível utilizá-los no diagrama de seqüência também). A figura 43 mostra um exemplo de diagrama de colaboração. Observar a restrição ***{transient}*** junto ao objeto *Transaction* indicando que ele é criado e destruído na interação.

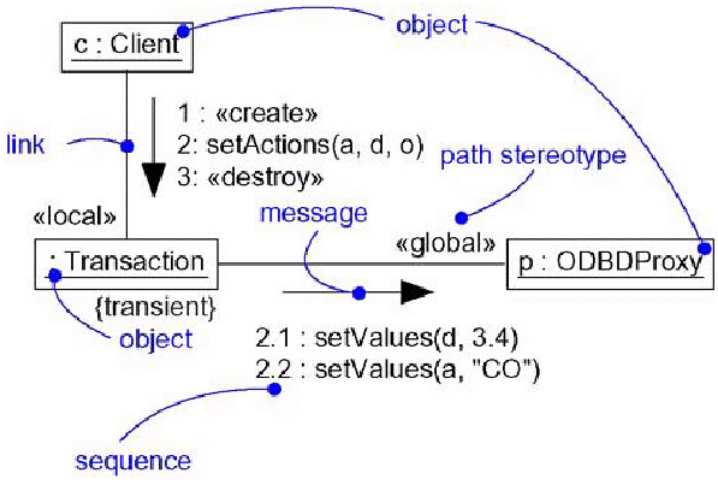


Figura 43: diagrama de comunicação.

A figura 44 mostra o diagrama de seqüência equivalente ao de comunicação da figura 43.

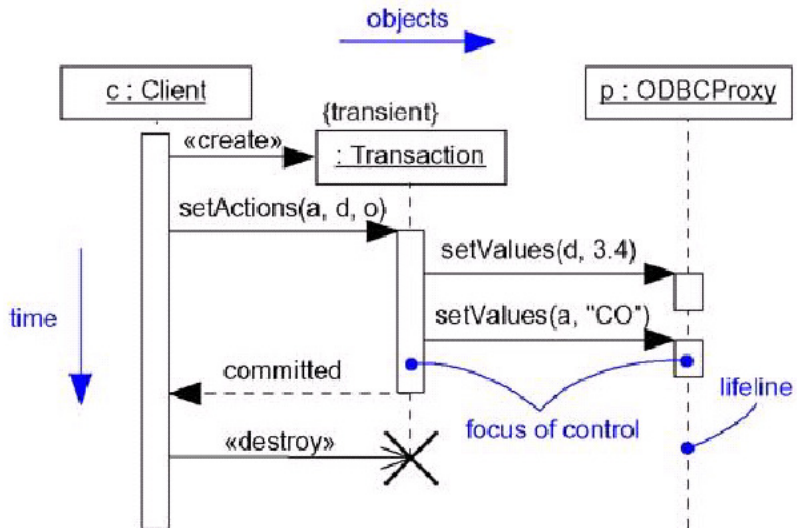


Figura 44: diagrama de seqüência equivalente.

3 EXEMPLO

Retoma-se o exemplo do conversor de graus Celsius para Fahrenheit. O cenário fluxo básico + subfluxo atualizar histórico é mostrado na figura 45. Observar que o diagrama é bastante informal, sem preocupação com argumentos de métodos, criação e destruição de objetos, armazenamento em banco de dados ou sistema de arquivos e se há lugar ou não no histórico para guardar uma nova conversão.

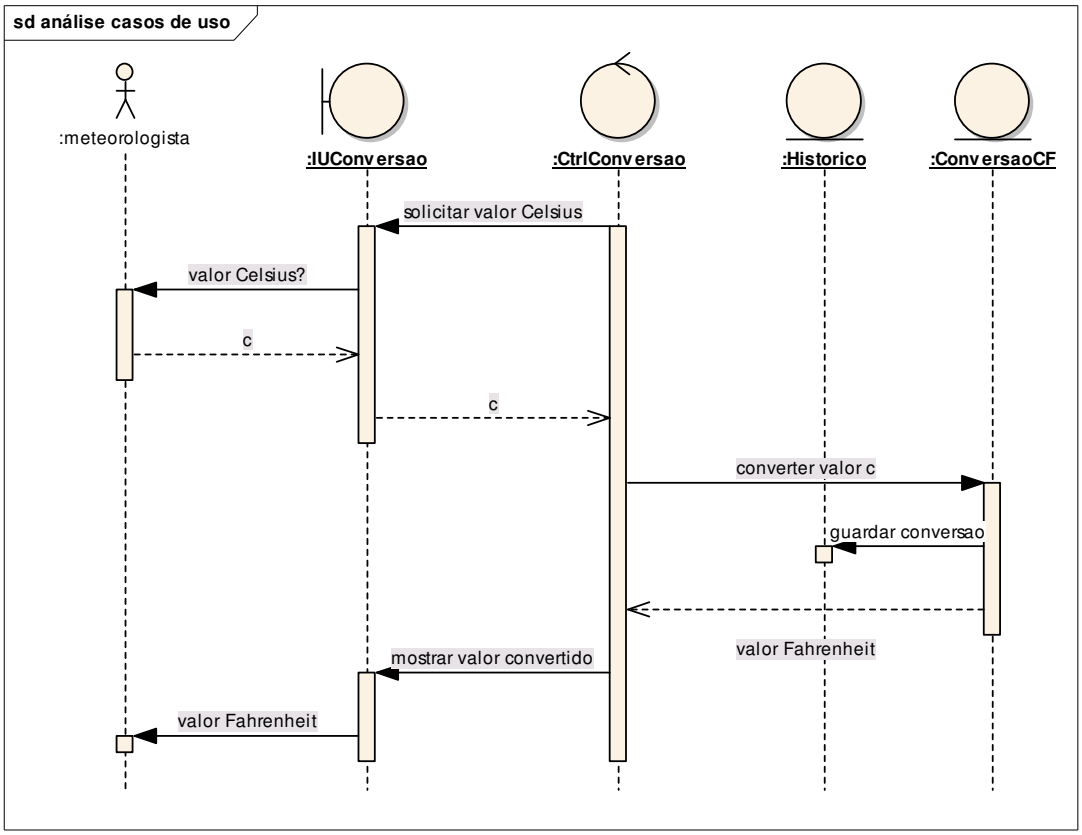


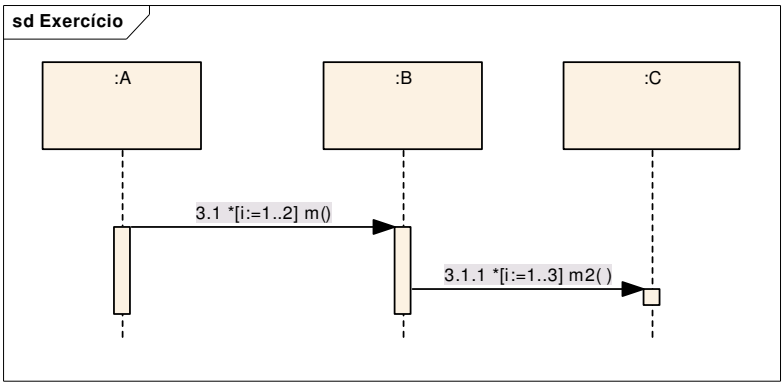
Figura 45: diagrama de seqüência para o caso realizar conversão C > F.

4 PACOTES

A estrutura de uma aplicação é dada pelas classes e suas relações. Classes podem ser agrupadas em **pacotes de análise** para facilitar o manuseio do modelo de análise (lidar com partes menores) ou à medida que o modelo de análise aumenta e precisa ser decomposto. Além de identificar os pacotes de análise, é preciso definir os **serviços** que cada um deles fornece.

5 EXERCÍCIOS

1. No diagrama de seqüência abaixo qual a ordem das mensagens enviadas (m1 e m2):



2. Desenhe o diagrama de comunicação equivalente ao diagrama de seqüência anterior.
3. Faça um diagrama de seqüência para representar um cliente que efetua uma retirada R$50,00 em um caixa automático. A retirada deve ser debitada da conta corrente do cliente.
4. Complete o exercício anterior para permitir saques somente quando há saldo na conta corrente e se o valor do saque for inferior a R$1.000,00.
5. Um casal tem três filhos. Cada filho tem seu quarto. Para facilitar as tarefas matinais, o casal dispõe de um relógio no quarto capaz de enviar sinais de rádio aos relógios dos filhos, mas não de receber. Ao anoitecer, o casal *seta* no relógio deles o horário para despertar, idêntico para todos os filhos. Ao amanhecer, na hora marcada, o relógio do casal envia o sinal de despertar para cada um dos relógios dos filhos. Desenhe o diagrama de seqüência.
6. Suponha o caso de uso apresentado em seguida resultante da fase de análise de requisitos. Na fase de análise (realização dos casos de uso) sua descrição pode ser detalhada sem se preocupar profundamente com a implementação. Descreva a realização do mesmo fazendo os itens abaixo:
   a. Detalhe a descrição do caso de uso considerando que ele será voltado a funcionar na web. Os clientes devem ter meios de realizar reservas on-line:
      i) Podem escolher a categoria de veículo ao invés de visualizar todos (passo 5).
      ii) Podem participar de um programa de fidelidade e, neste caso, informações de cadastro (ou perfil) são utilizadas para pré-popular os formulários.
      iii) Ao confirmar a reserva, o sistema deve mostrar os dados da mesma.
      iv) Se o usuário informou um endereço de e-mail válido, o sistema envia-lhe a confirmação por email.
   b. Identifique as classes de análise candidatas.
   c. Filtre as classes de análise reduzindo (possivelmente) o número de classes candidatas
   d. Identifique e descreva as responsabilidades de cada classe e possíveis relacionamentos entre elas
   e. Estude a interação entre os objetos das classes
   f. Identifique possíveis atributos para as classes

Sistema para locação de carros on-line (web)

NOME: Reservar veículo

DESCRIÇÃO: Este caso descreve como um cliente usa o sistema para reservar um veículo.

PRÉ-CONDIÇÕES: o cliente está conectado ao sistema.

PÓS-CONDIÇÕES: o cliente realiza uma reserva ou não.

REQUISITOS NÃO-FUNCIONAIS: serviço acessível pela WEB.

FLUXO BÁSICO DE EVENTOS

1. O caso de uso inicia quando o ator *cliente* escolhe a opção de reservar veículo.
2. O sistema solicita ao usuário os locais, datas e horários de retirada e de devolução do veículo.
3. O cliente informa os locais, datas e horários de retirada e de devolução do veículo.
4. O sistema pergunta ao usuário qual o tipo de veículo desejado.
5. O sistema apresenta todas as opções de tipos de veículo disponíveis no local de retirada para a data e horários escolhidos.
6. O cliente escolhe um veículo para locar.
7. O sistema solicita informações de identificação ao cliente (nome completo, telefone, e-mail, bandeira do cartão de crédito, data de expiração, etc.).
8. O cliente fornece as informações de identificação solicitadas.
9. O sistema apresenta informações sobre seguros e proteções e pergunta ao cliente para aceitar ou rejeitar cada oferta.
10. O cliente informa suas escolhas.
11. O sistema solicita confirmação da reserva.
12. O cliente aceita a reserva.

VI RELAÇÕES ENTRE CLASSES DE ANÁLISE

O diagrama de classes é um dos principais diagramas da UML, representa a estrutura do sistema (elementos que foram selecionados para fazer parte do sistema). A partir dele, por exemplo, o esqueleto do código fonte pode ser gerado automaticamente. Neste capítulo são apresentadas as relações mais usuais para a fase de análise. Posteriormente, apresentam-se relações mais próximas da implementação e, portanto, mais adequadas à fase de projeto.

O comportamento requerido do sistema é alcançado pela colaboração entre objetos. O diagrama de classes fornece uma **representação estática** da colaboração por meio de relacionamentos. Os relacionamentos são utilizados no diagrama de classes que é refinado incrementalmente (modelo do domínio, análise e, posteriormente, no projeto).

São apresentadas as seguintes relações:

- ◇ associação (mais comum),
- ◇ agregação (um tipo de associação),
- ◇ generalização/especialização

1 ASSOCIAÇÃO

É uma relação entre duas classes significando que os objetos destas classes possuem uma ligação. Por exemplo, a figura 46 representa *um professor* ***leciona*** *uma disciplina*.

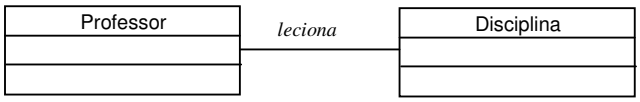


Figura 46: Exemplo de associação.

É possível designar que uma quantidade objetos de uma classe se relacionam com uma quantidade de objetos de outra classe por meio da notação de multiplicidade (tabela 7). As multiplicidades são colocadas nos extremos das linhas que representam as relações.

TABELA 7: MULTIPLICIDADE DE RELAÇÕES

Repr	Significado
1	Exatamente uma
0.. *	Zero ou mais
*	Zero ou mais
1.. *	Uma ou mais
0..1	Zero ou uma
5..8	Intervalo específico (5, 6, 7, or 8)
4..7,9	Combinação (4, 5, 6, 7, or 9)

Por exemplo, a figura 53 é interpretada como *um professor leciona uma ou mais disciplinas*, sendo ilimitado o número máximo de disciplinas. Observar que no lado do Professor a multiplicidade é igual a 1, então a participação de objetos Professor no relacionamento *leciona* é obrigatória, ou seja, um professor só pode existir se estiver associado a uma disciplina.

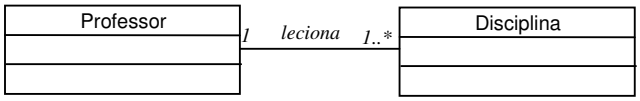


Figura 47: Exemplo de multiplicidade em associação.

Navegabilidade

As associações podem opcionalmente **direcionadas**. No exemplo abaixo, a partir de um professor pode-se chegar a uma disciplina, mas o caminho inverso não é possível o que é indicado pelo X sobre a associação.

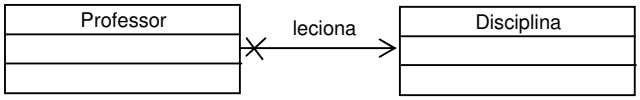


Figura 48: associação unidirecional.

Para representar uma associação bidirecional, navegável nos dois sentidos, utiliza-se uma flecha bidirecional. Para facilitar a leitura, pode-se indicar a direção da mesma (triângulo ao lado do nome da associação).

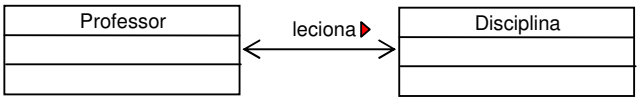


Figura 49: associação bidirecional.

Se não há definição sobre a navegabilidade pode-se deixá-la não-especificada. Neste caso, não se deve utilizar flechas. Por exemplo, a figura 50 representa uma associação 1:1 de navegabilidade não-especificada.

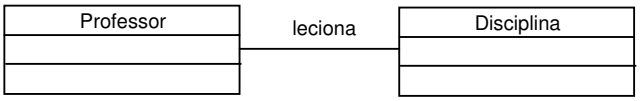


Figura 50: associação não-especificada.

Papéis

Podemos associar **papéis** às associações para clarificar as responsabilidades dos objetos participantes. A figura 51 ilustra uma associação entre Empresa e pessoa. Para evitar um nome ambíguo de associação pela utilização de papéis. Por exemplo, o nome trabalha pode ser interpretado como:

- pessoa *trabalha* para Empresa
- empresa *trabalha (presta um serviço)* para Pessoa;

O nome emprega pode ser interpretado como:

- empresa *emprega* Pessoa;
- pessoa *emprega (contrata)* Empresa

Assim, para representar os dois primeiros itens pode-se utilizar papéis, a Empresa desempenha o papel de *empregador* de Pessoa e Pessoa, *empregado*.

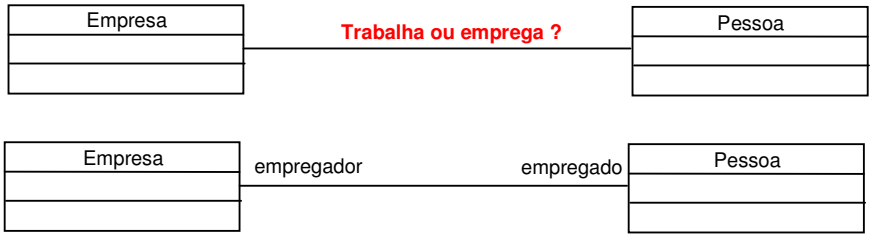


Figura 51: Papéis nas relações.

