Treinamento IBTA

Formação Java

Introdução a Orientação a Objetos

Módulo I

IJ 100

Versão 1.0 Junho 2005

Índice

Iniciando

Objetivos

Este curso tem como objetivo introduzir o aluno nos conceitos da Orientação a Objetos.

Os principais tópicos são: os princípios da orientação a objetos e a aplicação desses conceitos para escrever pequenas rotinas com a tecnologia Java abrangendo as funções essenciais da programação.

No final do curso, os alunos serão capazes de entender as bases da orientação a objetos e construir pequenos programas em Java.

Habilidades Adquiridas

Após a conclusão deste curso, os alunos deverão estar aptos a:

- Analisar um projeto de programação utilizando a análise orientada a objetos e fornecer um conjunto de classes, atributos e operações;
- Entender o processo básico de abstração de software e análise de problemas;
- Definir classes e sua responsabilidade;
- Entender os benefícios do encapsulamento, polimorfismo e herança;

1. Fundamentos do desenvolvimento de *software* orientado a objetos

Depois de muitos esforços no campo do desenvolvimento de software, a comunidade de tecnologia desenvolveu uma técnica muito interessante, simples e mais natural do ponto de vista humano para a realização do processo de análise de problemas e construção de aplicações baseadas em software, essa técnica é chamada de Orientação a Objetos.

Como as técnicas de desenvolvimento de software evoluíram muito nos últimos dez anos a comunidade de tecnologia seguiu essa evolução, migrando do modelo procedural ou da análise essencial para o modelo Orientado a Objetos visando ter uma ferramenta mais objetiva e simples para se construir software e aplicações.

1.1. O Paradigma Procedural

O **modelo procedural**, utilizado até os tempos atuais tem como base a execução de rotinas ou funções, seqüenciadas e ordenadas para atender os requisitos funcionais de uma aplicação.

Examinando a figura abaixo, vemos que neste modelo as funções e os dados estão em áreas de memória e estruturas computacionais diferentes, as funções ficam em programas e os dados em bases de dados, fazendo com que o desenvolvedor se esforce muito para unir os dados e as funções de um elemento do sistema, tornando os sistemas muito complexos.

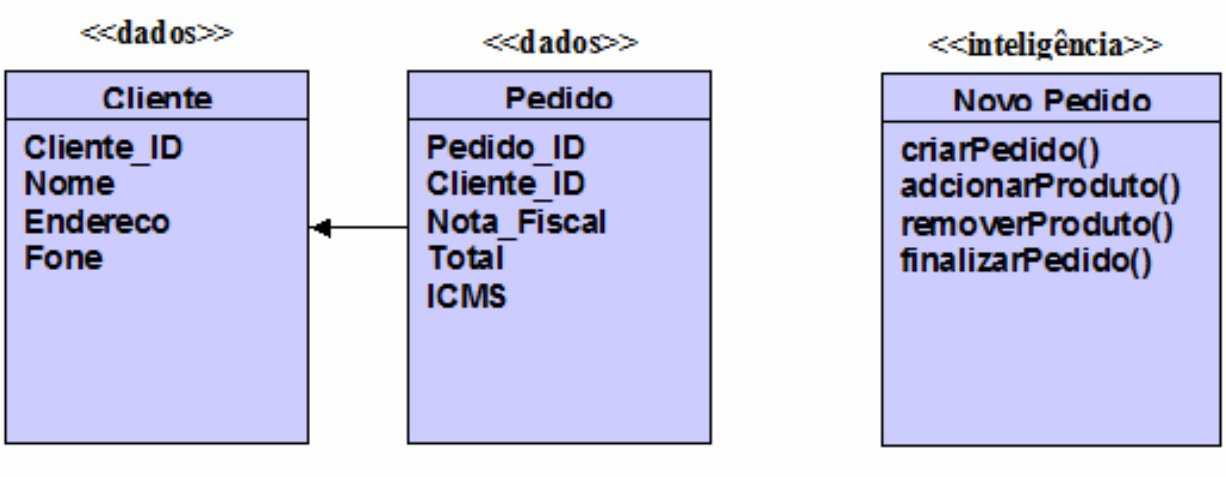


Figura 1

O Paradigma Orientado a Objetos

O **modelo objeto** tem como base a execução de métodos (pequenas funções que atuam diretamente sobre os dados de um objeto), levando em consideração como o usuário enxerga o sistema e suas funções.

Examinando a figura abaixo vemos que neste modelo as funções e os dados estão na mesma área de memória, num mesmo local, tornando mais fácil e menos complexo a construção dos sistemas.

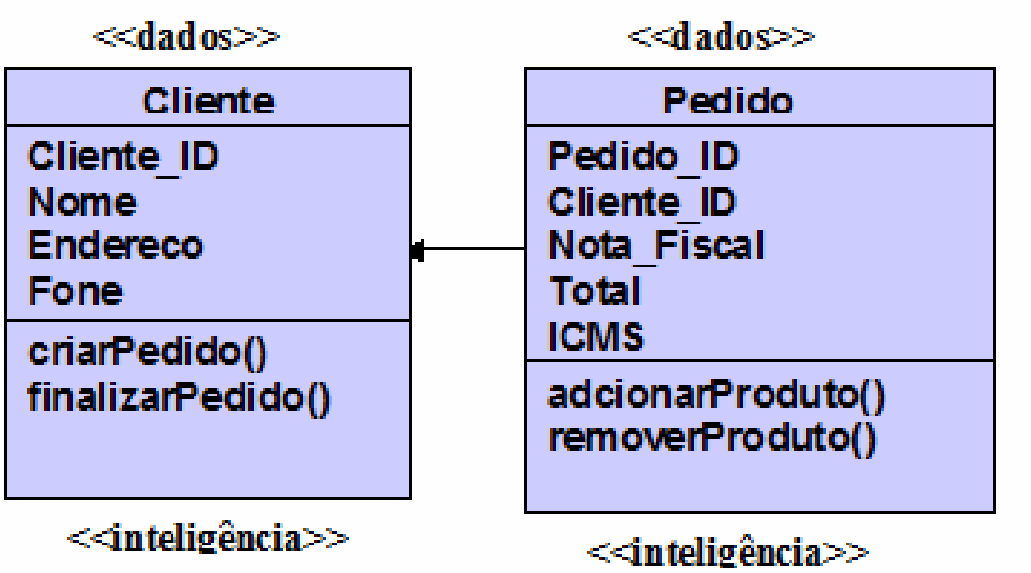


Figura 2

Por esse motivo, a Orientação a Objetos tem sido adotada cada vez mais como tecnologia eficiente para o desenvolvimento de sistemas computacionais.

Atualmente tanto comercialmente como academicamente temos várias linguagens que permitem a aplicação total das técnicas da Orientação a Objetos, entre elas temos: ***Java***, C++, Ada95 (linguagem usada nos projetos da Nasa), SmallTalk, Eifel, Microsoft .NET (C#, VB.NET, J++, etc) e etc.

O modelo orientado a objeto por ser mais simples e lingüisticamente natural, permite a redução de custos e um melhor entendimento do que um sistema computacional é, e o que ele deve fazer.

Ao contrário do que se pensa, o Java é uma evolução vinda da derivação do modelo de linguagem do SmallTalk e dos mecanismos computacionais do C++. Sendo mais simples e mais robusta que ambas.

Nos tempos atuais, para se garantir a qualidade do desenvolvimento, manutenção e evolução de sistemas computacionais, devemos usar uma metodologia aderente à técnica da Orientação á Objetos.

Atualmente podemos usar a Programação Orientada a Objetos nas linguagens que suportam a Orientação a Objetos.

Discussão

O que se entende por Paradigmas?

Que outras linguagens podem ser analogicamente avaliadas e categorizadas como Procedurais ou Orientadas a Objetos?

Para que servem sistemas ou aplicações nas empresas, no governou ou em áreas específicas da sociedade?

Por que mudar do modelo Procedural para o modelo Orientado a Objetos? Existem benefícios? Quais as dificuldades?

Para que serve a análise de sistemas?

2. Conceitos da Programação Orientada a Objetos

Como estamos falando de Programação Orientada a Objetos, devemos antes de começar a programar em Java entender as técnicas e conceitos da Orientação a Objetos.

Quando queremos programar em Java, devemos definir separadamente o que um sistema é (sua estrutura) e o que o sistema faz (suas funcionalidades).

Para podermos criar um sistema orientado a objetos e para que este sistema possa ser convertido em software Java, é necessário desenvolvermos e exercitar nossa capacidade na técnica da **abstração**.

Ao entender e **abstrair** os elementos de um sistema pode-se criar graficamente e descrever textualmente suas funcionalidades e estados num modelo ou protótipo.

Para isso veremos a seguir as etapas deste processo.

2.1. Abstraindo e descrevendo um problema de negócio

O processo de abstração está baseado na capacidade de entender um sistema real, por exemplo, um jogo de futebol, ou imaginário, por exemplo, um sistema de contas corrente, e seguindo padrões internacionais criar um modelo documentado que reflita os seus comportamentos (estrutura e funcionalidades).
I went to hight school in São Paulo, I wanted to be accountant but I don't know why I started matematic university for two years

Para isso usamos o processo de ***abstração***.

Abstração: é processo pelo qual modelamos sistemas reais ou imaginários, extraindo do seu contexto os seus principais elementos "objetos", e descrevemos cada objeto, com suas principais "características" de dados (estruturais) e funcionalidades (comportamentos).

2.2. Uma idéia de abstração

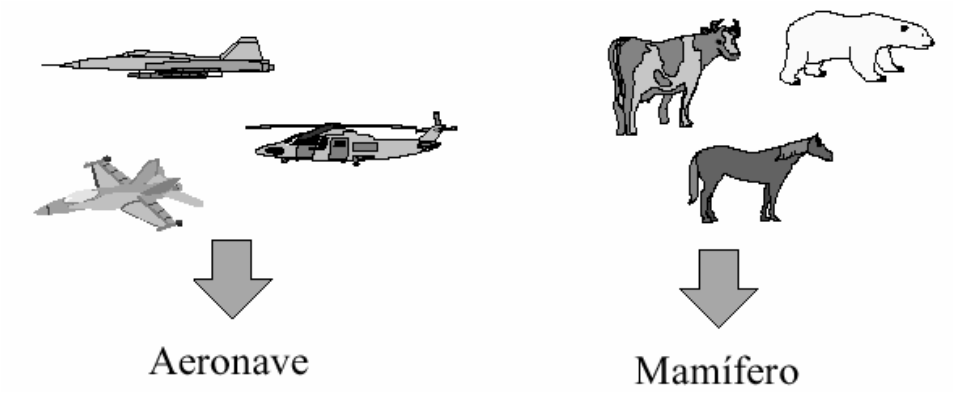


Figura 3

As características dos objetos são divididas em duas categorias:

- **Propriedades** (**atributos**): descrevem o que o objeto de fato é, o que ele representa no sistema, e os seus estados;

- **Funcionalidades/Comportamentos** (**métodos**): descrevem as operações que um objeto realiza, e representa o seu comportamento dentro do sistema;

2.3. Objetos e Classes

O **objeto** é um conceito mental e para podermos definir um tipo de objeto dentro de um modelo de software, devemos criar uma **classe** que o represente tanto em propriedades como em funcionalidades.

Usamos as **classes** para definir, dentro de um modelo de software, nossos tipos de objetos. Uma classe define um determinado tipo (conjunto de objetos), abstraído de um modelo.

Podemos analogicamente dizer que uma classe representa e define uma categoria de objetos.

2.4. Entendendo o processo de abstração

Imaginando uma camisa, daquelas que temos em nossos armários, dentro do mundo real, poderíamos **abstrair** algumas propriedades, **atributos**, tais como:

Atributos:
tamanho, cor,
código, tecido,
etc.

Figura 4

No processo de **abstração de dados ou estrutural**, enumeramos as propriedades do objeto seguindo as necessidades do sistema em ele está inserido.

No processo de **abstração funcional ou comportamental**, enumeramos as funcionalidades do objeto seguindo as necessidades do sistema em ele está inserido.

Imaginando uma Conta, daquelas que temos em bancos, dentro do mundo imaginário, poderíamos **abstrair facilmente** algumas propriedades, **atributos** e **funcionalidades** tais como:

Atributos: saldo, limite, juros, vencimento, etc.

Funcionalidades: débito, crédito, consultar saldo, etc.

Figura 5

2.5. Definindo o modelo de domínio de um sistema

O modelo de domínio de um sistema representado por objetos, é descrito textualmente e representado graficamente por objetos, como eles se comunicam entre si e as funcionalidades executadas por eles.

Após a construção do modelo de domínio, poderemos classificar os objetos, com todos os seus atributos e comportamentos, que serão convertidos em software, vale lembrar que o componente de software que representa um objeto é a sua classe.

Para criarmos um modelo de domínio, devemos antes identificar um problema, e para efeitos didáticos, usaremos o seguinte:

2.6. Descrição textual de um problema de um sistema.

"Um pequeno restaurante, necessita de um aplicativo que controle as contas de gastos de seus clientes, que será operado pelo caixa do restaurante. O cliente é identificado pelo nome e sua conta por um número.

Cada cliente faz um "depósito inicial" que lhe dará crédito para consumir os produtos e serviços do restaurante.

A compra só é efetuada se houver crédito suficiente na conta do cliente para o débito em questão.

O cliente só poderá fazer um novo depósito ou encerrar a conta quando seu crédito for inferior a R$ 20,00.

A conta não pode ficar com saldo negativo.

O cliente pode consultar seu saldo num terminal de consultas."

2.7. Representação gráfica do modelo de domínio

Usamos a forma abaixo para representar um modelo de domínio, ou seja, quais objetos existem num sistema e como ele se relacionam entre si.

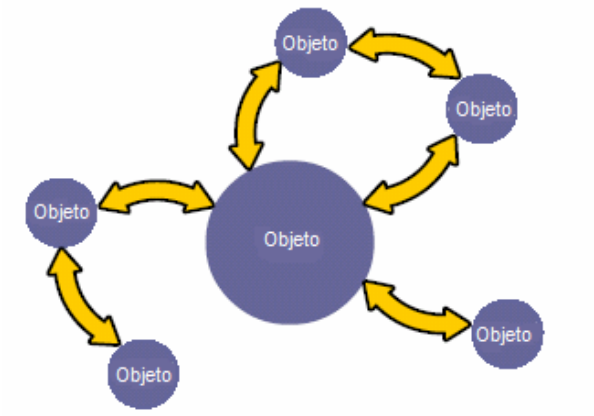


Figura 6

2.8. O que é um objeto?

Um **objeto** é um elemento computacional que representa, no domínio da solução, alguma entidade (abstrata ou concreta) de interesse do sistema sob análise. Objetos similares são agrupados em classes.

No paradigma da orientação a objetos, tudo pode ser potencialmente representado como um objeto. Sob o ponto de vista da programação, um objeto não é muito diferente de uma variável no paradigma de programação convencional.

Por exemplo, quando se define uma variável em C ou em Java, essa variável terá:

- Um espaço em memória para registrar o seu estado atual (um valor);
- Um conjunto de operações associadas que podem ser aplicadas a ela, através dos operadores definidos na linguagem que podem ser aplicados a valores inteiros (soma, subtração, inversão de sinal, multiplicação, divisão inteira, resto da divisão inteira, incremento, decremento).

Da mesma forma, quando se cria um objeto, esse objeto adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos, definidos pela classe) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).

Um **programa orientado a objetos** é composto por um conjunto de objetos que interagem através de "trocas de mensagens" entre eles.

Na prática, essa troca de mensagem traduz-se na chamada de métodos (funções) entre objetos.

2.9. O que são classes?

Definimos uma classe quando desejamos representar num modelo um conjunto de um tipo de objetos.

A partir de um conjunto de objetos em comum, definimos uma classe. A partir dessa classe podemos criar objetos de um determinado tipo.

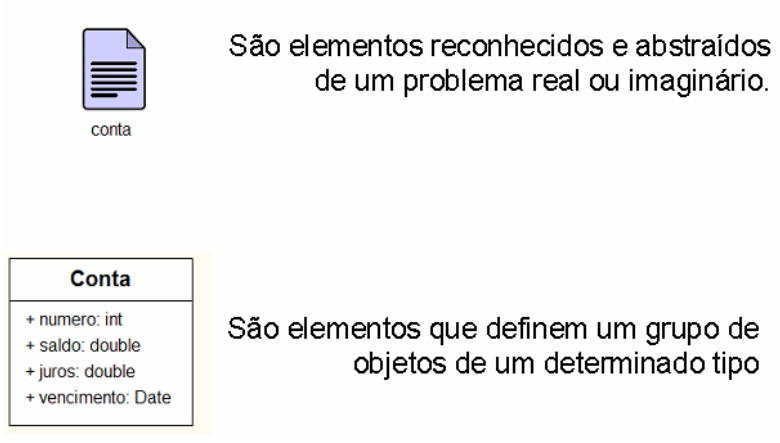


Figura 7

2.10. Identificando os objetos de um sistema usando os processos de abstração:

A técnica de identificação de objetos (OMT - Object Method Technique) é bem simples:

- Os substantivos que realizam ações são objetos;
- Os adjetivos ou substantivos que representam qualidades ou situações de estados dos objetos são atributos;
- Todas as ações verbais são funcionalidades;

Para facilitar a identificação, vamos separar as palavras que representam os candidatos a objetos, atributos e funcionalidades:

Objetos	Propriedades	Funcionalidades
restaurante sistema conta cliente caixa produto serviço terminal	saldo da conta nome do cliente número da conta	debitar creditar consultar saldo comprar abrir conta encerrar conta

Fazendo um estudo, podemos ver que existem alguns objetos, atributos ou métodos que serão irrelevantes na construção do sistema, pois representam sinônimos ou não tem relação com o problema que temos de resolver.

Algumas funcionalidades estão escondidas dentro da descrição do problema, e a capacidade de encontrá-las vai depender da experiência do profissional que executa esta tarefa.