1.1 Associação reflexiva

Os objetos de uma classe podem se relacionar com objetos da mesma classe. Por exemplo, uma relação do tipo pai-filho ou chefe-empregado cada objeto desempenha um papel diferente. Neste tipo de associação é freqüente a utilização de papéis ao invés de nome de associação para evitar ambigüidades na leitura.

Na associação reflexiva da classe Pessoa (figura 52), a interpretação é a seguinte: uma pessoa tem zero ou um pai e uma pessoa pode ser pai de vários filhos. A multiplicidade zero no papel filho permite que mulheres e homens sem filhos não participem desta associação. A multiplicidade zero no papel pai pode parecer estranha, pois todo filho tem um pai, porém, se colocarmos 1 na multiplicidade, todos os pais do mundo teriam que ser representados no sistema.

Figura 52: Exemplos de associações reflexivas.

Na associação reflexiva da classe Empregado (figura 52), a interpretação é a seguinte: um empregado que desempenha o papel de gerente tem zero ou mais subordinados. Pode parecer estranho um gerente sem subordinados, mas se colocássemos 1 ao invés de zero na multiplicidade no lado do papel subordinado, todo empregado teria que ter ao menos um subordinado. Portanto, este zero significa que empregados que não desempenham o papel de gerente não precisam estar associados a subordinados. No sentido contrário, todo empregado está associado a exatamente um gerente. Neste caso, define-se que um gerente é subordinado a ele mesmo no nível mais alto da hierarquia, por isso podemos deixar multiplicidade igual a 1 no lado gerente.

1.2 Classes associativas

Quando uma relação associativa possui atributos próprios pode-se criar uma classe associativa. Estas classes são úteis quando queremos armazenar o histórico de uma associação (relacionamentos que ocorrem e interessam serem salvos). No exemplo abaixo (figura 53) quando existir uma associação entre uma instância de aluno e uma de turma haverá também uma instância de inscrição para armazenar o resultado escolar.

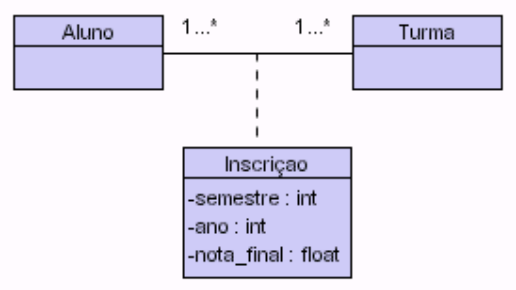


Figura 53: Exemplo de classe associativa.

Algumas características das classes associativas:

- ◇ Classes associativas são comuns em relações de multiplicidade *:*, embora não seja uma regra definitiva.
- ◇ A linha que representa a associação não é nomeada, o nome da classe associativa deve ser suficiente para identificar a relação.
- ◇ Classes associativas podem estar relacionadas a outras classes.

1.3 Relações Ternárias

É possível representar em UML relações entre objetos de três ou mais classes. No exemplo abaixo, está representado o fato de um professor lecionar numa sala para vários alunos (o professor sempre utiliza a mesma sala).

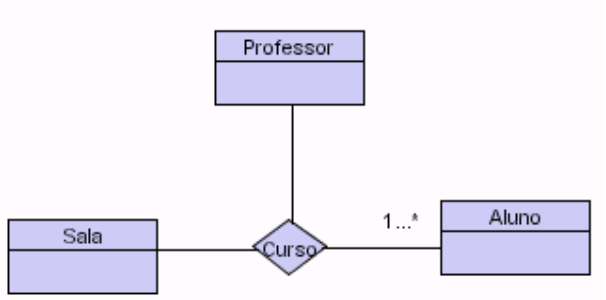


Figura 54: exemplo de relação ternária.

Podemos ter classes associativas também nas associações múltiplas. No exemplo abaixo, o jogador participa de vários times ao longo de uma temporada, para cada participação num time armazena-se os dados de gols marcados, gols levados, vitórias e derrotas do time.

Figura 55: exemplo de relação ternária com classe associativa.

1.4 Levantamento das associações

Normalmente as associações vêem as regras do negócio, dos requisitos funcionais e dos diagramas de interação. Por exemplo, em uma universidade é prática comum que cada professor tenha entre zero e quatro turmas. Em uma biblioteca, alunos não podem emprestar mais de quatro livros ao mesmo tempo.

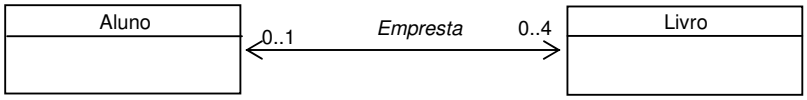


Figura 56: associação capturada a partir de uma regra do negócio.

Ao observar o diagrama de seqüência da conversão de graus Celsius para Fahrenheit (figura 45, pg. 50), é possível identificar as seguintes classes estão de alguma forma associadas:

- IUConversão e CtrlConversão
- CtrlConversão e ConversãoCF
- ConversãoCF e Histórico

O diagrama de classes de análise pode ser enriquecido com estas relações e com outras provenientes do diagrama de seqüência do caso de uso *Visualizar histórico* produzindo o diagrama de classes de análise da figura 57. Observar que as associações que envolvem classes de controle e de fronteira não são nomeados, pois apresentam uma mesma semântica: comunicação.

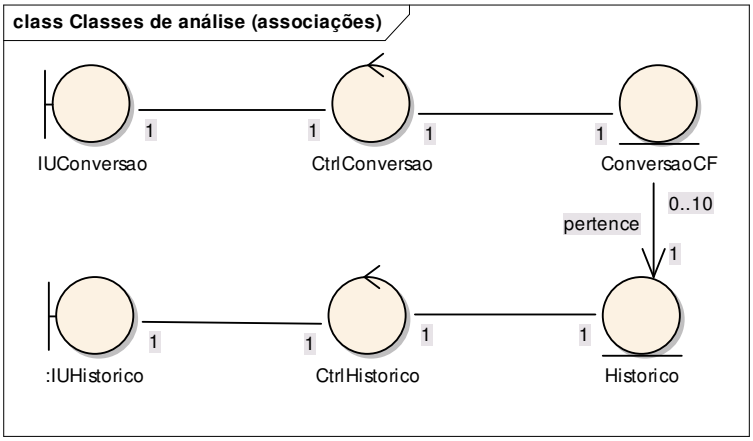


Figura 57: classes de análise do conversor Celsius-Fahrenheit com associações.

2 AGREGAÇÃO

É um caso especial de associação utilizada para representar **relacionamentos de pertinência ("parte de")**. Permite representar o fato que um objeto ou mais objetos de uma classe fazem parte de um objeto de outra classe. Um exemplo típico é uma janela de interface com o usuário composta por diversos botões, campos texto, *scrolls bars*, etc. A agregação representa uma ligação entre o objeto o todo (a janela) e as partes (Botão, ComboBox, ScrollBar). Um comportamento que se aplica ao todo se propaga as partes, por exemplo, ao movimentar a janela, todos seus elementos de deslocam também.

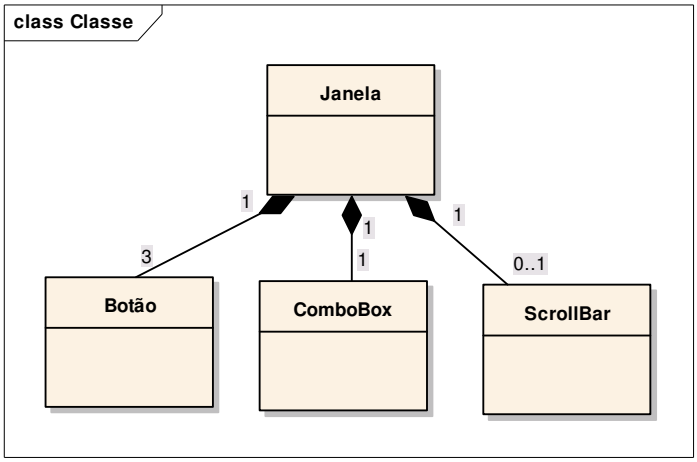


Figura 58: exemplo de agregação.

2.1 Notação

Podem-se incluir nomes, papéis, multiplicidades e navegabilidades, enfim aceita todos os adornos de uma relação de associação.

2.2 Multiplicidade

Frequentemente será 1 do lado da classe *todo* visto que um objeto normalmente é parte de um só objeto. Porém, há diversas exceções como o exemplo da figura 59. Nela representa-se que um time é composto por vários jogadores (inclusive por nenhum) e que um jogador pode participar de vários

times. Observar também a representação da navegabilidade: do todo é possível alcançar todas as partes e de cada uma das partes é possível alcançar o todo.

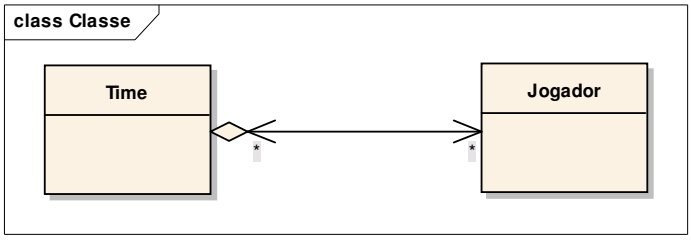


Figura 59: Exemplo de agregação com multiplicidade.

2.3 Tipos de agregações

Nos exemplos anteriores, foram utilizados dois tipos de agregações, de composição e de associação, representadas respectivamente por losangos preenchidos e vazios.

Composição (*composite aggregation*)

É uma agregação de fato, o todo é composto pelas partes. Existe uma relação forte entre o todo e as partes, pois quando o todo é destruído as partes também o serão, ou seja, a eliminação do todo se propaga para as partes. De outra forma, o todo e as partes têm tempos de vida semelhantes.

Associação

É uma forma mais branda de agregação, embora exista uma relação de composição os todos os comportamentos são propagados para as partes. Por exemplo, uma equipe de futebol composta por vários jogadores pode deixar de existir, mas os jogadores continuam. A figura 60 mostra um exemplo similar, onde uma turma é composta por alunos (de 0 a 45). A turma pode ser destruída sem afetar a existência dos objetos alunos dentro do sistema.

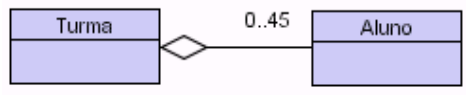


Figura 60: Exemplo de agregação por associação.

No exemplo da figura 61, se destruirmos o cinema a bilheteria será destruída. A coleção de filmes não terá o mesmo fim.

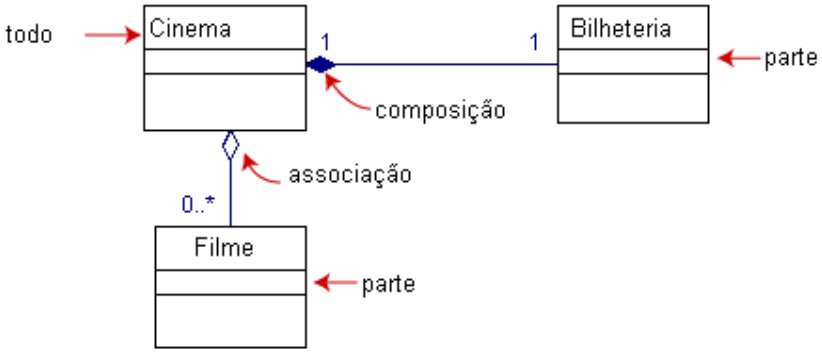


Figura 61: Exemplo de agregações por composição e por associação.

2.4 Levantamento

A identificação de agregações pode ser feita a partir de:

- ◇ **decomposição:** quando uma classe torna-se muita complexa ou extensa podemos dividi-la em várias classes que comunicam entre si. O risco é perder a visão do todo.
- ◇ **composição**: identifica-se um conjunto de objetos que reunidos formam um objeto maior. Por exemplo, barra de rolagem, menu e *text area* compõem uma janela.
- ◇ **partes comuns**: se duas ou mais classes apresentam um conjunto de atributos semelhantes podemos agrupá-los num objeto desde que tenham uma identidade.

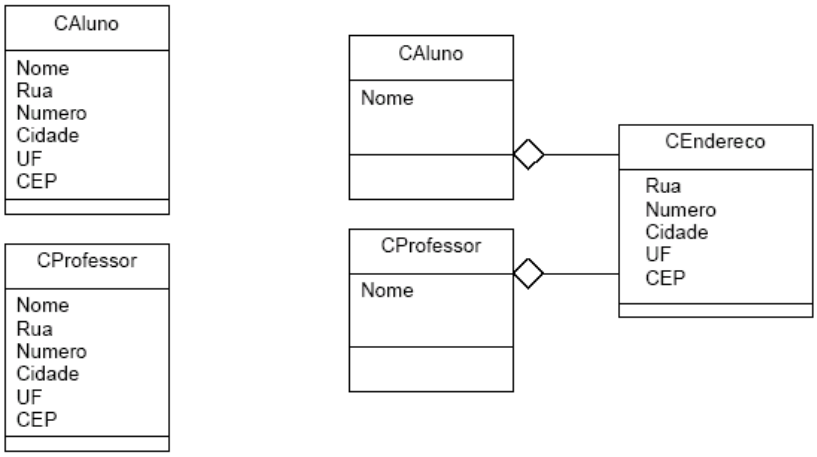


Figura 62: Exemplo de agregação a partir de partes comuns.

No exemplo do conversor Celsius-Fahrenheit, a associação entre a classe ConversãoCF e Histórico pode ser substituída por uma agregação por composição. Se o objeto histórico é destruído todas as conversões também o serão.

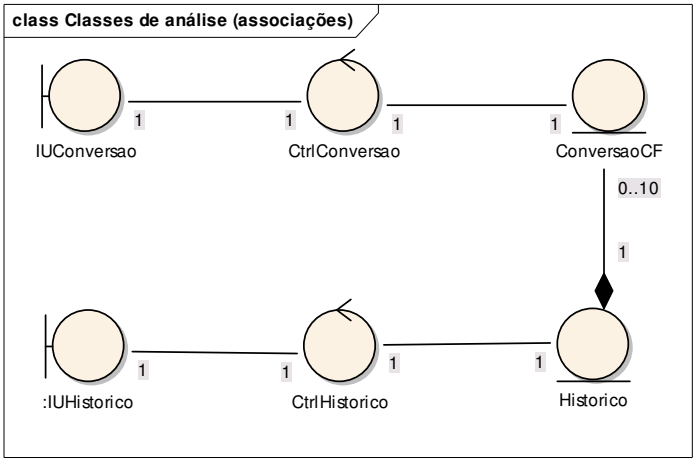


Figura 63: substituição da associação entre ConversãoCF e Histórico por uma agregação.

3 GENERALIZAÇÃO

A relação de especialização é utilizada quando um conjunto de classes compartilha comportamentos e atributos, pode-se então generalizá-las agrupando seus comportamentos e atributos comuns.

A relação de generalização recebe muitos nomes, tais como herança e especialização. A leitura pode ser feita, partindo-se da classe base em direção a derivada, como “é um tipo de”, “é uma” ou “é um”. Na figura 64, aluno (classe derivada) é um tipo de pessoa (classe base).

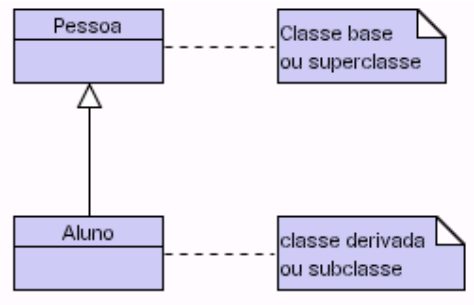


Figura 64: exemplo de relação de generalização/especialização.

A implementação da relação de generalização ocorre pelo mecanismo de herança. Por herança, a classe derivada incorpora os métodos e atributos da classe base. A classe derivada pode incluir novos métodos e atributos e redefinir os métodos herdados (polimorfismo de reescrita). Em UML quando repetimos a assinatura de uma operação numa classe derivada significa que temos uma nova implementação. No exemplo da Figura 65 o cálculo do IPVA na classe derivada leva em conta a quantidade de cavalos do carro.

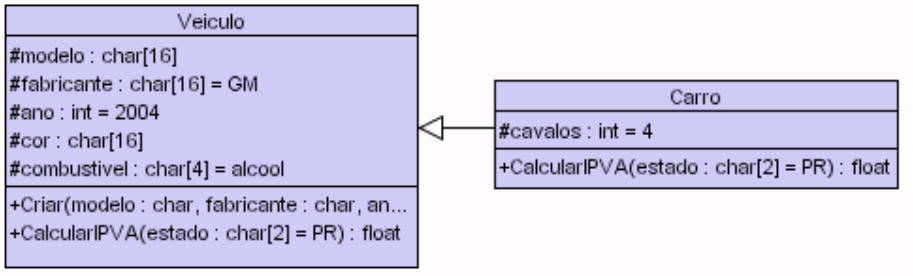


Figura 65: Relação de generalização/especialização com reescrita de método.