Verificando os candidatos, ficamos então com:

Objetos	Propriedades	Funcionalidades
Conta cliente caixa produto	saldo da conta nome do cliente número da conta	debitar creditar consultar saldo comprar abrir conta novo cliente encerrar conta consultar produto

Podemos agora construir o modelo de domínio, que representa o sistema e a forma como os objetos se comunicam:

Representação gráfica de modelo de domínio de um sistema de objetos:

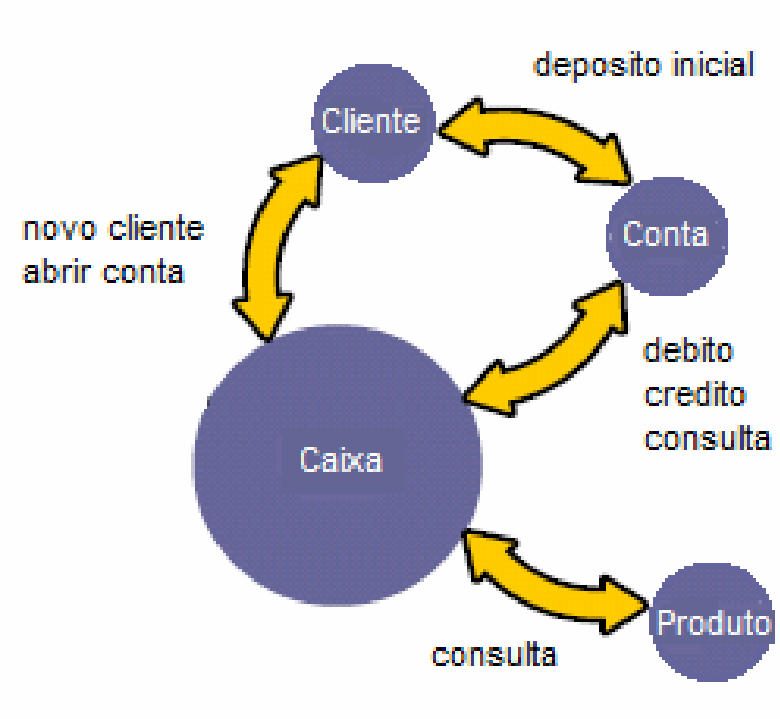


Figura 8

Visando garantir a qualidade do software, a técnica prevê o uso de uma notação universal e padronizada chamada de **UML** para facilitar a documentação construção e evolução do software como descrito anteriormente.

Discussão

Uma abstração coerente depende:

- Da experiência pessoal de quem faz?
- De um escopo de problema definido?

Podemos definir uma abstração como correta ou errada?

Qual é mais fácil de fazer, abstração funcional ou estrutural?

Qual a importância da descrição textual de um problema? O que ela nos fornece de informação?

Analisando a descrição textual de um problema e aplicando a técnica da OMT, podemos criar o modelo de domínio de qualquer sistema, num modelo orientado a objetos?

O que é uma instancia de um objeto?

3. Conceitos da Orientação a Objetos

Para facilitar o entendimento de como modelar sistemas Orientados a Objetos devemos entender sos seguintes conceitos: **Encapsulamento, Herança e Polimorfismo.**

São esses conceitos que nos permitem entender e projetar quaisquer sistemas, simples ou complexos, de uma forma mais natural.

3.1. Encapsulamento

Processo pelo qual escondemos detalhes de implementação fornecendo uma interface de comunicação simples, geralmente obtida no processo de abstração.

Assim, definimos os atributos (propriedades) dos objetos como "privados do objeto" (observe o sinal de - no diagrama da classe Pedido), e definimos um conjunto de funcionalidades (métodos) "públicas á todos outros objetos" (observe o sinal de **+** no diagrama da classe Pedido).

Praticamente em todas as linguagens comerciais existentes no mercado é possível se implementar o conceito do encapsulamento.

Exemplo descritivo:

> Durante a compra de um veículo, podemos “descrever” para o vendedor as características que gostaríamos que o mesmo tivesse (inclusive detalhes de motor), mas não é necessário informar detalhes de peças que compõem o motor do veículo, apesar de sabermos que sem tais peças, o motor não funcionaria.

Exemplo gráfico: Objeto de Pedido

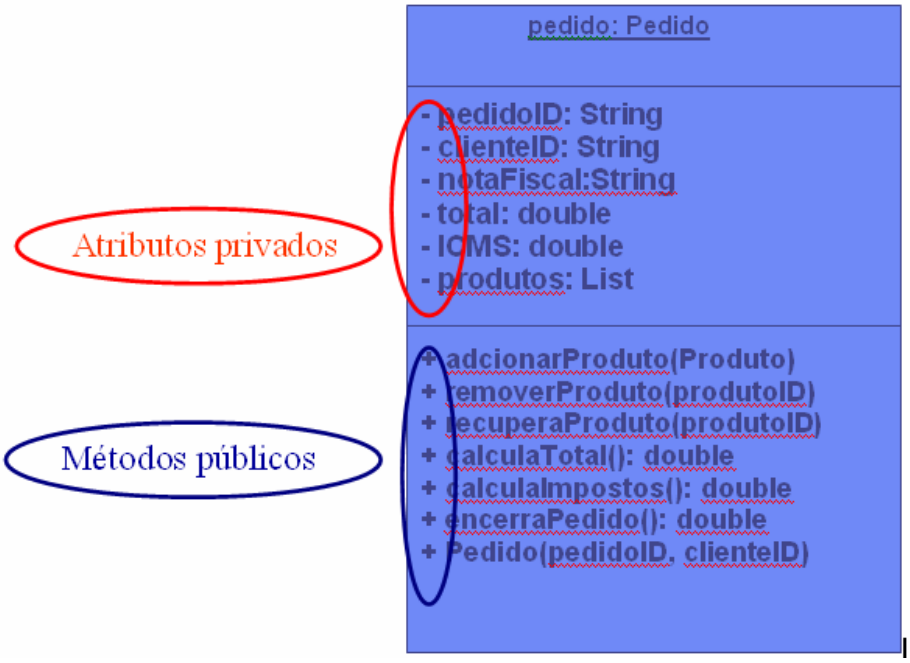


Figura 9

Dessa forma, um objeto "Cliente" que queira acessar os dados de um "Pedido", só tem acesso aos métodos públicos fornecidos, esses métodos foram definidos no processo de abstração.

Figura 10

3.2. Herança

O conceito de definir uma estrutura (atributos) e um conjunto de comportamento (métodos) em um tipo não é exclusivo da orientação a objetos; particularmente, a programação por tipos abstratos de dados segue esse mesmo conceito.

O que torna a orientação a objetos única é o conceito de **herança (no inglês: inheritance)**.

Herança é um mecanismo que permite que características comuns a diversas classes com comportamentos comuns ou parecidos (kind-of), sejam abstraídas e centralizadas em uma classe base, ou **superclasse**. Relação de “IS A” (do inglês: É UM).

A partir de uma **superclasse** outras classes podem ser especificadas. Cada classe derivada ou **subclasse herda** (recebe sem codificação extra) as características (atributos e métodos) da superclasse, e ainda podemos acrescentar a uma subclasse elementos particulares a ela. Os construtores são as únicas estruturas de uma classe que não são herdados.

Sendo uma linguagem de programação orientada a objetos, a linguagem Java oferece mecanismos para definir subclasses a partir de superclasses. Na linguagem Java uma subclasse somente pode herdar uma única superclasse diretamente.

É fundamental que se tenha uma boa compreensão sobre como objetos a partir de subclasses são criados e manipulados, assim como das restrições de acesso aos membros das superclasses.

Herança é sempre utilizada em Java, mesmo que não explicitamente. Quando uma classe é criada e não há nenhuma referência a uma superclasse, implicitamente o compilador Java insere na classe uma relação de herança direta com a superclasse ***`Object`***.

Existem dois processos que nos ajudam a definir onde atributos e métodos serão colocados dentro de um conjuntos de classes que estão relacionadas por herança, esse dois processos são: **Generalização** (Bottom-Up, de baixo para cima) e **Especialização** (Top-Down, de cima para baixo)

Generalização

Processo pelo qual identificamos atributos comuns em classes diferentes que são colocados numa superclasse, menos específica e mais genérica.

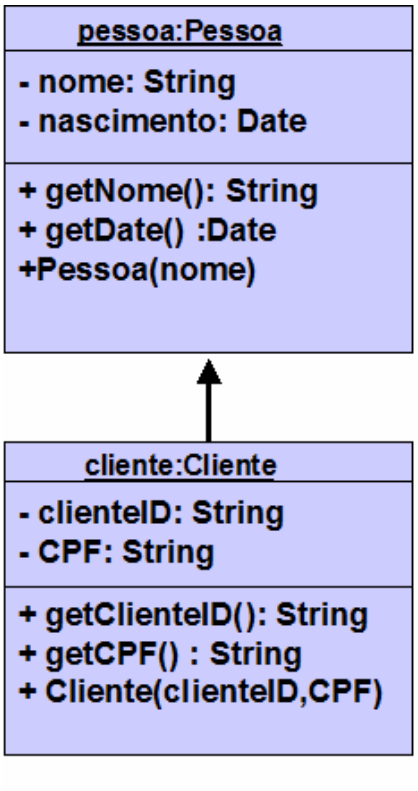
Especialização

Processo pelo qual identificamos atributos incomuns em classes diferentes que são colocados nas subclasses de uma superclasse, tornando-a mais específica e menos genérica.