3.1 Hierarquia de classes

Nas classes base colocamos os atributos e operações que são comuns a todas as classes derivadas, evita-se redundância e facilita-se reutilização. A figura 66 ilustra uma situação onde atributos e operações comuns às classes Aluno e Professor foram colocados na classe base Pessoa, desta forma não é preciso redeclará-las em cada uma das especializações.

Figura 66: Exemplo de hierarquia de classes.

3.2 Levantamento de generalizações

Basicamente, há duas maneiras de se identificar generalizações:

- Identificação de partes comuns: a partir de classes específicas tenta-se estabelecer classes genéricas.
- Síntese de classes base: pode-se iniciar a concepção a partir de classes abstratas tentando especializá-las.

3.3 Qualidade de uma classificação

Uma boa classificação é estável e extensível.

- **Estável**: os critérios de classificação não mudam ao longo do tempo.
- **Extensível**: é fácil de incluir novas classes derivadas na hierarquia.
- Não classificar os objetos em função de critérios que caracterizam seus estados ou critérios muito vagos como ilustra a figura 67. Ao tentar estender a hierarquia desta figura com uma janela do tipo *Dialog*, torna-se evidente que o critério de classificação não foi bem escolhido.

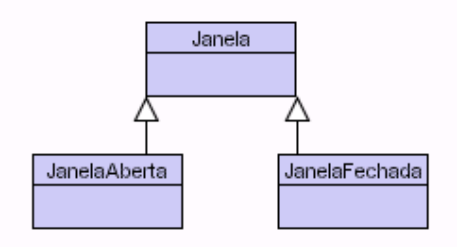


Figura 67: exemplo de taxonomia em função de estados.

Dica

Uma boa relação de herança deve respeitar o **princípio da substituição**: qualquer instância da classe base pode ser substituída por uma instância de uma classe derivada sem alterar a semântica de um programa escrito para (em função da) a classe base.

3.4 Herança múltipla

Uma classe é derivada de mais de uma classe base. Ocorre perda conceitua,l pois uma classe base não representa um caso geral completo da classe derivada, ela representa uma parte do conceito representado na classe derivada. No exemplo abaixo, Cautomovel não é uma generalização completa de Ccarro pois não leva em conta aspectos do carro ligados ao patrimônio.

```
class CAutomovel
{   char modelo[30];
    char fabricante[30];
    int  ano;
    char cor;
  public:
    DefineAuto(char,char,char,int);
};

class CBemMovel
{   int   numPatrimonio;
    float preco;
    int   depreciacao;
    int   anoCompra;
  public:
    DefineBem(int, float, int, int);
};

class CCarro::public CAutomovel, public
CBemMovel
{   int n_portas;
    int placa;
  public:
    DefineCar(char,char,char,int,int,int)
};
```

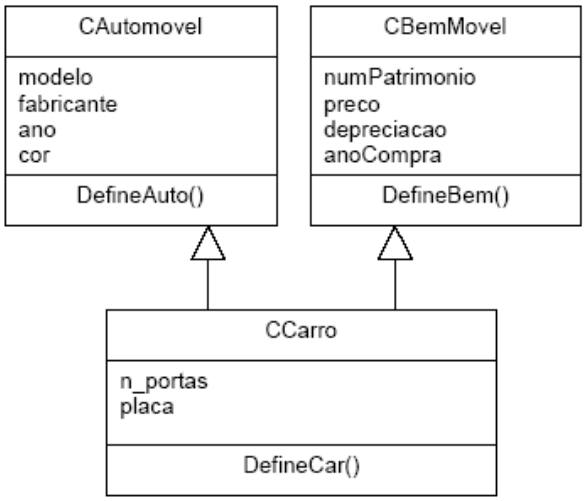
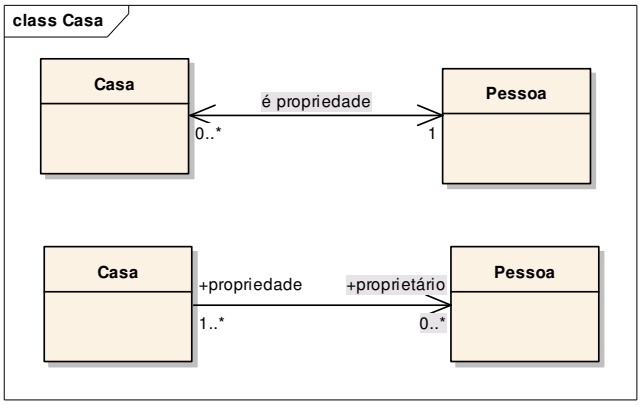


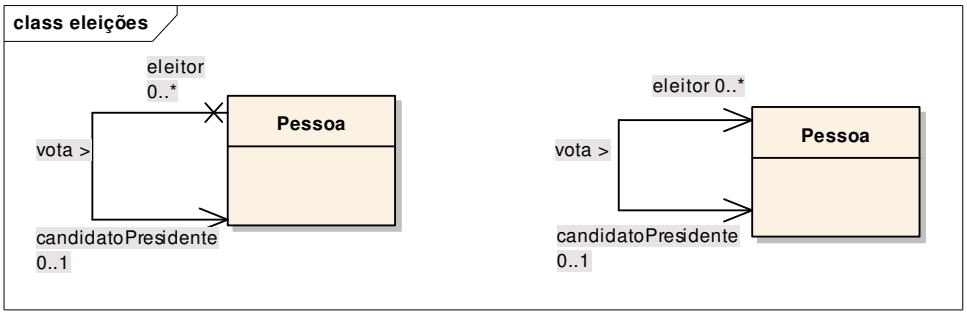
Figura 68: Exemplo de herança múltipla (extraída da apostila do Paulo)

4 EXERCÍCIOS

1. Em relação aos relacionamentos abaixo responda:
   a. Qual a representação mais correta – a primeira ou a segunda relação? Por quê?
   b. O que é preciso mudar na segunda relação para representar que uma casa possui diversos proprietários ao longo do tempo?

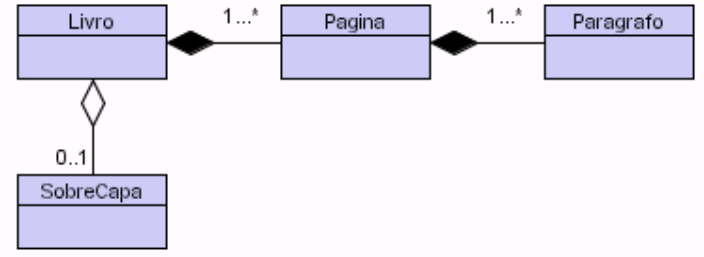


2. Qual a diferença de interpretação entre as duas representações? Qual seria mais indicada para um tribunal regional eleitoral?

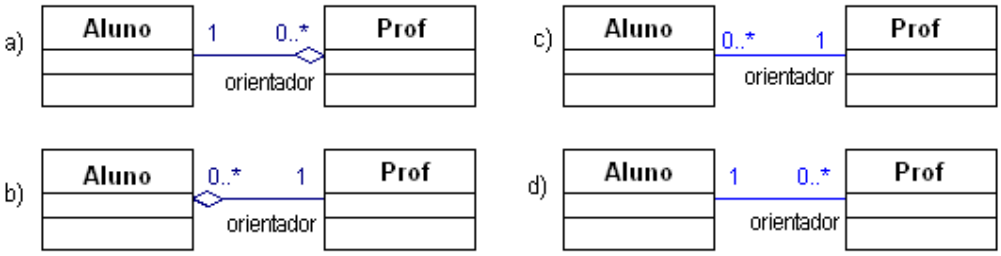


3. Represente um e-mail na forma de um diagrama de classes. Identifique os componentes (destinatário, assunto, etc.) e suas relações (fazer o diagrama no editor).

4. Qual a diferença de interpretação entre os relacionamentos livro-sobrecapa e livro-páginas?



5. Todo aluno matriculado em trabalho de diplomação será orientado por um professor. Alguns professores orientam vários alunos e outros, nenhum. Qual dos diagramas melhor representa esta relação?



6. Construa uma hierarquia de classes para os seguintes tipos de obra: romance, livro de ficção, livro de auto-ajuda, gibi, rock, MPB, filme ficção e comédia.

7. Um cliente pode fazer diversos contratos de locação de carros numa locadora de veículos. A locadora aluga caminhões, carros de passeio categoria A, B e C e motos. Os contratos diferem em valor e imposto segundo o tipo de veículo locado. Construa um diagrama de classes para representar as classes e seus relacionamentos.

8. Retoma as classes de análise e o diagrama de seqüência do exercício 6 (pg. 51) e defina as relações entre as classes de análise (arquivo disponível na página do curso versão Enterprise Architect).

VII PROJETO DE CASOS DE USO

De forma geral, a atividade de projeto de casos de uso é a transformação das classes de análise em uma ou mais classes de projeto. A representação das classes na atividade de projeto é muito mais próxima da implementação, ou seja, leva em conta as tecnologias que são utilizadas na implementação do sistema.

1 PROJETO

Resultados

Os resultados da atividade de projeto de casos de uso são:

- ◇ Classes de projeto
- ◇ Operações e atributos de cada classe
- ◇ Diagramas de interação para mostrar como as classes de projeto colaboram para realizar os processos do negócio capturados nos casos de uso.
- ◇ Arquitetura de software (*SAD Software Architecture Document*)
- ◇ Interfaces com subsistemas específicos ou software de terceiros
- ◇ Subsistemas

Método

- ◇ Criar realizações de casos de uso: na análise, as realizações contêm classes de análise e objetos que aparecem em diagramas de classes e de interação. No projeto, são utilizados praticamente os mesmos diagramas, a diferença é que as informações são mais próximas da implementação.
- ◇ Descrever interações entre objetos e refinar diagrama de classes
- ◇ Simplificar diagramas de seqüência utilizando subsistemas (opcional)
- ◇ Descrever comportamentos associados à persistência

2 CLASSES DE PROJETO

Na atividade de análise, ao construir os diagramas de interação (seqüência e comunicação), considerou-se que os objetos estavam disponíveis e persistidos, não havia preocupação com a criação e destruição dos mesmos e nem em acessá-los e associá-los a outros objetos. Na atividade de projeto, estas questões devem ser levadas em conta. Além disso, classes de análise podem ser transformadas em várias classes de projeto, reduzidas a um método de uma classe de projeto, suprimidas, etc.

Classes de Análise x Projeto

Qual a diferença entre uma classe de análise e uma de projeto?

- ◇ Uma classe de projeto é especificada na linguagem de programação alvo (tipos de dados, operações, atributos são definidos utilizando-se a sintaxe da linguagem de programação).
- ◇ As visibilidades dos atributos e operações de uma classe de projeto são especificadas na maior parte das vezes.

- ◇ As relações entre classes num diagrama de classes de projeto influenciam diretamente na implementação destas classes. Por exemplo, associações e agregações são mapeadas para atributos das classes para que os objetos envolvidos possam se referenciar.
- ◇ Operações das classes de projeto são traduzidas de forma direta em código.

3 ESTUDO DA INTERAÇÃO ENTRE OBJETOS

Esta seção exemplifica o estudo da interação entre objetos para realizar o caso de uso conversão de Celsius para Fahrenheit em uma aplicação *desktop* com interface textual. O intuito é mostrar como refinar um diagrama de classes a partir do detalhamento do diagrama de seqüência.

3.1 Realização Converter Celsius-Fahrenheit desktop

O diagrama de seqüência representa o cenário onde o meteorologista realiza várias conversões de Celsius para Fahrenheit que são guardadas numa coleção denominada histórico. Ao decidir encerrar a sessão, o histórico é persistido em disco por meio de um objeto serializável.

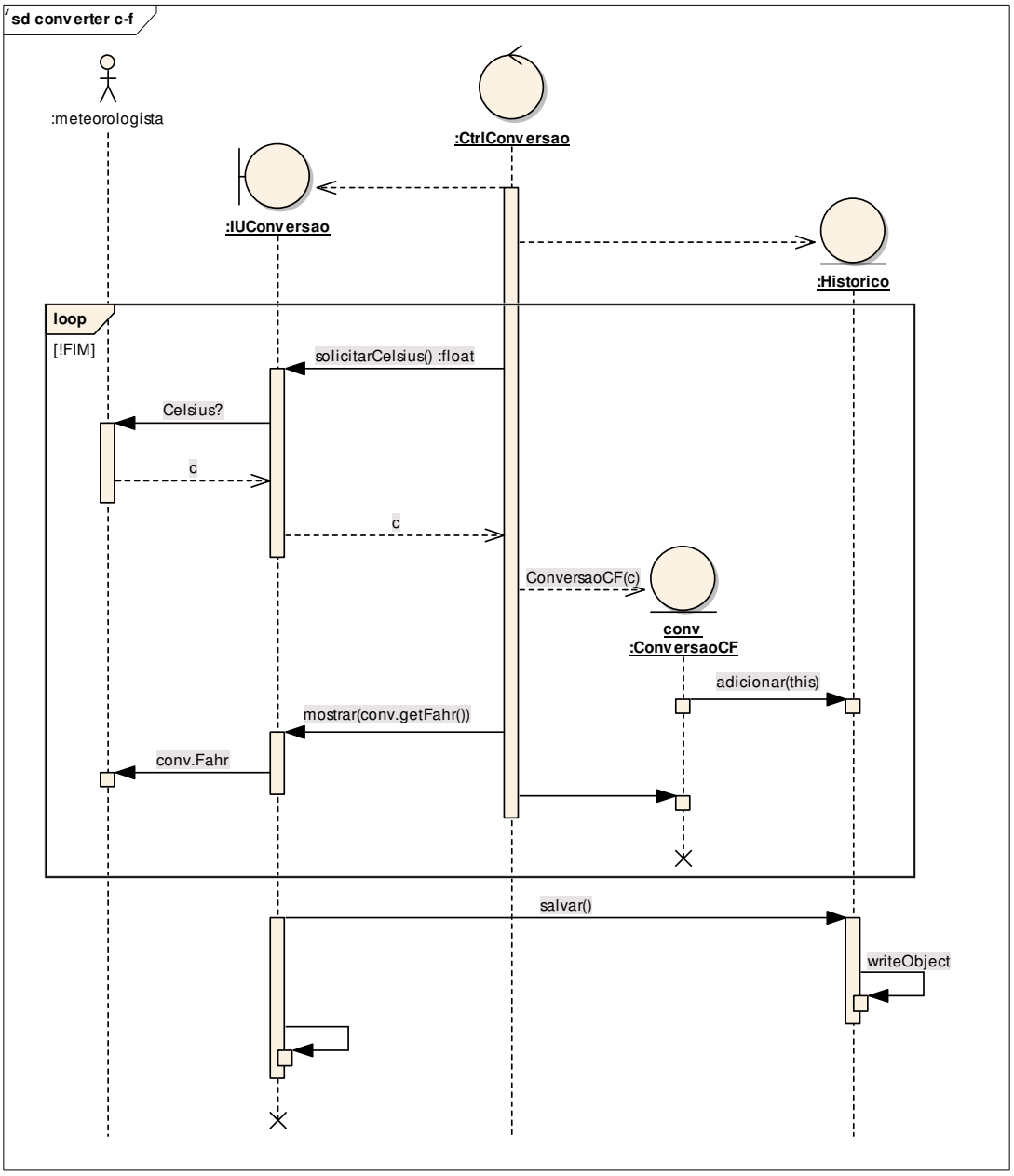


Figura 69: Diagrama de seqüência (converter Celsius-Fahrenheit textual).

Das colaborações expressas na Figura 69, pode-se derivar o seguinte diagrama de classes de projeto.