Representação da Herança

A orientação a objetos possui uma notação especifica para representar que uma classe estende o comportamento de outra, definindo a **herança**.

Exemplo:



Vemos então que a classe “cliente” é filha da classe “pessoa”, e herda os atributos e métodos do pai.

Para o objeto pessoa, posso recuperar o nome e a data de nascimento.

Para o objeto cliente, posso recuperar o nome, a data de nascimento, o CPF e o código do cliente.

Figura 11

A capacidade de **herança** é o mecanismo que garante a linguagem Java altos índices de reaproveitamento de código, permitindo uma melhor componentização.

Um objeto do tipo Cliente, terá todos os atributos e métodos de uma Pessoa e de Cliente.

Os construtores não são herdados, por isso devemos na subclasse especificar construtores que chamem os construtores da **superclasse.**

Árvore de Classes

Uma arvore de classes define um modelo completo de classes para solucionar um problema computacional.

Processo para a criação de árvore de classes:

1. Leitura e entendimento do Problema de Negócio ou Descrição Textual do Problema, ou História do Usuário;
2. Aplicação da técnica da OMT, separando objetos, atributos e métodos.
3. Desenhando as classes aplicando encapsulamento;
4. Redesenhando as classes aplicando os conceitos de herança, especialização (bottom-up) e generalização (top-down)

Exemplo de uma arvore de classes:

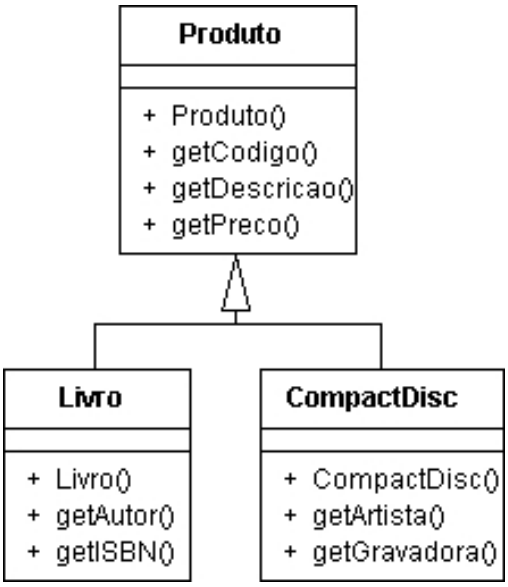


Figura 12

A árvore de classes é um importante elemento na hora de codificarmos nossos sistemas orientados a objetos.

3.3. Tópico Avançado: Formas de herança

Há várias formas de relacionamentos em herança:

Extensão: uma subclasse estende uma superclasse, acrescentando novos membros (atributos e/ou métodos) que definem novos comportamentos que são somados aos já existentes na superclasse. A superclasse permanece inalterada, motivo pelo qual este tipo de relacionamento é tecnicamente referenciado como **herança estrita**.

Especificação: a superclasse abstrata especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Ou seja, especifica-se que apenas a *interface* (conjunto de especificação dos métodos públicos) da superclasse é herdada pela subclasse. Dando a subclasse a responsabilidade de transformar os conceitos abstratos da superclasse em implementações concreta.

Combinação de extensão e especificação: a subclasse herda a interface de uma classe abstrata, e uma implementação padrão dos métodos concretos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse, ou ter que oferecer alguma implementação para métodos que a superclasse tenha declarado como abstratos. Normalmente, este tipo de relacionamento é denominado **herança polimórfica**.

Estes conceitos serão revistos nos próximos módulos e foram inseridos a título de informação.

3.4. Polimorfismo

Analisando a palavra **POLIMORFISMO**, ela significa Muitas Formas. Ou seja, para uma árvore de herança, temos **muitas formas de objetos e métodos** a partir de uma **superclasse** e suas **subclasses**.

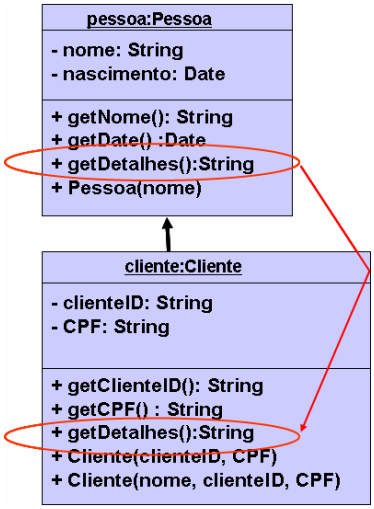
Polimorfismo é o princípio pelo qual usamos objetos construídos a partir de uma arvore de herança, através de referências do tipo de uma superclasse da hierarquia.

Para podermos explorar o polimorfismo, a linguagem que usaremos deve suportar a herança na sua totalidade, provendo, classes e subclasses, que por sua vez tem atributos, métodos e construtores.

Formas de polimorfismo:
Subclasses;
Sobrescrita de método;
Sobrecarga de método ou construtor;

Sobrescrita de método - Method Overriding

Esta utilidade da orientação a objetos e das linguagens orientada a objetos nos permite escrever numa subclasse um ou mais métodos presentes numa das superclasses da árvore de herança podendo alterar o comportamento da superclasse.



Vemos que o método **getDetalhes()** foi escrito na superclasse Pessoa e na subclasse Cliente.

Quando numa subclasse **reescrevemos** um método já existente numa superclasse, chamamos de sobrescrita de método ou reescrita de método, e o termo técnico é **Method Overriding**.

Figura 13

A sobrescrita de métodos só é válida quando numa subclasse o método tenha exatamente a mesma identificação (nome, tipos, etc.) do método da superclasse.

Chamada virtual de métodos - Virtual Method Invocation

Quando através de uma referência de um objeto chamamos a execução de um método, a decisão sobre qual método será executado dentro de uma arvore de herança, de acordo com o tipo da classe, é tomada em tempo de execução através do mecanismo de introspecção da **linguagem Java,** esse processo é chamada virtual de métodos (Virtual Method Invocation**).**

Quando for chamado para execução o método **getDetalhes()** da instancia de Pessoa, o código executado será: **Pessoa.getDetalhes()**

Quando for chamado para execução o método **getDetalhes()** da instancia de Cliente, o código executado será: **Cliente.getDetalhes()**

Ou seja, quando chamarmos um método de um objeto, o método escolhido para ser executado será a primeira ocorrência encontrada daquele método num arvore de herança.

Sobrecarga de métodos - Method Overloading

O Polimorfismo ainda permite que numa mesma classe tenhamos métodos com o mesmo nome, desde que o número ou tipos de parâmetros passados sejam diferentes.

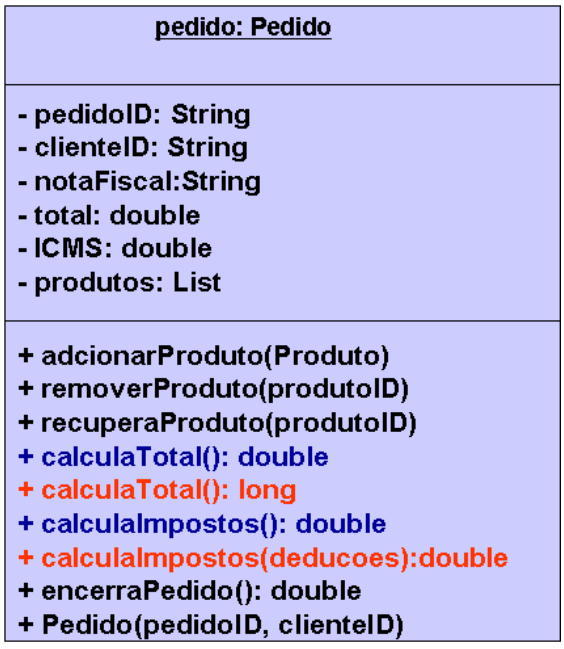


Figura 14

A **sobrecarga de métodos** ou **method overloading**, é uma ferramenta poderosa, pois nos permite criar vários métodos com o mesmo nome, facilitando o entendimento de problemas mais complexos, pois não é necessária a criação de nomes de funcionalidades sem sentido aparente.

3.5. Benefícios no Polimorfismo

O uso do polimorfismo introduz os conceitos relacionados de *herança* e a motivação para a definição de comportamentos (métodos e atributos) melhor abstraídos, melhorando a modelagem para garantir a evolução das aplicações.

É importante sempre lembrar que quando a herança e o polimorfismo estão sendo utilizados, o comportamento que será adotado por um método só será definido durante a execução (invocação virtual de métodos).

Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode ser não-intuitivo, pois acontece em tempo de execução.

Podemos ainda a partir de uma superclasse tratar objetos das suas subclasses.