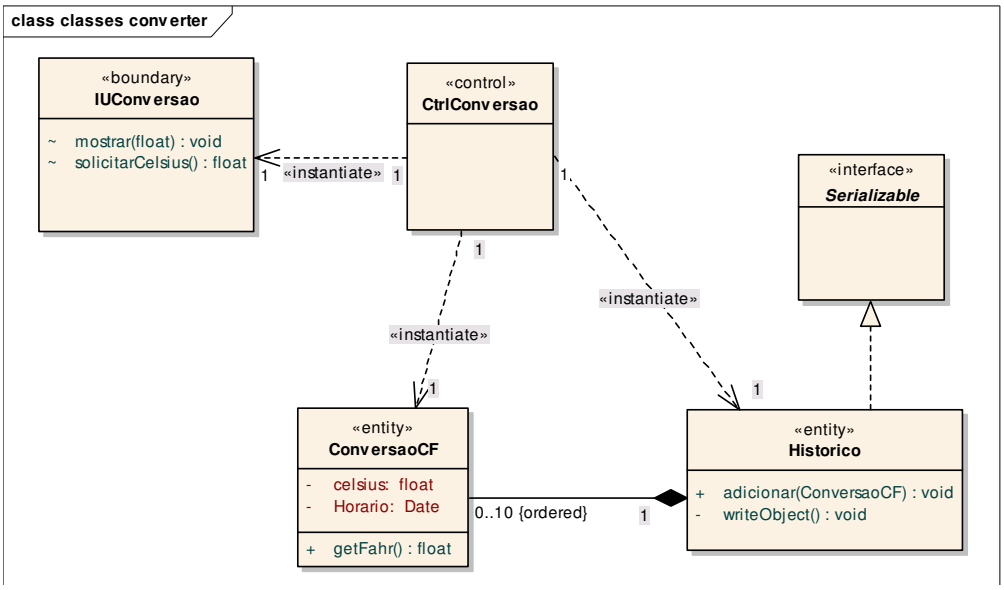


Figura 70: Diagrama de classes de projeto (converter Celsius-Fahrenheit textual).

Observar que as associações entre CtrlConversao e as demais classes foram transformadas em dependências. Deixou-se claro que Histórico implementa (realiza) uma classe de interface serializável. Histórico é uma coleção ordenada de objetos ConversaoCF. Os atributos e operações também possuem tipos de dados compatíveis com Java (a suposta linguagem de implementação) e visibilidades. É importante notar que as mensagens **que chegam num objeto** são transformadas em operações (ex. solicitarCelsius() é transformada numa operação em IUConversao).

Neste exemplo didático, as classes de análise não sofreram grandes modificações. Também não houve necessidade de dividir o sistema em subsistemas ou pacotes.

4 REFINAR O DIAGRAMA DE CLASSES

A partir dos diagramas de seqüência de projeto, detalham-se as relações do diagrama de classes. Alguns refinamentos no diagrama de classes podem ser feitos em função da implementação desejada.

4.1 Dependência

Uma relação de dependência indica que uma classe depende do auxílio de outra classe para implementar seus comportamentos. É um relacionamento utilizado prioritariamente na fase de projeto. Dadas duas classes A e B, as dependências possíveis entre elas podem ser assim resumidas:

- **Variável local**: Classe A utiliza em alguma operação uma variável de tipo B
- **Parâmetro de operação**: Classe A utiliza como parâmetro de alguma operação cujo tipo é B.
- **Instanciação**: a Classe A instancia um objeto de B.

Para representar os tipos de dependência ilustrados, estereótipos podem ser utilizados (figura 71).

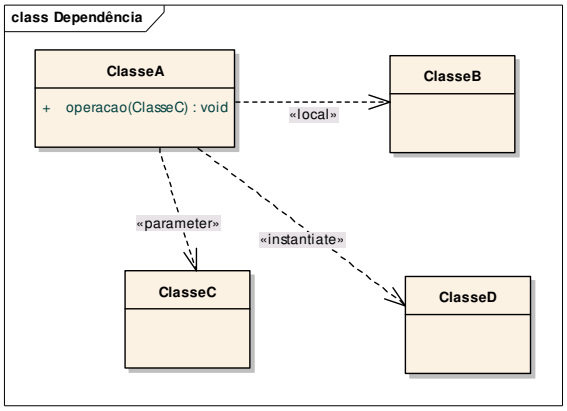


Figura 71: Alguns tipos de dependência.

4.2 Implementação de associações e agregações

Não há diferença entre a implementação de uma associação e uma agregação. Portanto, o texto que segue serve indistintamente para os dois tipos de relações. Também não há diferenças significativas entre uma agregação por associação e uma agregação por composição. Nesta última, a lógica da classe *todo* deve garantir que os comportamentos se propagam em direção às partes sempre que necessário. Em uma composição, uma parte é exclusiva do todo e, portanto, não deve participar de outras agregações.

Multiplicidade 1:1 e variações

Associações unidirecionais de multiplicidade 1:1 são de simples implementação: basta colocar um atributo na classe origem da relação. Se a relação for bidirecional, coloca-se um atributo em cada uma das classes que participam da associação.

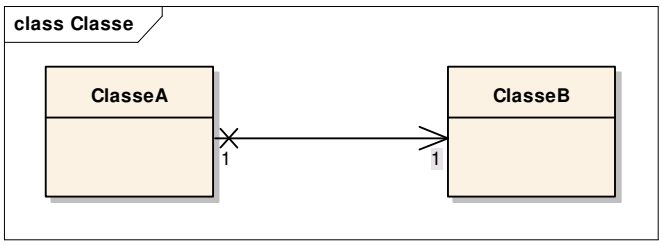


Figura 72: Implementação de uma associação unidirecional de multiplicidade 1:1

O código abaixo ilustra uma implementação possível para a associação da figura 72.

```
1   class ClasseA {
2       Private ClasseB objB = new ClasseB(); // pode ser instanciado em outro local
3       // outros atributos
4       // métodos
5   }
```

O código abaixo ilustra uma implementação para uma associação **bidirecional** de multiplicidade 1:1.

```
1   class ClasseA {
2       Private ClasseB objB = new ClasseB(); // pode ser instanciado em outro local
3       // outros atributos
4       // métodos
5   }
6   class ClasseB {
7       Private Classe objA = new ClasseA(); // pode ser instanciado em outro local
8       // outros atributos
9       // métodos
```

```
10   }
```

Multiplicidade 1:* e variações

Quando a multiplicidade máxima é menor que infinito (e não muito grande) pode-se alocar espaço na instanciação do objeto do lado 1 e povoar o vetor neste instante ou posteriormente.

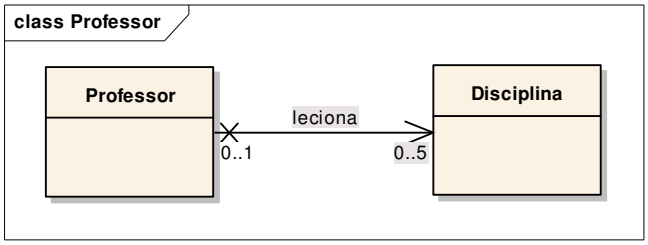


Figura 73: associação de multiplicidade 0..1:0..5

Por exemplo, para a associação representada na figura 73 as seguintes implementações são possíveis.

```
1    class Professor {
2       private Disciplina[] disciplinas = new Disciplina[5]; // aloca espaço para 5 objs
3
4       public void adicionarDisciplinas(Disciplina[] disc){
5           // inicializar vetor disciplinas com parâmetro disc
6       }
7    }
```

```
1    class Professor {
2       private Disciplina[] disciplinas = {
3           // instancia os objetos disciplina
4           new Disciplina(),
5           new Disciplina(),
6           new Disciplina(),
7           new Disciplina(),
8           New Disciplina(),
9       };
10   }
```

Quando multiplicidade máxima for igual a "*" (figura 74) recomenda-se utilizar uma coleção parametrizada como ilustra o código seguinte.

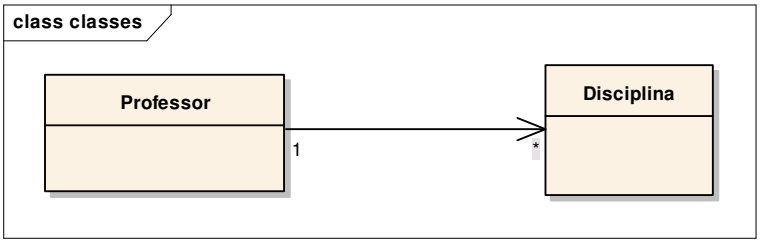


Figura 74: associação de multiplicidade 1:*

```
1    class Professor {
2       private Vector<Disciplina> disciplinas = new Vector<Disciplina>(4, 2);
3       public void adicionarDisciplinas(Disciplina[] disc){
4          disciplinas.add(disc[0]);
5          ...
6       }
7    }
```

Dica Enterprise Architect: para gerar código nesta situação, em tools>options>Java>[Collection classes], definir Vector como default collection class. Nas propriedades da associação, no lado target (*), colocar Member Type = Vector<Disciplina>.

Multiplicidade *:*

A implementação de associações com multiplicidade *:* envolve a utilização de coleções. Para o exemplo da figura 75, utilizou-se a coleção Vector. Em seguida, mostra-se o código Java correspondente.

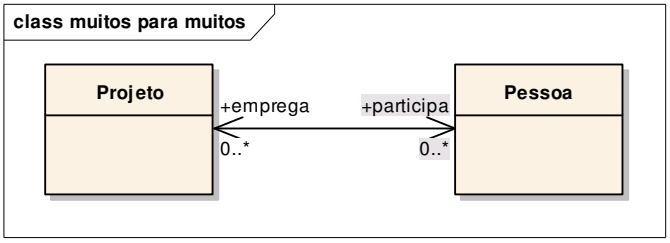


Figura 75: Associação muitos para muitos.

```
1   public class Projeto {
2
3      public Vector<Pessoa> participa;
4      ...
5   }


1   public class Pessoa {
2      public Vector<Projeto> emprega;
3      ...
4   }
```

Associações reflexivas

O mapeamento para Java de uma associação reflexiva não difere dos mapeamentos entre duas classes distintas.

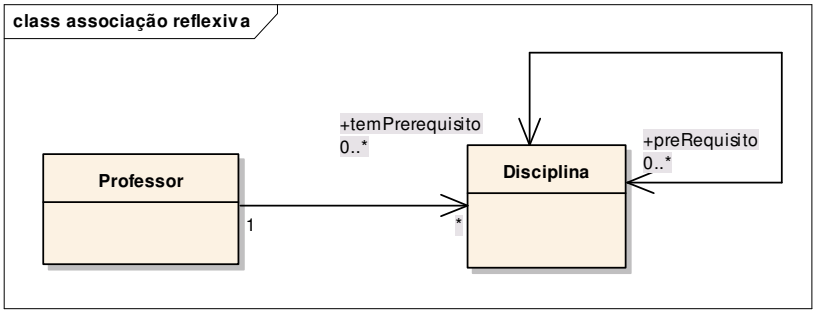


Figura 76: Associação reflexiva muito para muitos.

A implementação da classe Disciplina é ilustrada no código seguinte:

```
1   public class Disciplina {
2
3      public Vector<Disciplina> temPrerequisito;
4      public Vector<Disciplina> preRequisito;
5
6      ...
7   }
```

Classes associativas

Classes associativas podem ser transformadas em classes comuns e, a partir daí, as associações caem nos casos anteriores. Por exemplo, a figura 77 ilustra um caso onde um projeto pode ter várias pessoas e uma pessoa pode participar de vários projetos. Cada participação possui uma carga horária, data de entrada no projeto e de saída.

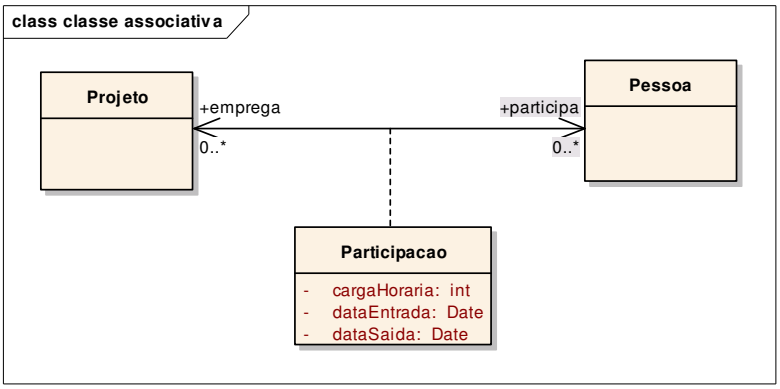


Figura 77: Classe associativa.

O diagrama anterior poderia ser representado da seguinte forma

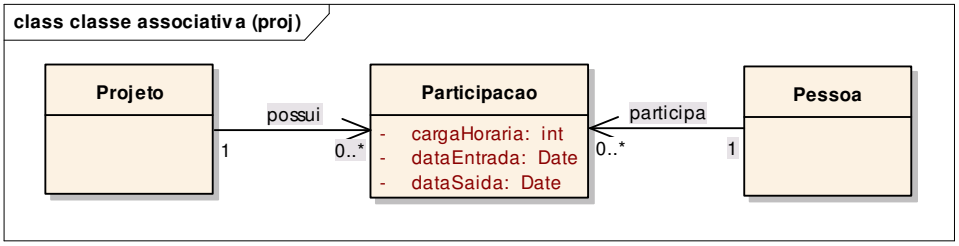


Figura 78: Classe associativa transformada em classe "normal".

A implementação correspondente ao diagrama da figura 78 ficaria:

```
1   public class Projeto {
2      public Vector<Participacao>  m_Participacao;
3      ...
4   }

1   public class Pessoa {
2      public Vector<Participacao>  m_Participacao;
3      ...
4   }

1   public class Participacao {
2      private int cargaHoraria;
3      private Date dataEntrada;
4      private Date dataSaida;
5      public Projeto projeto;    // se relação for bidirecional
6      Public Pessoa pessoa;      // se relação for bidirecional
7      ...
8   }
```

Agregação por composição

É uma agregação de fato, a criação/alocação de um objeto é feita dentro de outro. Existe uma relação forte, pois quando *todo* é destruído as partes também o serão, ou seja, a eliminação do todo se propaga para as partes. O *todo* e as *partes* agregado têm tempos de vida semelhantes.

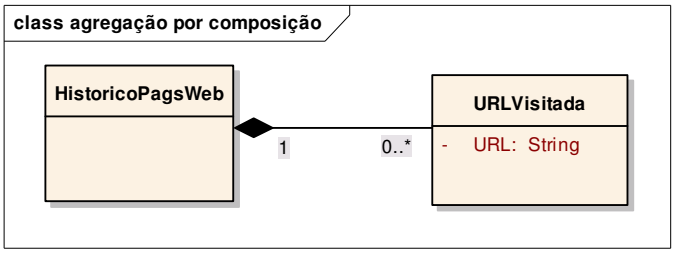


Figura 79: Agregação por composição.

A seguir duas implementações possíveis para a composição da figura 79. Um histórico é uma agregação de URLS visitadas, portanto, uma vez que o histórico é destruído as instâncias de URLs visitadas também o serão. Em Java não há necessidade de destruir os objetos devido ao *Garbage Collection*. Todos os objetos criados são eliminados da memória quando não são mais referenciados por nenhuma variável, portanto na implementação da agregação por composição não precisamos nos preocupar com a igualdade dos tempos de vida do todo e das partes.

```
1   class URLVisitada {
2      private String url;
3
4      URLVisitada (String url) {
5         this.url = url;
6      }
7   }
8
9   class Historico {
10     // cria três objetos da classe URLVisitada
11     private URLVisitada[] historico = {  // poderia ser com alguma coleção
12        new URLVisitada("http://www.uol.com.br"),
13        new URLVisitada("http://www.terra.com.br"),
14        new URLVisitada("http://www.lemonde.fr"),
15     };
16  …
17  }
```

É possível implementar uma agregação com classes aninhadas, como ilustra o código abaixo.

```
1   class Historico {
2      // como private, a URLVisitada só pode ser instanciada dentro do escopo do todo
3
4      private class URLVisitada {
5         private String url;
6         URLVisitada (String umaURL) {
7            url = umaURL;
8         }
9         ...
10     }
11     private Vector<URLVisitada> historico = new Vector<URLVisitada>(4, 2);
12     ...
13
14  }
```

Agregação por associação

A implementação é idêntica a da associação.

Realização

É uma relação que indica que uma classe realiza ou segue o contrato definido por uma classe de interface (no sentido Java). No diagrama de classes do conversor de Celsius-Fahrenheit, a classe Histórico realiza a interface Serializable (*implements*).

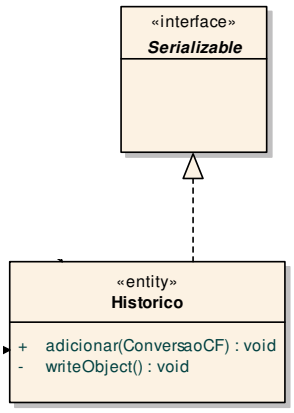


Figura 80: Relação de realização.

A implementação em Java de uma realização é mostrada em seguida.

```
1   public interface Serializable {
2    ...
3   }

1   public class Histórico implements Serializable {
2    ...
3   }
```

5 SUBSISTEMAS

Após estudar em detalhes a interação entre objetos e refinar o diagrama de classes, é possível identificar subsistemas. Uma forma de identificar subsistemas é observar comportamentos repetidos nos diagramas de seqüência.

Subsistemas podem conter outros elementos de modelagem (ex. classes, pacotes) e apresentam comportamento definido pelas interfaces que realiza. Em UML, subsistemas são representados como pacotes, porém com um estereótipo <<subsistema>>.

Considere o exemplo onde durante a análise foi identificada uma classe de análise <<fronteira>> com um sistema de faturamento. No projeto, esta classe se mostrou muito complexa e foi transformada num subsistema onde várias classes colaboram para realizar as responsabilidades previstas.

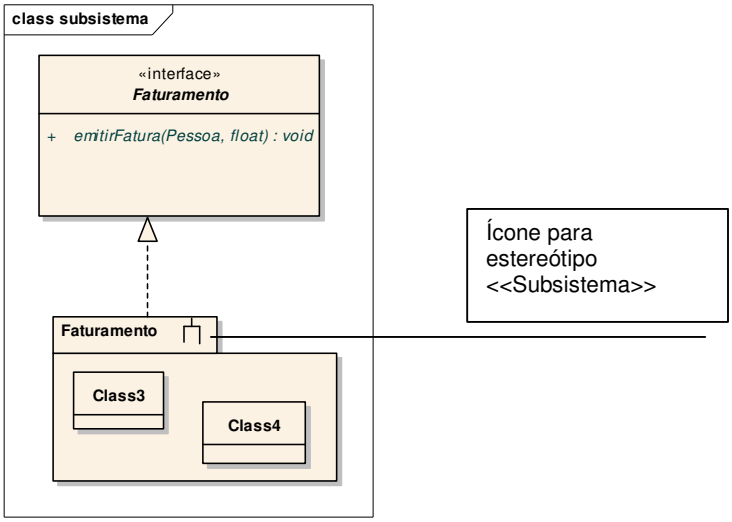


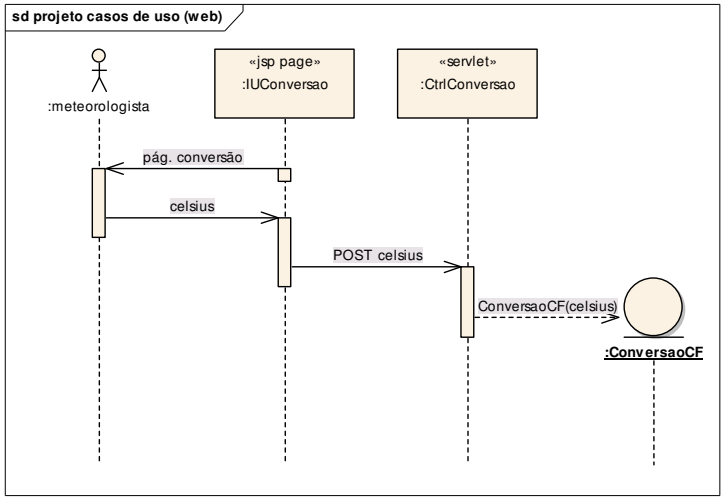
Figura 81: Exemplo de subsistema.

6 COMPORTAMENTOS ASSOCIADOS À PERSISTÊNCIA

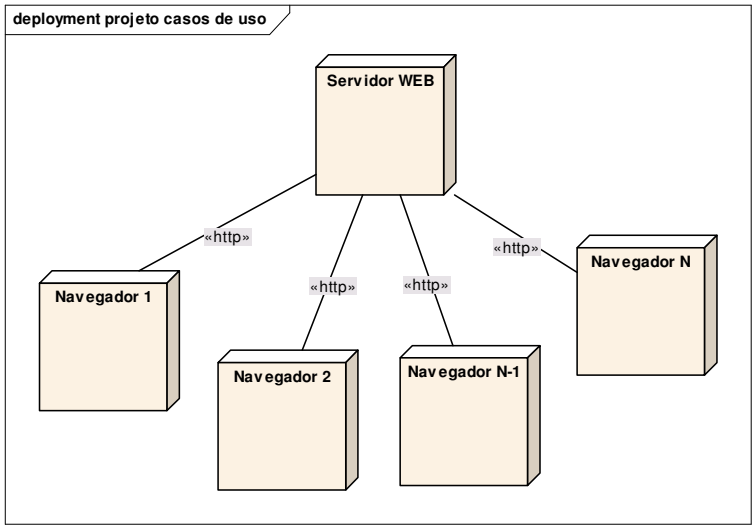
Não é interessante que objetos da camada de modelo conheçam detalhes do banco de dados (ex. conexão, tabelas, formato de registros). O problema é que qualquer mudança no esquema do banco de dados se reflete nos objetos do modelo. Para desacoplar os objetos da camada do modelo do banco de dados é possível utilizar padrões, tais como *factory*, que cria objetos do modelo já preenchidos com informações do banco de dados e *da*, um intermediário que faz a ligação entre o(s) objeto(s) do modelo e o banco de dados. Pode haver aumento da complexidade se o acesso ao banco de dados for simples. Por outro lado diminui-se a dependência do modelo em relação ao banco de dados. *Dao* pode ser feito num nível de granularidade maior, por exemplo, para manipular um conjunto de registros (visão) ou uma tabela.

7 EXERCÍCIOS

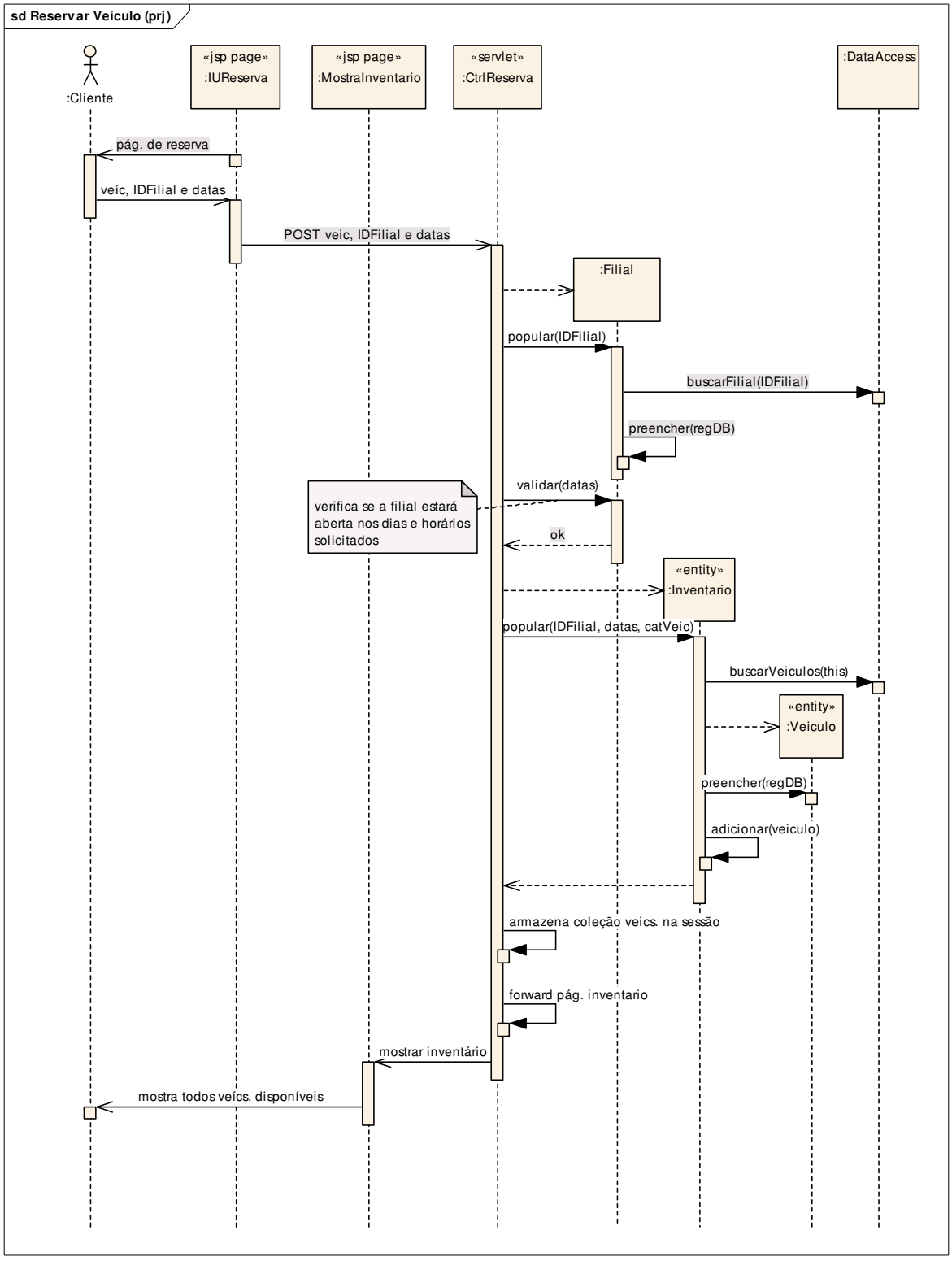
1. Supor que no projeto do conversor de graus Celsius para Fahrenheit, a persistência do Histórico agora é somente pelo tempo de uma sessão Web. A partir destas decisões de projeto, uma nova realização de projeto do caso de uso Converter Celsius-Fahrenheit deve ser realizada. Supor o cenário onde o cliente faz o primeiro acesso ao sistema, logo uma sessão deve ser criada para armazenar as conversões realizadas daquele momento em diante. O diagrama de seqüência parcial é mostrado em seguida. Complete o diagrama de seqüência e refine o diagrama de classes.



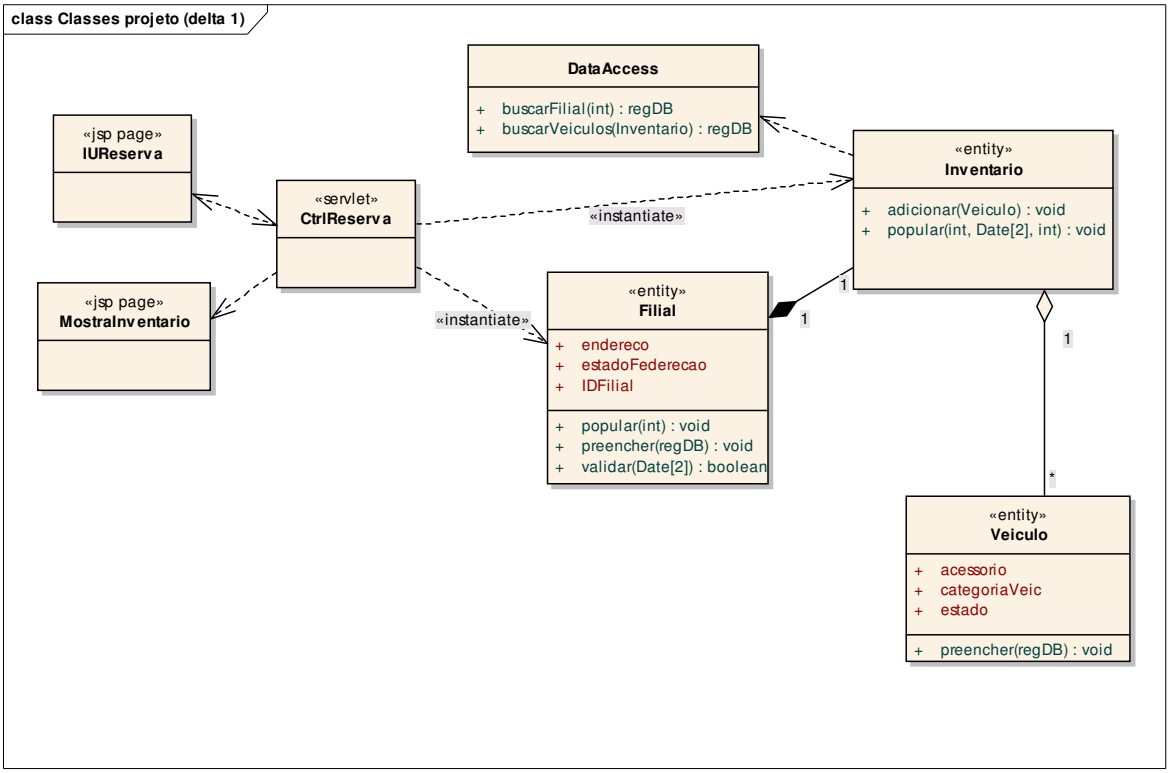
2. Para o sistema de locação de veículos, caso de uso Reservar Veículo, supor que na fase de projeto o arquiteto de software decidiu por uma arquitetura Web como mostra o diagrama de implantação. Além disso, o arquiteto de software optou por utilizar o conhecimento da equipe desenvolvedora em Java, Java ServerPages, servlets e Java Script.



Com estas decisões, o diagrama de seqüência "reservar veículo" foi parcialmente refinado, ou seja, somente a parte de seleção de veículo por parte do cliente em um cenário onde tudo funciona como mostra a figura seguinte. Não está incluído o tratamento do perfil do cliente.



A partir do diagrama de seqüência, refinou-se o diagrama de classes. A figura seguinte mostra apenas as modificações em relação ao diagrama de classes de análise. Note que operações foram adicionadas e outras classes dependentes da arquitetura escolhida, tais como as páginas JSP.



Pede-se:
Faça o diagrama de seqüência que mostre a efetivação da reserva por parte do usuário
Faça as mudanças necessárias no diagrama de classes.
(*arquivo disponível na página do curso – formato Enterprise Architect)*

VIII DIAGRAMA DE ESTADOS

1 ELEMENTOS BÁSICOS

Casos de uso e cenários (diagramas de interação – seqüência e comunicação) são utilizados para descrever o comportamento do sistema. Frequentemente é necessário descrever o comportamento interno de uma classe de objetos. Não é necessário fazer diagramas de estado para todas as classes do sistema, somente para as mais complexas onde o comportamento dos objetos é modificado pelos diferentes estados.

Um diagrama de estados mostra os eventos que causam as **transições** de **estados** e as **ações/atividades** tomadas em conseqüência da transição. Condições de guarda podem ser associadas aos eventos, neste caso, a ocorrência do evento não garante a ocorrência da transição, é preciso que a condição de guarda seja satisfeita para que ela ocorra.

- **Estado** é a situação na qual se encontra o objeto. Ao longo da sua vida, um objeto é criado no estado inicial, passa por estados intermediários e morre no estado final. Estados recebem nomes no particípio ou gerúndio. Por exemplo, um objeto pode estar emprestado ou disponível.
- **Evento (*trigger event*)**: é algo que ocorre de forma instantânea no tempo e pode ocasionar uma transição de estado em um objeto.
- **Ação**: uma ação é algo executado de forma imediata e atômica, ou seja, o tempo de execução é muito pequeno e a ação não pode ser interrompida. Está frequentemente associada a uma transição embora possa aparecer dentro de um estado relacionada às palavras-chave *entry* e *exit* ou a transições internas a um estado.
- **Atividade**: é similar a uma ação, porém pode ser interrompida. É associada a um estado por meio da palavra-chave *do*.
- **Condição de guarda**: expressão lógica que deve ser verdadeira na ocorrência do evento para que a transição ocorra.

1.1 Notação básica

Diagrama de estados é um grafo dirigido onde os nodos são os estados e os arcos, transições entre estados como ilustra a figura seguinte.

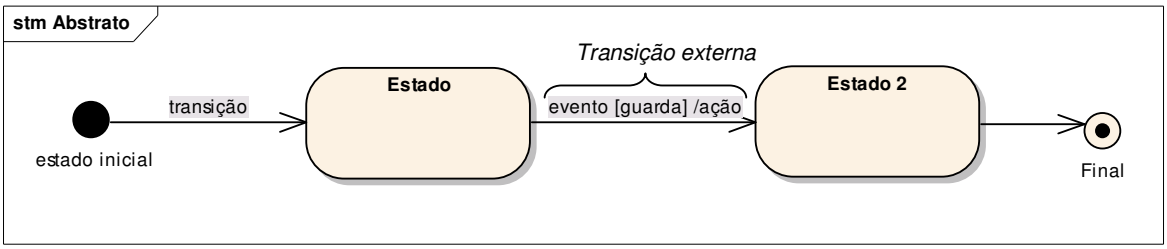


Figura 82: elementos básicos de um diagrama de estados.

A Figura 83 mostra o comportamento de uma janela que inicia no tamanho original e pode ser minimizada, podendo trocar de estados várias vezes, até ser fechada.