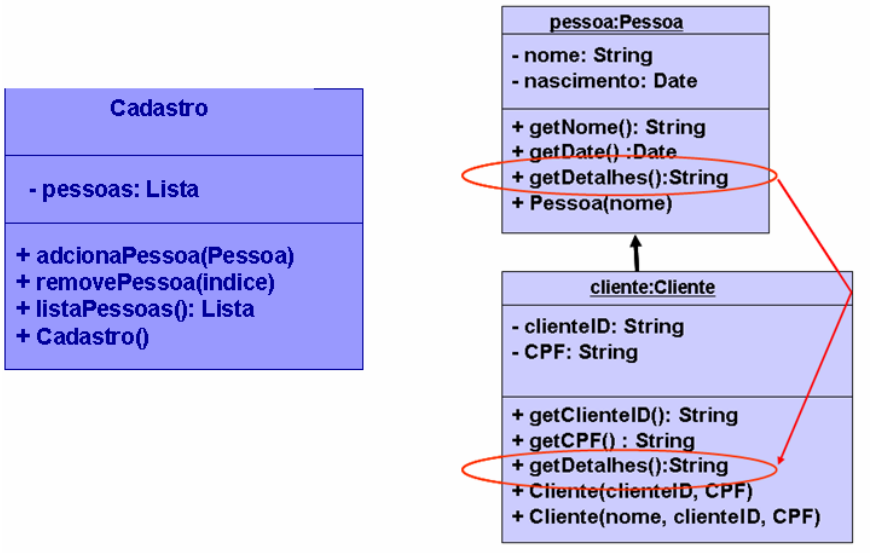


Figura 15

Vendo estas classes de objetos podemos ver que o cadastro, pode cadastrar (ele entende) qualquer pessoa, inclusive os clientes, devido à herança.

Entenderemos melhor esse mecanismo no decorrer do curso quando forem feitos testes com codificação.

Discussão

Qual o maior benefício do encapsulamento?

Quando devemos aplicar e usar a herança?

Existe uma melhor medida de relação entre Número de Superclasses e Subclasses?

4. Desenvolvimento Orientado a Objetos

4.1. Modelos

Como vimos no capítulo passado, os sistemas de *software* de hoje estão cada vez mais complexos. Para lidar com essas complexidades dividimos o problema em pequenos pedaços chamados módulos. Cada paradigma sugere uma forma diferente de obter esses módulos.

Quando estamos analisando e projetando um sistema, é necessário uma forma de representar como quais módulos vão existir, como um módulo está organizado internamente e como eles se comunicam um com os outros. Além disso, precisamos de uma forma de comunicação entre as outras pessoas dentro de uma equipe de desenvolvimento. Para isso utilizamos os modelos.

> Um **modelo** é uma simplificação da realidade.

Os modelos são desenvolvidos também para que possamos entender melhor o sistema que estamos desenvolvendo.

Em todas as disciplinas da engenharia, os modelos são fundamentais. Quando se deseja construir algo, tais modelos (desenhos, esquemas, maquetes, cálculos) são projetados para descrever a aparência e o comportamento do produto. O produto pode ser uma casa, uma máquina, um novo departamento de uma companhia, ou um *software*.

Os modelos ajudam a visualizar o sistema como ele é ou como será, permitindo especificar a estrutura e o comportamento do sistema. Também fornece uma fôrma que nos guiará durante a construção do sistema e padroniza uma documentação das decisões tomadas.

No contexto de desenvolvimento de *software*, modelos podem ser representados por elementos gráficos que seguem algum padrão lógico. Esses gráficos são chamados diagramas.

> Um **diagrama** é uma apresentação de uma coleção de elementos gráficos que possuem um significado predefinido.

Os diagramas fornecem uma representação concisa do sistema. Como diz o velho ditado, “uma figura vale por mil palavras”. No entanto, os modelos de software também são compostos por informações textuais. O conjunto diagrama e sua descrição textual formam a documentação de um modelo.

Além das vantagens técnicas dos modelos, que melhoram a qualidade do produto final, a construção de modelos permite reduzir os custos no desenvolvimento: já que o software é mais pensado antes de sua codificação, a quantidade de erros no produto final é menor e menos dinheiro é gasto para consertá-los.

A escolha de como modelar um problema influencia em como o problema e a solução serão montados. Cada forma de representação de um modelo está ligada a uma forma de solucionar uma parte do problema.

Um modelo pode ser expresso em diferentes níveis de precisão, ou seja, pode ser útil confeccionar modelos menos detalhados e depois detalhá-los mais à medida que o problema vai sendo melhor entendido.

O melhor modelo é aquele que melhor reflete a solução de um problema. Um bom modelo pode ser compreendido por outra pessoa que não aquela que o confeccionou.

Um modelo não é uma peça única mas é composto de diversos artefatos cada um representando uma faceta diferente do problema. Por exemplo, para representar um prédio a ser construído são necessários diversas plantas: a planta da estrutura, da rede hidráulica, da rede elétrica, etc.

4.2. Linguagem de Modelagem Unificada (UML)

O paradigma da orientação a objetos visualiza um sistema de *software* como uma coleção de agentes interconectados chamados objetos. Cada objeto é responsável por realizar tarefas específicas. É através da interação entre objetos que uma tarefa computacional é realizada.

Um sistema de *software* orientado a objetos consiste de objetos em colaboração com o objetivo de realizar as funcionalidades deste sistema. Cada objeto é responsável por tarefas específicas. É através da cooperação entre objetos que a computação do sistema se desenvolve.

Para se projetar um sistema orientado a objetos é necessário uma linguagem de representação de modelos que represente, pelo menos:

1. Quais as tarefas computacionais do sistema.
2. Quais os objetos do sistema.
3. Como os objetos vão colaborar para que cada tarefa seja realizada.

A UML(*Unified Modeling Language*) é uma linguagem padrão para a elaboração da estrutura de projetos de *software* orientados a objetos. A UML pode ser empregada para visualização, especificação, construção e documentação de sistemas complexos de *software*.

A UML pode ser utilizada para a modelagem de sistemas de informação corporativos distribuídos em aplicações baseadas em Web, e até sistemas embutidos de tempo real. É uma linguagem muito expressiva, abrangendo todas as visões necessárias ao desenvolvimento e implantação desses sistemas. Apesar de sua expressividade, é fácil compreender e usar a UML. Aprender a aplicar a UML de maneira efetiva, no entanto, pressupõe o conhecimento dos conceitos de orientação a objetos.

O objetivo da UML é criar uma linguagem única de modelagem que possa ser utilizada por homens e máquinas para modelar *software* com os conceitos de orientação a objetos.

A UML não é uma metodologia, pois o seu objetivo não é dizer como deve ser o processo de desenvolvimento de *software*. Os modelos são os produtos gerados quando se segue um método. Um modelo é expresso através de uma linguagem de modelagem. Uma linguagem de modelagem possui uma notação (símbolos utilizados nos modelos) e um conjunto de regras. As regras são sintáticas, semânticas e pragmáticas.

A **sintaxe** define os símbolos e como eles se combinam entre si na linguagem. Podemos comparar a sintaxe com as palavras na linguagem natural, é importante saber a ortografia correta das palavras e como podemos combiná-las para formar uma frase.

A **semântica** define o significado de cada símbolo e como ele pode ser interpretado de modo isolado e quando combinado com os demais símbolos. No nosso exemplo, trata-se do significado das palavras dentro do contexto.

As **regras pragmáticas** definem a intenção dos símbolos dentro do modelo, tornando o modelo compreensível por todos os que o analisam. Este corresponde na linguagem natural a um bom texto. Livros de estilos de redação ditam esse tipo de pragmatismo.

Os conceitos utilizados nos diagramas são chamados de elementos do modelo. Semântica é o significado de cada elemento representado na notação e das visões que representam a modelagem orientada a objetos. Notação é a representação formal dos elementos, seus relacionamentos e comportamentos. Extensões da notação é a descrição de elementos adicionais para descrição de casos específicos, não abordados na notação básica.

Visualizar, especificar, construir e documentar sistemas complexos de *software* são tarefas que requerem a visualização desses sistemas sob várias perspectivas. Diferentes participantes – usuários finais, analistas, desenvolvedores, integradores de sistema, os testadores, documentadores e gerente de projetos – cada um traz contribuições próprias ao projeto e observa o sistema de maneira distinta em momentos diferentes ao longo do desenvolvimento do projeto.

4.2.1. Visões da UML

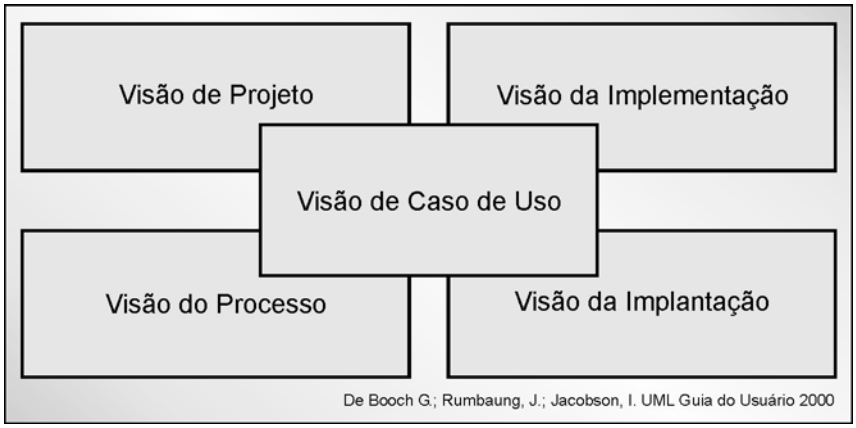


Figura 16.

A **visão do caso de uso** abrange os casos de uso que descrevem o comportamento do sistema conforme é visto pelos usuários finais, analistas e testadores. Ela descreve as tarefas computacionais do sistema.

A **visão de projeto** de um sistema abrange as classes, interfaces e colaborações que formam o vocabulário do problema e de sua solução. Essa perspectiva proporciona principalmente um suporte para os requisitos funcionais do sistema, ou seja, os serviços que o sistema deverá fornecer a seus usuários finais. Esta visão mostra quais os módulos do sistema e como eles interagem para realizar cada tarefa computacional.

A **visão do processo** abrange os *threads* (linhas concorrentes de execução de um programa) e os processos que formam os mecanismos de concorrência e de sincronização do sistema. Essa visão cuida principalmente de questões referentes ao desempenho, à escalabilidade e ao *throughput* (taxas de transmissão) do sistema.

A **visão de Implementação** de um sistema abrange os componentes e os arquivos utilizados para a montagem e fornecimento do sistema físico.

A **visão de Implantação** de um sistema abrange os nós (máquinas) que formam a topologia de hardware em que o sistema é executado. Essa visão direciona principalmente a distribuição, o fornecimento e a instalação das partes que constituem o sistema físico.

Cada uma dessas cinco visões pode ser considerada isoladamente, permitindo que diferentes participantes dirijam seu foco para os aspectos da arquitetura do sistema que mais lhes interessam. Essas cinco visões também interagem entre si – os nós na visão de implantação contêm componentes da visão de implementação que, por sua vez, representa a realização física de classes, interfaces, colaborações e classes ativas provenientes das visões de projeto e de processo. A UML permite expressar cada uma dessas cinco visões e suas interações.

Neste curso nos preocuparemos com duas visões da UML: a de casos de uso e a de projeto.

4.2.2. Partes da UML

Em UML, utilizamos diagramas como os gráficos que mostram os elementos do modelo dispostos de tal forma a fim de ilustrar um aspecto do sistema. Cada tipo de diagrama é uma parte específica de uma visão.

Neste curso aprenderemos a construir os seguintes diagramas:

1. **Diagrama de caso de uso** - descreve os atores externos e suas conexões com as funcionalidades que o sistema oferece.
2. **Diagrama de classe** - descreve a estrutura estática das classes no sistema.
3. **Diagrama de objetos** - demonstra as classes instanciadas num determinado momento no tempo.
4. **Diagrama de seqüência** - mostra a colaboração dinâmica entre os objetos. A seqüência das mensagens é o ponto forte deste diagrama.
5. **Diagrama de colaboração** - mostra a colaboração dinâmica entre os objetos, a ênfase é na troca de mensagens.

No entanto existem outros diagramas importantes na UML:

Diagrama de estados - complementa a descrição das classes. Mostra os estados que o objeto pode ter e os eventos que o fazem mudar de estado.

Diagrama de atividades - demonstra o fluxo seqüencial das atividades.

Diagrama de componentes - mostra a estrutura física da implementação em termos de código.

Diagrama de processos - mostra a arquitetura física do hardware e do software no sistema.

4.3. Programação Orientada a Objetos com Java

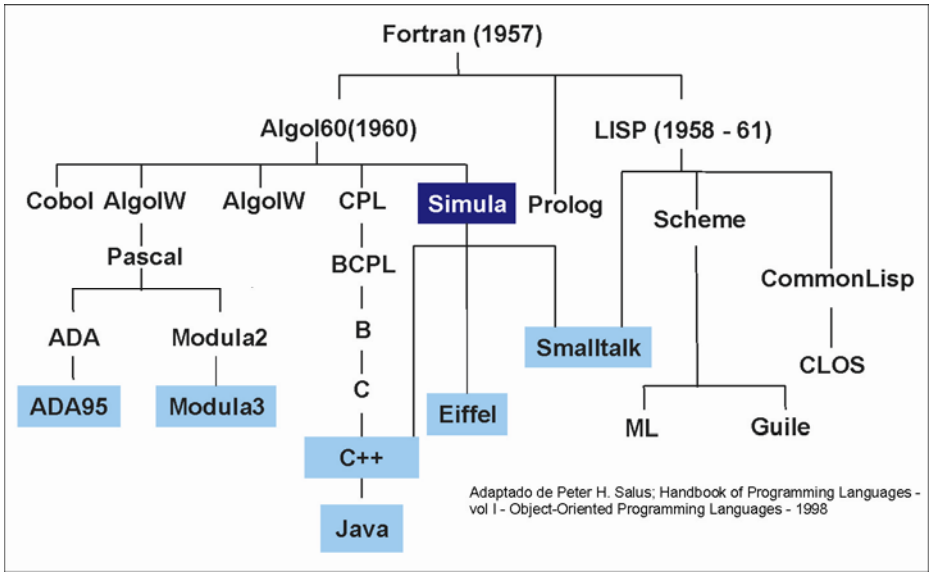


Figura 17.

O nosso objetivo neste curso é construir um programa orientado a objeto. Desta forma, a UML é apenas um meio de representarmos o programa antes de implementá-lo. Para construir, de fato, um programa é necessário uma linguagem de programação. Neste curso utilizaremos a linguagem Java.

Figura 18.

Esta linguagem foi desenvolvida pela Sun Microsytems (), resultado do projeto Green, de 1991. Inicialmente, a linguagem foi chamada por James Gosling, seu criador, de Oak (carvalho), em homenagem a uma árvore que dava para a janela do seu escritório na Sun. Logo descobriu-se que já havia uma outra linguagem com esse nome. Quando uma equipe da Sun visitou uma cafeteria local, o nome Java (ilha da Indonésia grande produtora de café) foi sugerido e agradou. Daí o símbolo da linguagem Java.

Hoje, a evolução da especificação da linguagem Java é gerenciada por uma comunidade, o JCP (*Java Community Process*). Várias empresas grandes fazem parte do comitê executivo da comunidade, tais como a própria Sun, IBM, HP, Google, SAP, Oracle, Nortel, Apple, Borland, etc. Várias empresas do mundo inteiro fazem parte da comunidade, sugerindo modificações e submetendo melhorias já implementadas.

A linguagem Java foi projetada para atender às necessidades do desenvolvimento de aplicações implantadas em diversas máquinas ligadas por uma rede (sistemas distribuídos), podendo inclusive possuir sistemas operacionais diferentes. A seguir, são apresentadas algumas características da linguagem Java que a tornam tão interessante e adequada para o desenvolvimento de uma nova geração de aplicações.

1. Alta produtividade e facilidade de reúso: Java vem acompanhada de um grande número de bibliotecas de classes já testadas e que proporcionam várias funcionalidades (E/S, redes, interface gráfica, multimídia, etc.). Estas bibliotecas podem ser facilmente estendidas, de acordo com as necessidades da aplicação em particular, via herança. Assim o desenvolvimento se torna mais produtivo, pois o programador não precisa “reinventar a roda” a cada novo projeto.
2. Robusta e Segura: Java foi projetada para criar software altamente confiável. Além de prover verificações durante a compilação, Java possui um segundo nível de verificações (em tempo de execução). As próprias características do Java ajudam o programador a seguir bons hábitos de programação. O modelo de gerência de memória é extremamente simples: objetos são criados por meio de um operador *new.* Não existem tipos como apontadores, nem aritmética de apontadores. A liberação da memória dos objetos que não são mais referenciados é feita por meio de um mecanismo chamado *garbage collection* (coleta de lixo). Como Java foi desenvolvida para operar em ambientes distribuídos, aspectos de segurança são de fundamental importância. A linguagem possui características de segurança e a própria plataforma realiza verificações em tempo de execução. Assim, aplicações escritas em Java estão “seguras” contra códigos não autorizados que tentam criar vírus ou invadir sistemas de arquivos.
3. Arquitetura Neutra e Portável: Para suportar a distribuição em ambientes heterogêneos, as aplicações devem ser capazes de executar em uma variedade de sistemas operacionais e de arquiteturas de *hardware*. Para acomodar esta diversidade de plataformas, o compilador Java gera *bytecodes*, um formato intermediário neutro projetado para o transporte eficiente em múltiplos ambientes de *software/hardware*. Assim, a natureza interpretada de Java resolve o problema de distribuição de código binário. A neutralidade da arquitetura é apenas uma parte de um sistema verdadeiramente portável. Java também define os tamanhos de seus tipos básicos e o comportamento de seus operadores aritméticos. Desta forma, não existem incompatibilidades de tipos de dados entre arquiteturas de *software/hardware* diferentes. Esta arquitetura neutra e portável é chamada de Máquina Virtual Java (*Java Virtual Machine*, JVM).
4. Bom desempenho: Desempenho é um fator que sempre deve ser levado em consideração. O interpretador da linguagem possui algumas otimizações que permitem a execução mais eficiente de código em Java. Além disso, aplicações que necessitam de grande poder computacional podem ter suas seções mais críticas reescritas em código nativo (compilado a partir da linguagem C, por exemplo). Obviamente, pela característica interpretada da linguagem, seu desempenho, em geral, é inferior ao de linguagens compiladas para linguagem de máquina (C++, por exemplo). Existem soluções como compiladores *Just-in-Time* (JIT), que compilam os *bytecodes* Java para a linguagem nativa da máquina em particular, que podem melhorar consideravelmente o desempenho das aplicações.
5. Multilinhas (*Multithreaded*) e Dinâmica: Aplicações de rede, como navegadores, tipicamente precisam realizar várias tarefas “simultaneamente”. Por exemplo, um usuário utilizando um navegador pode rodar várias animações enquanto busca uma imagem e faz a rolagem (*scroll*) da página *Web*. A capacidade da linguagem Java de executar várias linhas de processo (*threads*) dentro de um programa permite que uma aplicação seja construída de tal forma que para cada atividade seja reservada uma *thread* exclusiva de execução. Assim, obtém-se um alto grau de interatividade com o usuário da aplicação. Java suporta *multithreading* no nível da linguagem (suporte nativo e não por meio de bibliotecas externas). Além disso, a linguagem oferece séries de primitivas sofisticadas de sincronização (semáforos, por exemplo).