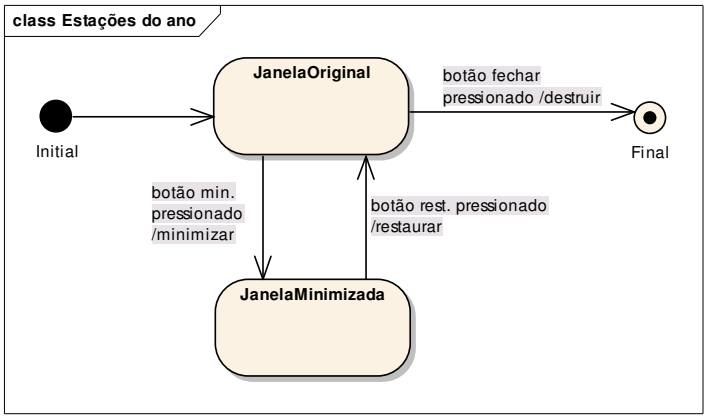


Figura 83: Diagrama de estados para uma janela (original, minimizada)

1.2 Ação nos estados (entry e exit)

Ações podem ser representadas nos eventos ou nos estados. Normalmente, a representação nos estados permite simplificar o diagrama. Observar na figura 84 que toda vez que o Livro passa ao estado Disponível a ação de NotificarInteressados é executada. Ao invés de repetir a ação em cada transição, pode-se colocá-la no interior do estado associada à palavra-chave *entry* (figura 85). Cada vez que o estado Disponível é alcançado, executa-se a ação *NotificarInteressados.*

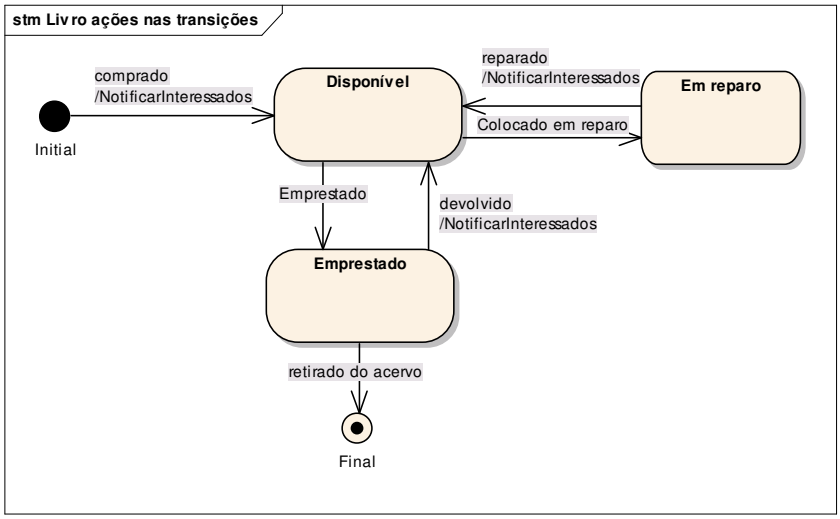


Figura 84. Ações nas transições que levam ao estado Disponível.

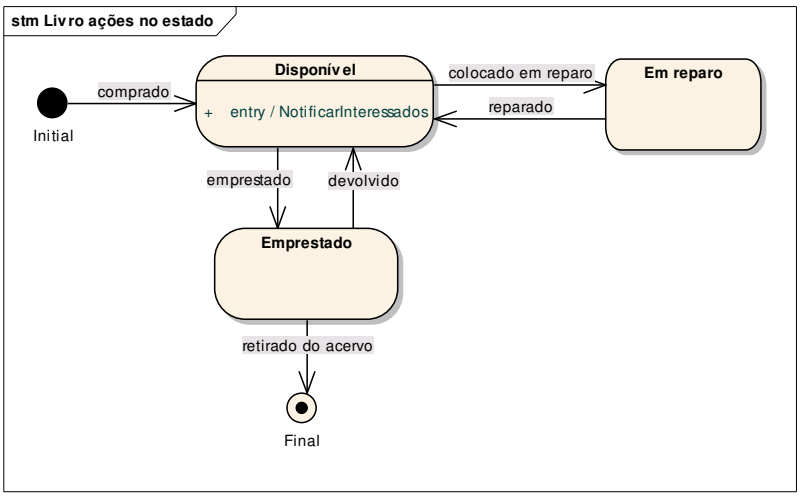


Figura 85. Ação no estado Disponível.

De forma similar, pode-se utilizar a declaração *exit <ação>* nos estados para representar que a *<ação>* será executada toda vez que se deixar o estado.

1.3 Atividade nos estados (*do*)

Uma atividade pode ser interrompida ou terminar por si só. Um atividade está sempre associada a um estado por meio da palavra-chave *do* (faça). Por exemplo, na a atividade *beep(tempo)* será executada toda vez que a impressora estiver sem papel. Esta atividade pode ser interropida a qualquer momento pela ocorrência do evento *impressão cancelada* ou *papel colocado*.

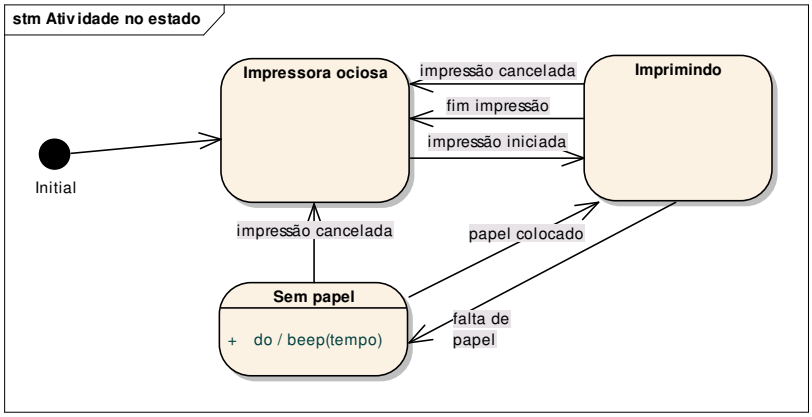


Figura 86. Atividade na transição Sem Papel.

1.4 Auto-transição

É possível que um evento provoque a transição de um estado para ele mesmo. Neste caso, todas as ações associadas ao estado por meio das palavras-chaves *entry* e *exit* e as ações associadas à palavra-chave *do* serão executadas. Na figura 87, toda vez que uma letra for teclada, dispara-se a transição do estado Contado letras para ele mesmo o que provoca a execução da ação *num++*. Notar que é possível definir atributos das classes nos estados onde são manipulados (num: int).

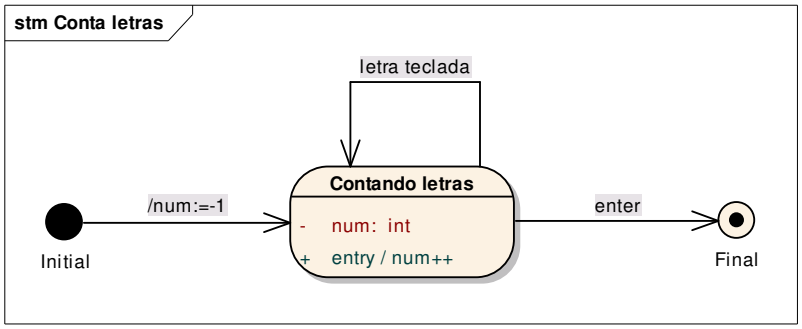


Figura 87: Auto-transição em máquina de estado.

1.5 Transição interna

É possível representar a ocorrência de um evento que não provoca uma mudança de estado, somente a execução de uma ação. Estes eventos são chamados de eventos internos e são representados no interior dos estados. A resposta a um evento interno difere daquela da auto-transição, pois não se deixa o estado para reentrar em seguida. Portanto, as ações associadas às palavras-chaves *entry e, exit* não são executadas e a atividade porventura em execução não é interrompida. No exemplo seguinda (Figura 88), há uma transição interna disparada pela tecla F1. Neste caso, a ação *mostrar help* será executada, porém a ação num++ não será executada.

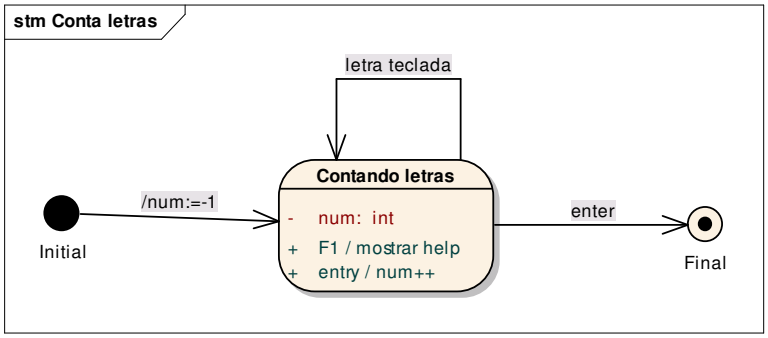


Figura 88: Exemplo de transição interna.

1.6 Ordem de execução de ações e atividades

A ordem de execução das ações e atividades é a seguinte, supondo-se que a máquina já se encontra num estado A e passa ao estado B pela ocorrência de um evento AB:

1. se existe uma atividade em execução em A, ela é interrompida;
2. executa-se a ação *exit* do estado A;
3. executa-se a ação associada ao evento AB;
4. executa-se a ação *entry* do estado B;
5. executa-se a atividade *do* em B, se existir.

No exemplo da figura 89, supõe-se a seguinte seqüência de eventos: início, ev AB, ev interno, ev B e ev FIM. Para esta seqüência, as ações são executadas na seguinte ordem: ***início*** - A0 A1, A2, A3 – ***ev AB*** - AB, A5, A6 – ***ev interno*** - A7 (sem matar A6) – ***ev B*** – (interrompe A6) A8, A4, A5, A6 – ***ev FIM*** – A8, A9.

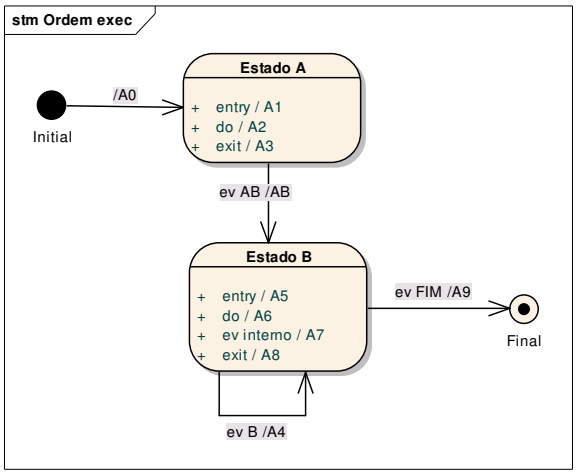


Figura 89: Ordem de execução das ações e atividades.

1.7 Condição de guarda

Condição de guarda é uma expressão lógica que deve ser satisfeita na ocorrência de um evento para que a transição correspondente ocorra. Por exemplo, na Figura 90, quando ocorrer o evento userInput há três opções mutuamente exclusivas dadas pelas condições de guarda:

1. opção = OK: a auto-transição sem ação associada será executada;
2. opção = NOK: a auto-transição com a ação mostrarMsg será executada;
3. opção = fim: o objeto será destruído.

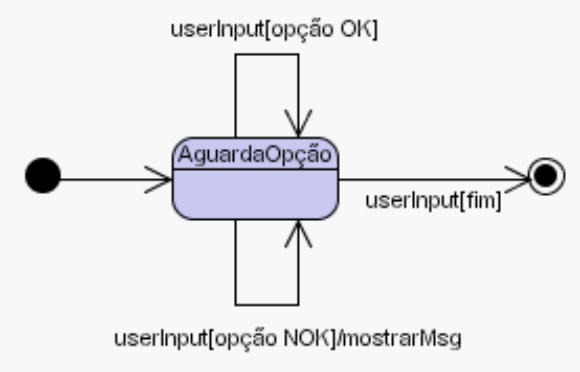


Figura 90: Condição de guarda.

A condição de guarda é importante para evitar conflitos entre transições. Se pela ocorrência de um evento, mais de uma transição pode ser disparada, então a máquina de estados é não determinística.

2 TIPOS DE EVENTOS

2.1 De chamada

É um evento síncrono, tipicamente uma chamada de método. Pode-se chamar um método da própria classe ou de outra classe. Por exemplo, a partir do diagrama de seqüência (figura 69, pg. 67) da realização do caso de uso *Converter Celsius-Fahreneheit* pode-se identificar que a classe CtrlConversão possui um estado inicial (onde instancia a interface e histórico) e passa automaticamente ao estado *Aguarda entrada* onde executa a ação associada à *entry.* Esta ação produz um evento de chamada na classe IUConversao que faz com que a máquina correspondente passe do estado *Aguarda cmdo controle*

para *Aguarda entrada do usuário.* Observar que do ponto de vista do CtrlConversao solicitarCelsius é uma ação e de IUConversao, um evento de chamada. Embora não esteja representado (e não é preciso representar todos os detalhes) fica subentendido que quando o usuário fornece um valor de entrada ao objeto da classe IUConversao, gera-se o evento *valor fornecido* no CtrlConversao (equivale à resposta ao evento de chamada *solicitarCelsius*).

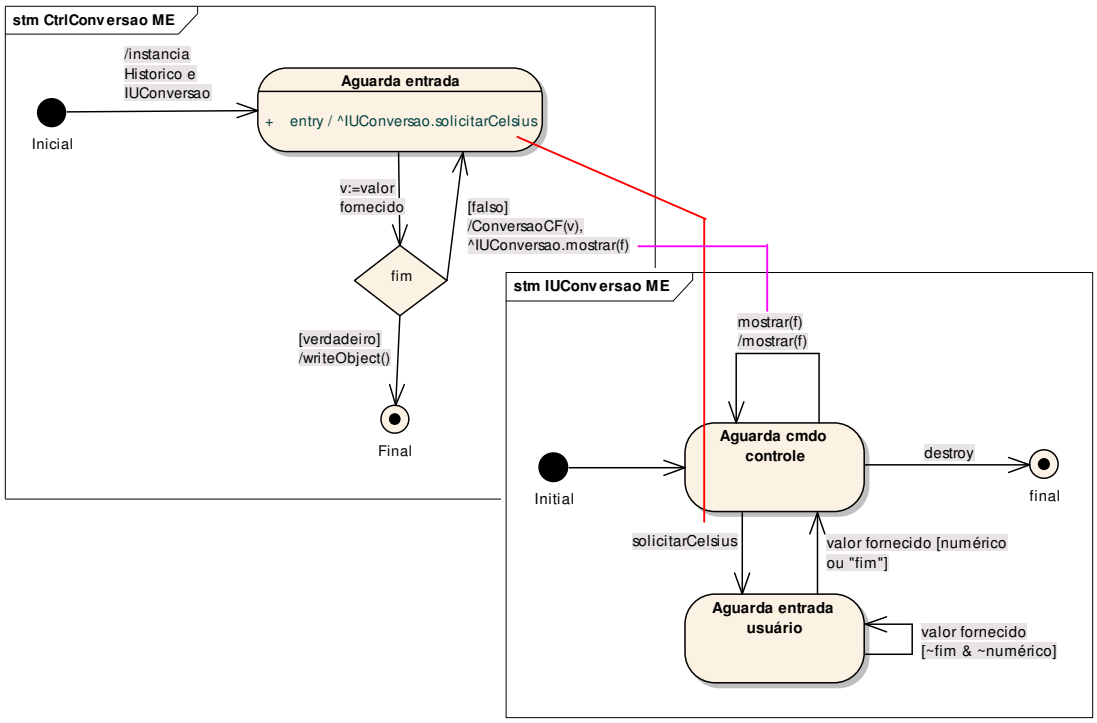
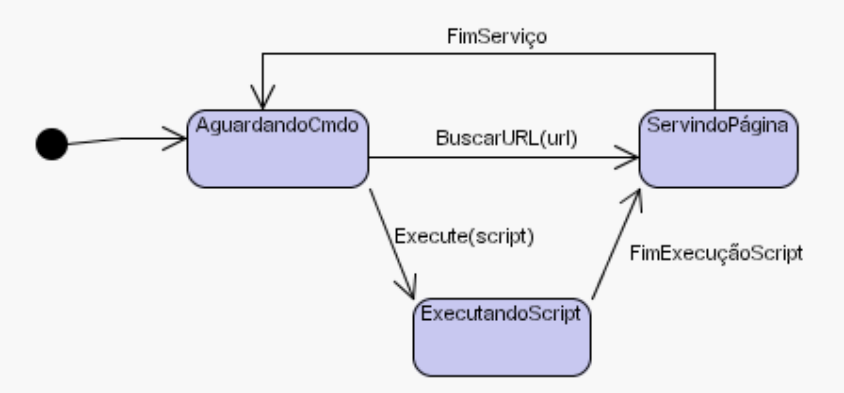


Figura 91: diagrama de estados para a classe CtrlConversao e IUConversao.