Exercícios

01. Pesquise, na Web, quais empresas estão hoje gerenciando a evolução da UML (você pode encontrar esta informação no site da OMG – *Object Management Group*). Por que essas empresas gerenciam a UML?

Bibliografia

PINTO, Alexandre de Souza, Introdução à Linguagem Java, apostila.

DEITEL, H. M.; DEITEL, P.J. Java – Como Programar. 4ª edição, Porto Alegre, RS, Editora Bookman, 2002

BOOCH, GRADY; RUMBAUGH, JAMES; JACOBSON, IVAR. UML – Guia do Usuário. Editora Campus. 2000.

Leitura complementar

Links interessantes

Orientação a objetos: http://www.mundooo.com.br

Grupo de usuários Java:

Sociedade e Usuário Java: http://www.soujava.org.br

Empregos para profissionais Java:

Revista Mundo Java:

Revista Java Magazine: http://www.javamagazine.com.br

Portal Java: http://www.portaljava.com.br

Desenvolvimento: http://www.imasters.com.br

Site oficial do Java:

Artigo: Alguns números dos 10 anos de Java

Publicado em: 04/04/2005

Fonte: http://www.financialexpress.com/fe_full_story.php?content_id=86910

O site publicou alguns números sobre o Java e como ele está hoje após seus 10 anos de crescimento. Veja abaixo alguns desses números:

- Globalmente, cerca de 4,5 milhões de desenvolvedores trabalham com Java.
- Java é uma indústria de 100 bilhões de dólares por ano.
- Anualmente é investido 2,2 bilhões de dólares em servidores de aplicação Java e 110 bilhões em TI relacionada.

- Há 579 milhões de telefones habilitados com Java.
- Sete entre dez aplicações móveis atualmente em construção usarão o Java Runtime Environment.
- O mercado de jogos para dispositivos móveis é estimado em torno de 3 bilhões de dólares.

Cerca de 750 milhões de Java Cards têm sido implantados globalmente.

5. Casos de Uso

5.1. Conceito

Para desenvolver um sistema é necessário saber exatamente o que este sistema deve fazer. Para isso é necessário fazer um estudo minucioso do ambiente onde o sistema de *software* vai funcionar e quais problemas ele vai resolver dentro do contexto do negócio. Em engenharia de *software*, esses problemas são chamados requisitos funcionais. Uma forma de representar estes requisitos são os casos de uso.

O modelo de casos de uso direciona diversas das tarefas posteriores do ciclo de vida do sistema de *software*. Além disso, o modelo de casos de uso força os desenvolvedores a moldar o sistema de acordo com o usuário.

Um **caso de uso** é a especificação de uma seqüência de interações entre um sistema e os agentes externos. Descreve o comportamento de um sistema, sem revelar a estrutura e como o comportamento deve ser obtido internamente pelo sistema.

> Um caso de uso representa quem faz o que (interage) com o sistema, sem considerar o comportamento interno do sistema.

Normalmente um sistema possui vários casos de uso.

O modelo de casos de uso é usado como fonte de informações essencial para atividades de análise, projeto e teste.

Estas são as pessoas que usarão o modelo de casos de uso:

- O cliente aprova o modelo de casos de uso. Depois de obter a aprovação, você saberá qual é o sistema que o cliente deseja. Você também pode usar o modelo para discutir o sistema com o cliente durante a fase de desenvolvimento.
- Possíveis usuários utilizam o modelo de casos de uso para conhecer melhor o sistema.
- O arquiteto de software utiliza o modelo de casos de uso para identificar a funcionalidade da arquitetura.
- Os *designers* utilizam o modelo de casos de uso para obter uma visão geral do sistema. Assim, por exemplo, quando você refina o sistema, precisa da documentação sobre o modelo de casos de uso para ajudá-lo no trabalho.

- O gerente utiliza o modelo de casos de uso para planejar e acompanhar a modelagem do caso de uso e também o design subseqüente.
- Pessoas que não participam do projeto, mas trabalham na organização, como executivos e comitês gerais de trabalho utilizam o modelo de casos de uso para ter uma idéia do que foi feito.
- Os designers utilizam o modelo de casos de uso como base para seu trabalho.
- Os testadores utilizam o modelo de casos de uso para planejar as atividades de teste (caso de uso e teste de integração) o mais cedo possível.
- Aqueles que desenvolverão a próxima versão do sistema utilizam o modelo de casos de uso para saber como a versão atual funciona.
- Os redatores da documentação utilizam os casos de uso como base para redigir os guias do usuário do sistema.

5.2. Identificação de casos de uso

Passos para identificar casos de uso:

1. Identifique a fronteira do sistema

Separe as atividades passíveis de automatização e aquelas que não podem ser automatizadas. Identifique também os sistemas que deverão ser integrados com o sistema que você está desenvolvendo. Assim é possível definir claramente o que deve ser feito pelo sistema e o que deve ser feito externamente. O nome do sistema deve dizer de forma clara o **objetivo geral** do sistema. Exemplos:

- Sistema de Gerenciamento de Biblioteca (nome claro, significa que o objetivo geral do sistema é gerenciar uma biblioteca inteira, incluindo seu acervo, usuários, empréstimos, etc).
- Sistema Comercial (nome ambíguo, não diz claramente qual é o objetivo do sistema).
- Sistema de Controle de Assinaturas de Revistas (nome claro, pois permite identificar o negócio e o objetivo do Sistema).
- Sistema de Contas (nome ambíguo, não dá para saber o que significa: conta de quê?)

2. Identificar atores

Atores são os elementos externos que interagem com o sistema. Logo, atores **não fazem parte** do sistema mas **trocam** informações com o sistema.

Um caso de uso representa uma seqüência de interações entre o sistema e um ator através de uma troca de informações entre eles para alcançar um objetivo. Este objetivo agrega valor ao ator, no contexto do domínio do sistema, e pode modificar o estado do sistema.

Um ator sempre inicia a seqüência de interações como o sistema, ou provoca um evento que inicia um caso de uso.

Algumas categorias de atores que podem ajudar na identificação:

- **Pessoas** (Empregado, Cliente, Gerente, Almoxarife, Vendedor, etc);
- **Organizações** (Empresa Fornecedora, Agência de Impostos, Administradora de Cartões, etc.);
- **Outros sistemas** (Sistema de Cobrança, Sistema de Estoque de Produtos, etc.);

- **Equipamentos** (Leitora de Código de Barras, Sensor, etc.);
- **Tempo** (um horário que dispara evento do sistema).

Um ator corresponde a um **papel** representado em relação ao sistema.

- O mesmo indivíduo pode ser o Cliente que compra mercadorias e o Vendedor que processa vendas.
- Uma pessoa pode representar o papel de Funcionário de uma instituição bancária que realiza a manutenção de um caixa eletrônico, mas também pode ser o Cliente do banco que realiza o saque de uma quantia.

Um ator pode participar de muitos casos de uso. Um caso de uso pode envolver vários atores, o que resulta na classificação dos atores em primários ou secundários.

Um ator primário é aquele que inicia uma seqüência de interações de um caso de uso. Atores secundários supervisionam, operam, mantêm ou auxiliam na utilização do sistema.

Exemplo: para que o Caixa (ator primário) cancele item de uma nota fiscal (sistema), um outro ator (secundário), o Fiscal deve ser chamado para autorizar tal operação.

3. Identificar os objetivos dos atores

É artificial linearizar estritamente a identificação dos atores e dos objetivos. No entanto, se for claro quem são os atores, facilita a identificação dos objetivos e, por conseguinte, dos casos de uso.

Um objetivo para um ator é tudo aquilo que ele pode obter utilizando o sistema. Por exemplo, o usuário de uma página não tem como objetivo “digitar nome e endereço”, e sim “cadastrar usuário”. “Introduzir cartão” não é um objetivo porque não tem valor isoladamente. “Sacar dinheiro” é um objetivo porque é útil ao ator.

Uma tabela ator-objetivo pode ajudar:

Ator	Objetivo
Caixa	• **Registrar venda** • **Cancelar venda** • **Registrar troca** • **Registrar devolução** • **Registrar pagamento**
Gerente	• **Emitir relatório de venda** • **Ativar promoção** • **Desativar promoção**

Tabela 1.