Quando um objeto envia um evento de chamada a outro objeto ele passa o controle da execução ao receptor. Uma vez que o objeto receptor processa o evento, disparando uma transição (se houver), o controle retorna ao invocador. Porém, contrariamente a chamada de um método, uma máquina de estados que recebe um evento de chamada pode continuar sua execução em paralelo (após retorno do controle) com o objeto invocador.

2.2 De sinal

São eventos assíncronos que, portanto, não bloqueiam o emissor, tal como um sinal enviado pela rede de comunicação de um processo a outro ou vindo da própria interface do usuário. Para ilustrar este tipo de sinal, considerar um servidor de páginas WEB. O funcionamento típico é o seguinte: o servidor aguarda mensagens que podem chegar a qualquer momento. Quando chega uma mensagem, o servidor a trata e envia uma resposta de forma assíncrona (sem se preocupar se a mensagem chegou ou não).



2.3 Temporal

Tipicamente são utilizados eventos nomeados por *After(30seg*) ou *when(data= 1/ 2/2004)* para indicar, respectivamente, um intervalo de tempo relativo ou um momento preciso no tempo. Por exemplo, um semáforo passa pelos estados vermelho, verde e amarelo, sendo 45s no vermelho, 45s no verde e 5s no amarelo.

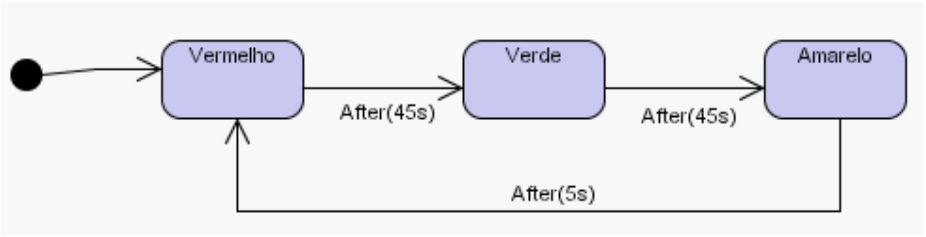


Figura 92: Exemplo de evento temporal em máquina de estados.

2.4 De mudança

É uma condição avaliada continuamente disparando um evento toda vez que se torna verdadeira. É representada frequentemente por meio do evento *When*. Difere de uma condição de guarda que é avaliada somente uma vez quando o evento ao qual está associada ocorre. A figura 93 ilustra a diferença entre um evento de mudança e uma condição de guarda:

- ◇ Parte superior da figura: "*encher o tanque* e *parar quando volume = 25 litros*"
- ◇ Parte inferior da figura: "*uma vez o tanque cheio, se o volume for maior ou igual a 25 ganha ducha, caso contrário, não ganha*".

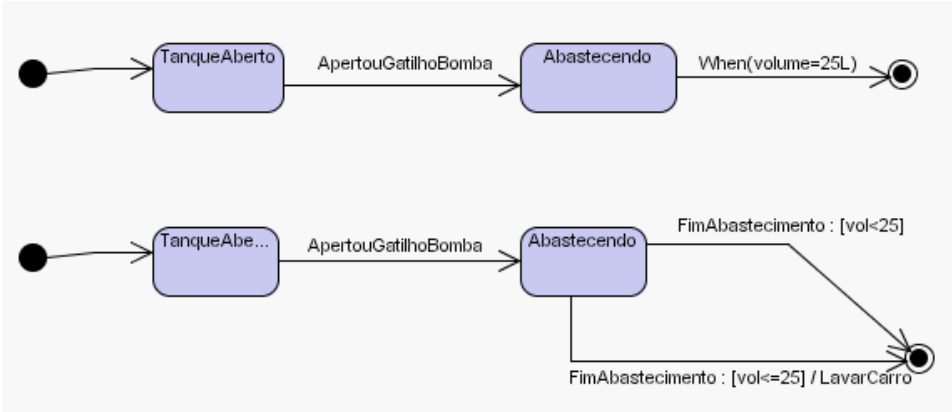
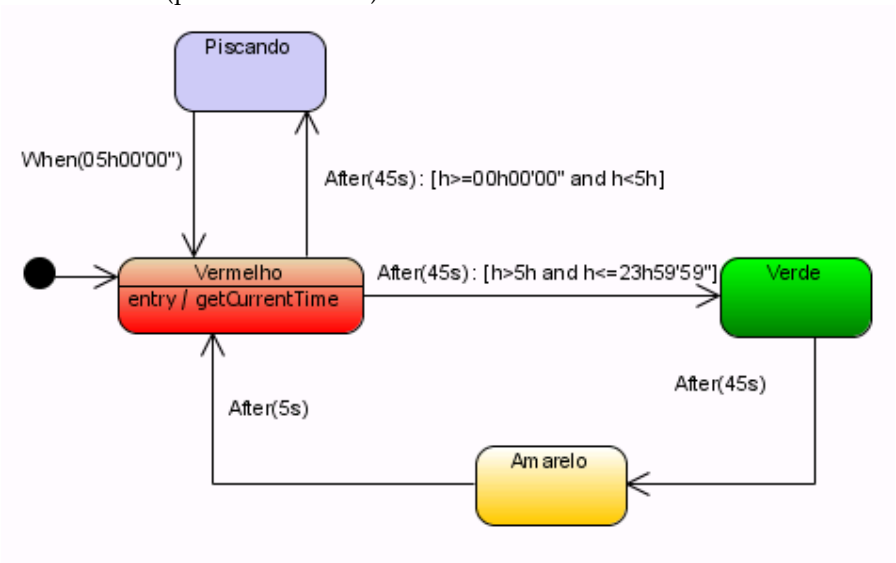


Figura 93: diferença entre condição de guarda e evento de mudança.

Exercício. Refaça o exemplo do semáforo para acomodar a seguinte situação: da meia-noite às 5h00 o sinal fica no estado alerta (piscando amarelo).



3 ESTADO COMPOSTO

Um estado composto pode ser decomposto em um conjunto de regiões, cada uma delas com vários subestados. Há diversas razões e, por conseqüência, formas ligeiramente diferentes de se representar estados compostos.

- ◇ Para representar decomposição de um estado, pode-se fazer um estado composto com uma única região com subestados diretos onde somente um deles está ativo por vez.
- ◇ Para representar concorrência entre estados, isto é, o fato de dois estados estarem ativos ao mesmo tempo, é preciso representar dividir o estado composto em duas ou mais regiões cada qual terá um estado ativo.
- ◇ Para representar uma submáquina, ou seja, um estado que referencia outra máquina de estados que conceitualmente substitui o estado em questão.

Para exemplificar a situação, imaginar um aparelho de fax capaz de enviar e transmitir simultaneamente. A representação, idêntica à utilizada no caso de *multithreading*, é ilustrada na figura 94. O estado composto Processando é dividido em duas regiões concorrentes, a superior, responsável pela recepção de fax, e a inferior, pelo envio. Em um dado momento, o aparelho de fax pode estar com os subestados *transmitindo* e *recebendo* ativos.

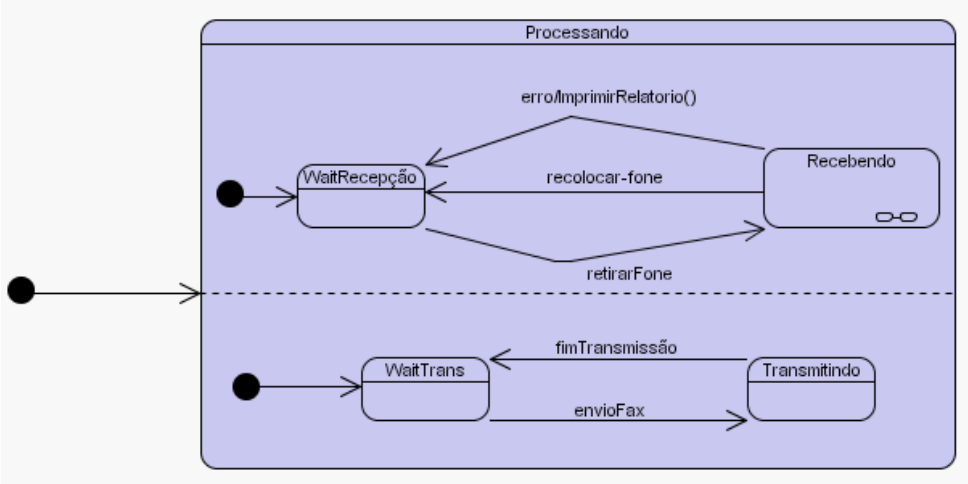


Figura 94: Estado composto Processando com duas regiões concorrentes.

Na figura 94, é possível observar um pequeno símbolo na parte inferior do subestado Recebendo que indica ser um estado composto. Sua decomposição está detalhada na figura 95.

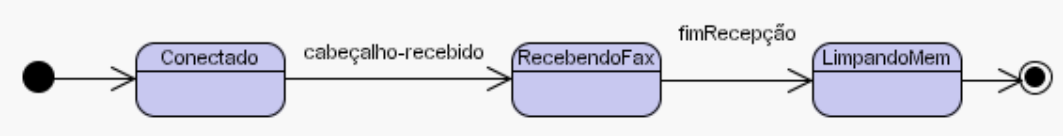


Figura 95: Detalhamento do estado *Recebendo*.

3.1 Histórico

O pseudo-estado histórico denotado por **H** é utilizado para memorizar o último estado ativo quando se deixou um estado composto. A **flecha do H** aponta para o estado *default*, ou seja, o subestado que é ativado na primeira vez em que o estado composto é alcançado. Por exemplo, a Figura 96 mostra um lava-car que inicia no estado composto, subestado lavagem. Caso ocorra o evento *parada de urgência* durante o estado *enxaguar*, retorna-se ao estado *enxaguar* após a ocorrência do evento retomada graças ao símbolo histórico.

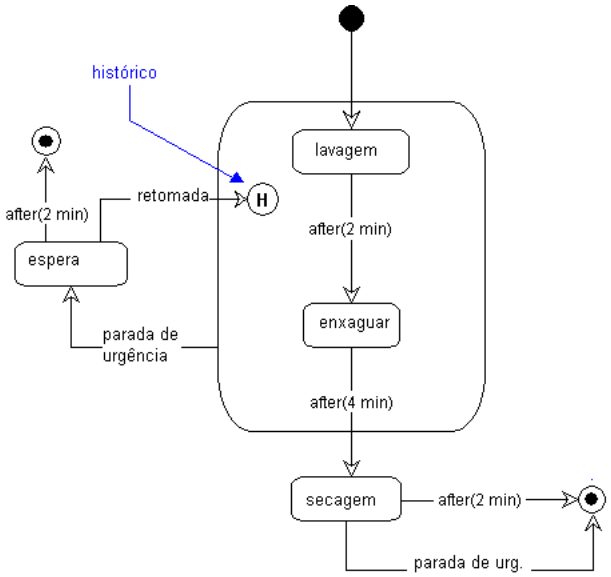


Figura 96: Memorização do último estado visitado (histórico).

4 EXERCÍCIOS

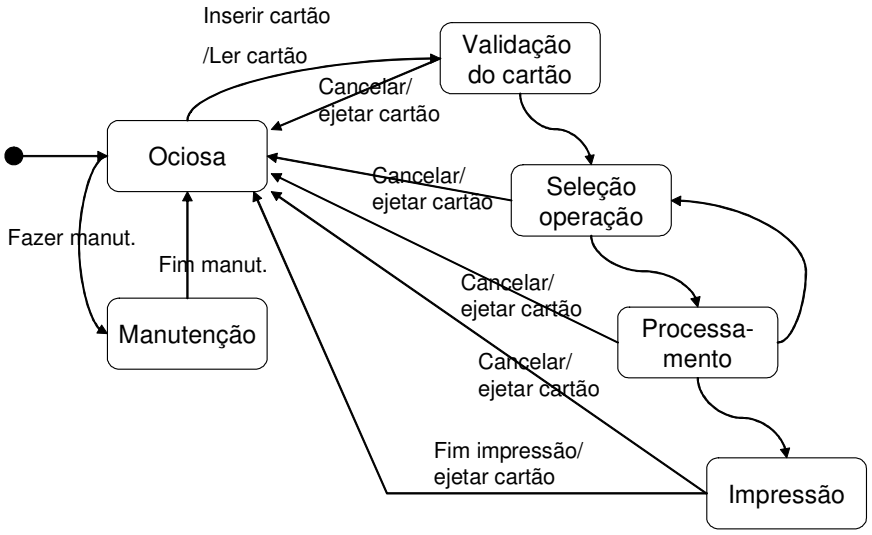
1. Fazer um diagrama de estados para as estações do ano. Fazer o diagrama de estados no editor UML e codificar a classe em JAVA.
2. Análise o código abaixo (CD Player) e construa a máquina de estados equivalente ao comportamento apresentado. Utilize condições de guarda e ações nas transições.

```
1    import java.io.*;
2 class CDPlayer {
3    static private int estado=0;
4
5    // AÇÕES
6    static private void desligar() {
7         System.out.println("*cortar fonte de energia*");
8    }
9    static private void ligar() {
10           System.out.println("*alimentação acionada*");
11      }
12      static private void pararCDabrirCompartimento() {
13         pararCD();
14         abrirCompartimento();
15      }
16      static private void fecharCompartimentoTocarCD() {
17         fecharCompartimento();
18         tocarCD();
19      }
20      static private void fecharCompartimento() {
21         System.out.println("*acionar motor fechamento*");
22      }
23      static private void abrirCompartimento() {
24         System.out.println("*acionar motor abertura*");
25      }
26      static private void tocarCD() {
27         System.out.println("*acionar leitor otico e giro CD*");
28      }
29      static private void pararCD() {
30         System.out.println("*parar leitor otico e giro CD*");
31      }
32
33
34      static private void mudarEstado(String evento) {
35         switch(estado) {
```

```
36            case 0: // estado desligado
37               if (evento.compareToIgnoreCase("l")==0) {
38                   estado=1;
39                   ligar();
40                   System.out.println("[CD LIGADO]");
41               }
42               break;
43            case 1: // estado ligado
44               if (evento.compareToIgnoreCase("l")==0) {
45                   estado=0;
46                   desligar();
47                   System.out.println("[CD DESLIGADO]");
48               }
49               else
50               if (evento.compareToIgnoreCase("p")==0) {
51                  estado=2;
52                  tocarCD();
53                  System.out.println("[TOCANDO]");
54               }
55               else
56               if (evento.compareToIgnoreCase("o")==0) {
57                  estado=3;
58                  abrirCompartimento();
59                  System.out.println("[ABERTO]");
60               }
61               break;
62
63            case 2: // tocando
64               if (evento.compareToIgnoreCase("l")==0) {
65                   estado=0;
66                   pararCD();
67                   desligar();
68                   System.out.println("[CD DESLIGADO]");
69               }
70               else
71               if (evento.compareToIgnoreCase("o")==0) {
72                  estado=3;
73                  pararCDabrirCompartimento();
74                  System.out.println("[ABERTO]");
75               }
76               if (evento.compareToIgnoreCase("s")==0) {
77                  estado=1;
78                  pararCD();
79                  System.out.println("[CD LIGADO]");
80               }
81               break;
82
83            case 3: // aberto
84               if (evento.compareToIgnoreCase("l")==0) {
85                   estado=0;
86                   fecharCompartimento();
87                   System.out.println("[CD DESLIGADO]");
88               }
89               else
90               if (evento.compareToIgnoreCase("p")==0) {
91                  estado=2;
92                  fecharCompartimentoTocarCD();
93                  System.out.println("[TOCANDO]");
94               }
95               else
96               if (evento.compareToIgnoreCase("c")==0) {
97                  estado=1;
98                  fecharCompartimento();
99                  System.out.println("[LIGADO]");
100              }
101              break;
102
103           }
104     }
105
106     static public void main(String args[]) {
107        BufferedReader rdr = new BufferedReader(new InputStreamReader(System.in));
108        String evento="";
109        System.out.println("[CD DESLIGADO]");
110        do {
111           System.out.println("(L)iga/desliga (P)lay (S)top (O)pen (C)lose (F)im");
112           try {
```

```
113              evento = rdr.readLine();
114              mudarEstado(evento);
115           } catch (IOException e) {
116              System.out.println("!!! Erro leitura !!!");
117           }
118        } while (evento.compareToIgnoreCase("f")!=0);
119     }
120  }
```

3. Refazer o exercício acima mostrando como colocar ações nos estados (*entry* e *exit*).
4. Refazer o diagrama de estados abaixo utilizando um estado composto.



5. Faça um diagrama de estados para uma classe que implemente um jogo de xadrez. Você pode usar, por exemplo, os eventos seguintes: branca move, preta move, branca desiste, preta desiste, xeque mate. Represente nos estados o jogador da vez (brancas ou pretas).
6. Faça o diagrama de estados para um despertador. O despertador pode estar em um dos estados seguintes: desarmado, esperando e despertando. O despertador inicia no estado desarmado. Para passar ao estado esperando, ele deve ser armado para disparar num determinado horário. No estado despertando ele soa por 30 segundos. Se o usuário desligá-lo ele volta ao estado desarmado. Caso o usuário não desligue, o despertador volta a soar em 2 minutos até 3 vezes. Se ao cabo destas 3 vezes o usuário não desligou-o então o despertador volta ao estado desarmado.
7. Desenhe o diagrama de estados correspondente ao algoritmo do fatorial de "n":
   0! = 1
   1! = 1
   n! se n > 1 = n*(n-1)!
8. Represente em 3 diagramas de estado, **uma televisão** que pode estar ligada ou desligada, um **DVD player**, que também pode estar ligado ou desligado e **um controle** remoto que tem dois modos de funcionamento ora liga/desliga a televisão e ora liga/desliga o DVD player. Os botões dos três aparelhos são do tipo liga/desliga (o mesmo botão realiza as duas funções).
9. Suponha um editor de textos capaz de manipular três tipos diferentes de objetos: textos, imagens e desenhos. Este programa manipula somente um tipo de objeto por vez de acordo com a seleção corrente do usuário. Quando o usuário seleciona um texto, o editor mostra um menu pop-up com as opções de edição textuais, o mesmo ocorre quando da seleção de uma imagem ou de um

desenho. Faça o diagrama de estados representado este comportamento e o pseudocódigo que implementa o diagrama de estados

IX DIAGRAMA DE ATIVIDADES

O diagrama de atividades é utilizado para representar a **dinâmica** do negócio, dos casos de uso ou para detalhar uma operação de uma classe. Em relação aos casos de uso, pode tanto representar o fluxo básico como um cenário particular de execução. É importante ressaltar que diagramas de atividade não devem tomar o lugar dos diagramas de estado nem dos diagramas de interação, pois ele não detalha como os objetos se comportam nem como interagem.

Normalmente é utilizado na fase de inicialização para descrever os fluxos dos casos de uso, amadurecendo na fase de elaboração. Posteriormente pode ser utilizado para representar o fluxograma para uma operação.

1 ELEMENTOS BÁSICOS

- ◇ Ação: realizada de forma instantânea.
- ◇ Atividade: demora certo tempo para ser realizada.
- ◇ Transição:
- ◇ Decisão
- ◇ Paralelismo ou bifurcação (fork)
- ◇ Sincronização ou união (join)
- ◇ Raias

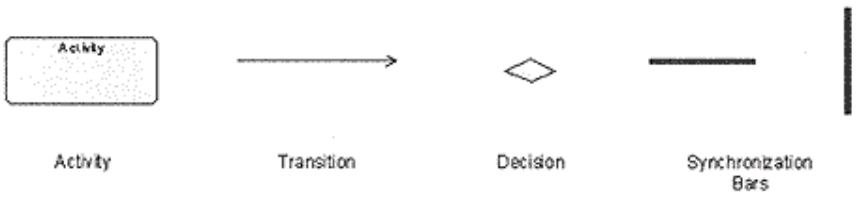


Figura 97: Elementos básicos do diagrama de atividades.

A figura 98, ilustra um caso de uso de oferta de disciplinas em uma universidade. Professores escolhem as disciplinas e horários (que formam as turmas), os alunos recebem uma notificação e também uma cópia do catálogo é impressa. As seguintes atividades estão representadas:

- ◇ Criar turmas (entrar com as informações das turmas para o semestre)
- ◇ Escolher turmas (professor escolhe turmas)
- ◇ Associar professor às turmas
- ◇ Criar catálogo de disciplinas (na verdade das turmas)

As transições entre as atividades representam o fluxo de controle do caso de uso. Há atividades seqüenciais, pontos de decisão, barras de paralelismo e de sincronização, raias que identificam os responsáveis pelas atividades e atividades inicial e final.

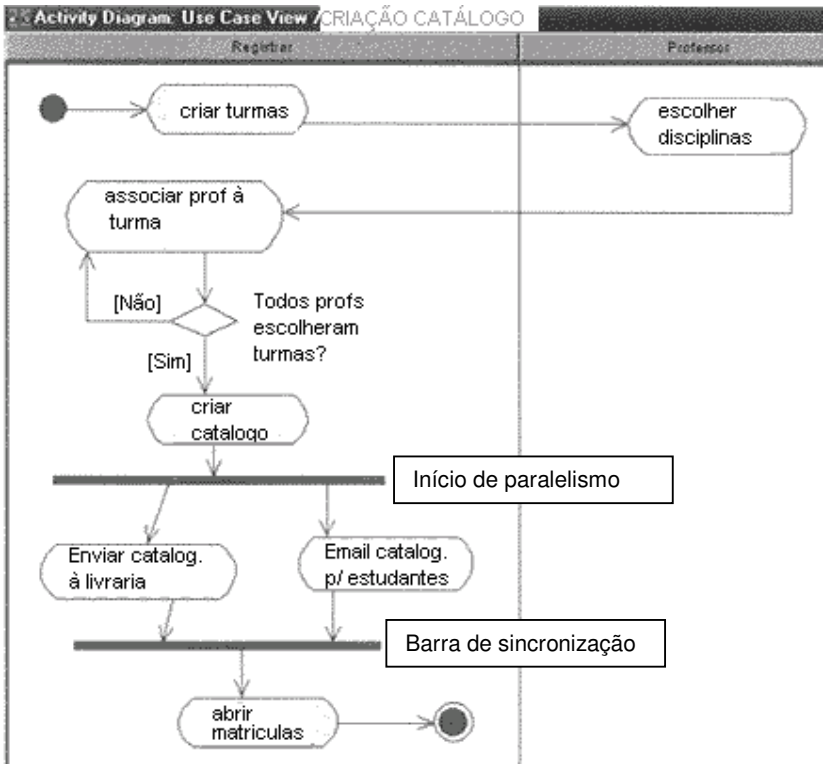


Figura 98: Exemplo de diagrama de atividades.

2 EXERCÍCIOS

Representar por meio de um diagrama de atividades a regra de negócio de um sistema de controle de notas escolares que determina se um aluno foi aprovado ou reprovado em função de:

- O aluno de vê ter freqüência igual ou maior a 75%, caso contrário estará reprovado por faltas.
- Em relação à média das notas parciais n1 e n2:
- se superior ou igual a 7,0, aprovado por média
- se superior ou igual 5, 0 e inferior a 7,0, em final
- se inferior a 5,0, reprovado por nota.
- Se aluno em final, realiza uma terceira prova:
- Se média + final / 2 for maior ou igual a 5,0, aprovado, caso contrário reprovado por nota.

X DIAGRAMA DE COMPONENTES E IMPLANTAÇÃO

1 DIAGRAMA DE COMPONENTES

Quando um classificador (classe, pacote, subsistema, componente) realiza uma interface, significa que ele implementa uma ou mais operações especificadas pela mesma. Uma interface define, portanto, uma coleção de serviços que devem ser implementados em algum ponto do sistema.

Um componente é uma parte física do sistema, como um arquivo executável, que realiza uma interface e é, portanto, substituível por outro componente que realize a mesma interface. A figura 99 ilustra um *componente.java* que realiza a interface *ImageObserver* e um outro, *image.java*, que depende da interface (e não da implementação da mesma).

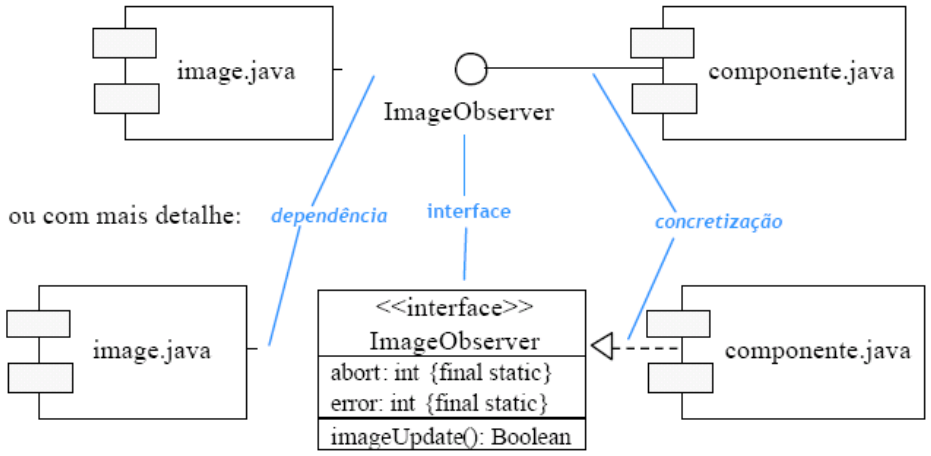


Figura 99: Exemplo de componentes e de suas relações[3].

A UML define três tipos de componentes (Bezerra, 2004):

- **De execução**: existem em tempo de execução, por exemplo, processos, threads, etc.
- **De instalação**: por exemplo, arquivos executáveis, controles Active-X, DLLs, etc.
- **De trabalho**: são aqueles a partir dos quais os componentes de instalação são criados, tais como documentos, fontes, bibliotecas estáticas, etc.

Há vários estereótipos para componentes:

- Executável
- Biblioteca: bibliotecas de classes ou funções;
- Tabela: repositório de dados;
- Documento: arquivos texto (help), manual do usuário;
- Arquivo: código fonte ou outro arquivo qualquer.

[3] Extraído de http://paginas.fe.up.pt/~jpf/teach/POO/componentes.pdf

2 DIAGRAMA DE IMPLANTAÇÃO

O diagrama de implantação (*deployement*) representa a arquitetura física do sistema e, opcionalmente, os componentes que existem em tempo de execução. Os elementos básicos são:

- **Nós**: representa um recurso computacional (processadores, dispositivos, sensores, roteadores);
- **Conexões**: ligam os nós, normalmente são meios físicos de comunicação (fibra ótica, cabo coaxial) ou protocolos de comunicação (TCP/IP, HTTP, UDP).

A figura 100 mostra um exemplo de diagrama de implantação.

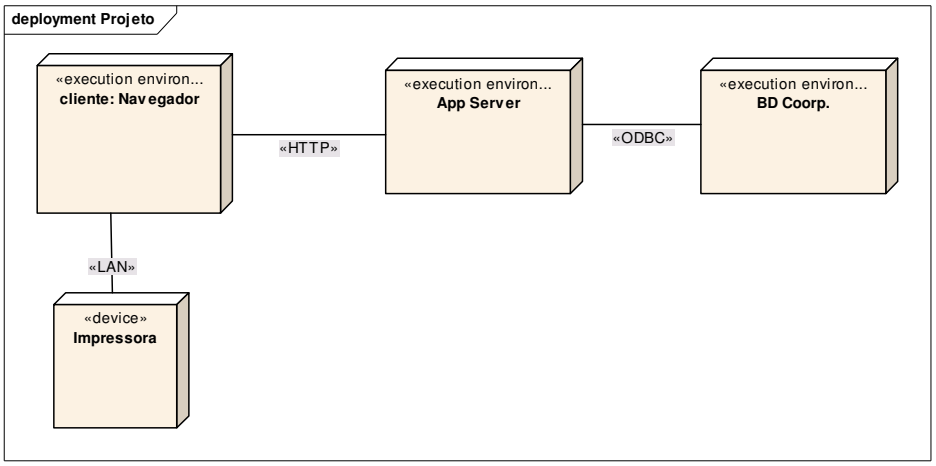


Figura 100: Exemplo de diagrama de implantação.

O diagrama de implantação pode ser associado ao de componentes para representar a distribuição dos componentes. Pode ser importante para analisar o sistema quanto à distribuição de carga de trabalho. O diagrama de componentes da figura 101 pode ser associado ao diagrama de implantação da figura 100 resultando naquele ilustrado pela figura 101.

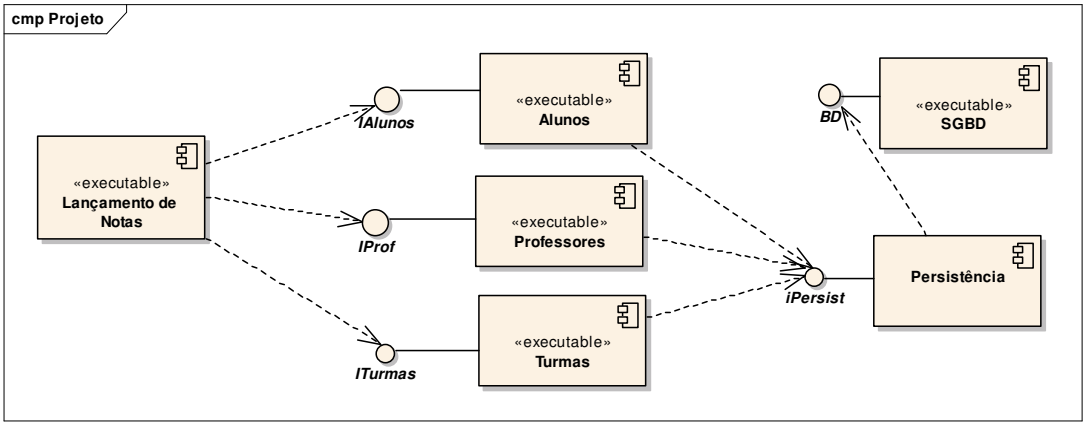


Figura 101 Exemplo de diagrama de componentes.

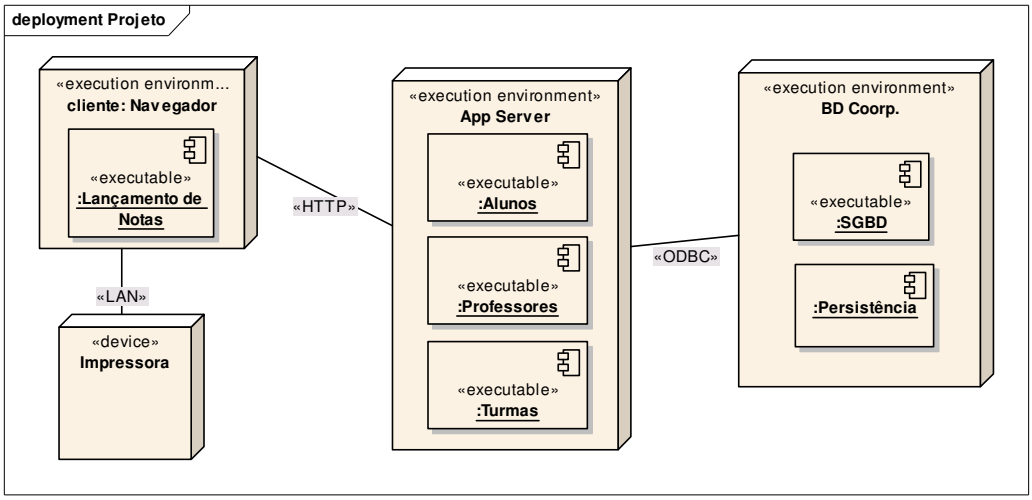


Figura 102 Exemplo de diagrama de componentes.

XI REFERÊNCIAS BIBLIOGRÁFICAS

BOOCH G; RUMBAUGH J; JACOBSON I. **UML: guia do usuário**. Rio de Janeiro: Campus, c2000. 472p. ISBN 85-352-0562-4

BEZERRA E. **Princípios de Análise e Projeto de Sistemas com UML**, Rio de Janeiro, Elsevier, 2002, 286p, ISBN 85-352-1032-6.

FOWLER M; SCOTT K. **UML essencial: um breve guia para a linguagem-padrão de modelagem de objetos**. 2.ed. Porto Alegre: Bookman, 2000 169 p. ISBN 8573077298

FOWLER M. Padrões de Arquitetura de Aplicações Corporativas, Bookman (ed.), 494p.

GUEDES G. T. A.; **UML Uma Abordagem Prática**, Inovatec, 319p., 2004.

OMG, **Unified Modeling Language 2.1.1**, 732p., Fevereiro/2007. Disponível em http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf, acesso em 16/04/2007.

PRESSMAN R. S. **Engenharia de software.** São Paulo: Makron, 1995. 1056 p. ISBN 85-346-0237-9

RUMBAUGH J., JACOBSON I., BOOCH G. **The Unified Modeling Language Reference Manual**, 2ed., Pearson Education, 2005, ISBN 0-321-24562-8.

QUATRANI T. **Visual Modelling with Rational Rose 2000**, Addison-Wesley, 2ª edição, 1999, 288p.