4. Definir casos de uso

Defina um caso de uso para cada objetivo. Lembre-se que o nome de um caso de uso deve ser composto de um verbo no tempo infinitivo e um objeto. Por exemplo: “Registrar Troca” e não somente “Troca”.

5.3. Descrição de narrativas

Cada caso de uso é descrito através de uma narrativa das interações que ocorrem entre o(s) elemento(s) externo(s) e o sistema.

Há várias formas de se descrever casos de uso.

Grau de abstração: O grau de abstração de um caso de uso diz respeito à existência ou não de **menção à tecnologia** a ser utilizada na descrição deste caso de uso.

Um caso de uso **essencial** não faz menção à tecnologia a ser utilizada.

Um caso de uso **real** apresenta detalhes da tecnologia a ser utilizada na implementação deste caso de uso.

Grau de detalhamento: O grau de detalhamento a ser utilizado na descrição de um caso de uso também pode variar. Um caso de uso **sucinto** descreve as interações sem muitos detalhes. Um caso de uso **expandido** descreve as interações em detalhes.

Formato:

- **Descrição contínua**

O Cliente chega ao caixa eletrônico e insere seu cartão. O Sistema requisita a senha do Cliente. Após o Cliente fornecer sua senha e esta ser validada, o Sistema exibe as opções de operações possíveis. O Cliente opta por realizar um saque. Então o Sistema requisita o total a ser sacado. O Sistema fornece a quantia desejada e imprime o recibo para o Cliente.

- **Descrição numerada**

1. Cliente insere seu cartão no caixa eletrônico.
2. Sistema apresenta solicitação de senha.
3. Cliente digita senha.
4. Sistema exibe menu de operações disponíveis.
5. Cliente indica que deseja realizar um saque.
6. Sistema requisita quantia a ser sacada.
7. Cliente retira a quantia e o recibo.

- **Descrição particionada**

Cliente	Sistema
– Insere seu cartão no caixa eletrônico – Digita senha – Solicita realização de saque – Retira a quantia e o recibo	– Apresenta solicitação de senha. – Exibe operações disponíveis. – Requisita quantia a ser sacada

Tabela 2.

5.3.1. Documentação dos Casos de Uso

Não existe nenhum padrão de documentação de casos de uso. Cada empresa e cada autor utilizam um conjunto de campos para descrever um caso de uso. Abaixo, apresentamos a sugerida por Eduardo Bezerra (vide bibliografia).

[Código] Nome

Descrição: Descrição do caso de uso. Um parágrafo é suficiente.

Importância: Essencial (funcionalidade base para outros caso de uso), Importante (indispensável, mas precisa de outros para funcionar), Desejável (foi solicitado pelo cliente mas não tem urgência e/ou grande valor ao negócio do cliente).

Ator Primário: Ator que inicia o caso de uso

Atores Secundários: Participam da interação com o sistema.

Precondições: O que deve ser verdade no estado inicial do sistema, antes de o caso de uso ser executado.

Fluxo Principal ou Básico:

O fluxo principal descreve a seqüência normal de caso de uso.

O caso de uso inicia quando um ator faz alguma coisa. Em geral, o ator escolhe iniciar o caso de uso. Um ator sempre inicia casos de uso. O caso de uso deve descrever o que o ator faz e qual a resposta do sistema para as ações do ator. Isto pode ser representado como um diálogo entre o ator e o sistema.

O caso de uso deve descrever o que acontece dentro do sistema, mas não como ou por quê. Se ocorrer troca de informação, seja específico sobre o que é passado para o ator e para o sistema. Por exemplo, não é interessante dizer que o ator entra com informação do cliente; é melhor dizer que o ator entra com o nome e o endereço do cliente.

No fluxo principal evite “se ... então ... senão...”. A descrição do fluxo principal deve ser linear e deve conter todos os passos obrigatórios para a execução do caso de uso.

Fluxos alternativos:

Descreve caminhos alternativos que o ator pode tomar para executar o caso de uso. Deve ter uma numeração própria e fazer referência aos passos do fluxo principal, informando quando a alternativa pode ser iniciada e para qual passo do fluxo principal voltará (se voltar).

Fluxos de exceção:

Prevê os possíveis erros do caso de uso, citando o que causou o erro e como o sistema vai reagir. Deve ter uma numeração própria e fazer referência aos passos do fluxo principal onde o erro pode ocorrer, informando o que o sistema vai fazer e para qual passo do fluxo principal voltará (se voltar).

Pós-condições: Condição que deve ser verdadeira quando o caso de uso for executado com sucesso.

Regras do Negócio: referência a alguma regra de negócio utilizada pelo sistema ao executar o caso de uso.

Exemplo (Sistema de automação de vendas)

[CS01] Registrar venda

Descrição: O operador de caixa registra os itens vendidos e registra o pagamento em dinheiro.

Ator primário: Operador de Caixa.

Ator secundário: N/A.

Importância: Essêncial.

Precondições: Os produtos devem estar cadastradas no sistema.

Fluxo Principal

1. O operador escolhe a opção registrar venda.
2. O sistema solicita o código do produto.
3. O operador fornece o código do produto.
4. O sistema exibe os dados do produto: nome e preço. O sistema solicita a quantidade de itens vendidos.
5. O operador fornece a quantidade de itens vendidos.
6. O sistema exibe o subtotal da compra e volta ao passo 2.
7. O operador escolhe finalizar venda.
8. O sistema exibe o valor total da venda e solicita a quantia recebida como pagamento.
9. O operador fornece o valor da quantia.
10. O sistema exibe o valor do troco e autoriza a abertura da gaveta da máquina registradora.

Fluxos Alternativos

1A. No passo 2, o operador escolhe a opção “Buscar produto”, iniciando o caso de uso CS02. Após a finalização deste, o sistema volta ao passo 4.

Fluxos de Exceção

1E. No passo 3, o operador fornece um código de produto inexistente, o sistema exibe uma mensagem “Código inválido” e volta ao passo 2.
2E. No passo 9, o operador fornece um valor menor que o valor total da venda, o sistema exibe mensagem “Valor insuficiente” e volta ao passo 8.
Pós-condições: O sistema terá uma nova venda registrada e terá o valor da venda acrescido no valor do dinheiro em caixa.

5.4. Relacionamentos

Casos de uso e atores não existem sozinhos. Pode haver relacionamentos entre eles.
A UML define diversos relacionamentos no modelo de casos de uso:

Relacionamento de comunicação: Existe somente entre casos de uso e atores. É o relacionamento de interação. Se um ator pode acessar um caso de uso, então eles têm um relacionamento de comunicação.

Relacionamento de inclusão: Existe somente entre casos de uso. Se um caso de uso chama um outro caso de uso a partir do fluxo principal, então existe um relacionamento de inclusão entre eles.

Quando dois ou mais casos de uso incluem uma seqüência de interações comum, esta seqüência comum pode ser descrita em um outro caso de uso. E os casos de uso ligados por inclusão tornam a descrição dos casos de uso mais simples.

Exemplo: considere um sistema de controle de transações bancárias. Alguns casos de uso deste sistema são **Obter Extrato**, **Realizar Saque** e **Realizar Transferência**. Há uma seqüência de interações em comum: a seqüência de interações para identificar o cliente. Assim podemos agrupar esta seqüência em um caso de uso “Identificar Cliente” e chamá-lo no início do caso de uso.

Relacionamento de extensão: Existe somente entre casos de uso. Utilizado para modelar situações onde diferentes seqüências de interações podem ser inseridas em um caso de uso. Sejam A e B dois casos de uso. Se um caso de uso A apresenta a opção de chamar o caso de uso B, ou seja, um dos fluxos alternativos do caso A contém uma chamada do caso de uso B. Neste caso dizemos que B estende A.

Cada uma das diferentes seqüências representa um comportamento opcional, que só ocorre sob certas condições ou cuja realização depende da escolha do ator.

Exemplo: considere um processador de textos. Considere que um dos casos de uso deste sistema seja **Editar Documento**. No cenário típico deste caso de uso, o ator abre o documento, modifica-o, salva as modificações e fecha o documento. Mas, em outro cenário, o ator pode desejar que o sistema faça uma verificação ortográfica no documento. Em outro, ele pode querer realizar a substituição de um fragmento de texto por outro. Assim, um fluxo alternativo do caso de uso **Editar**

documento, é a chamada para o caso de uso **Verificar ortografia**. Um outro fluxo alternativo possível é a chamada para o caso de uso **Substituir texto**.

Relacionamento de generalização

Relacionamento no qual o reúso é mais evidente. Este relacionamento permite que um caso de uso (ou um ator) herde características de um caso de uso (ator) mais genérico.

O caso de uso (ator) herdeiro pode especializar o comportamento do caso de uso (ator) base.

Pode existir entre dois casos de uso ou entre dois atores.

Na **generalização entre casos de uso**, sejam A e B dois casos de uso. Quando B herda de A, as seqüências de comportamento de A valem também para B. Quando for necessário, B pode redefinir as seqüências de comportamento de A. Além disso, B participa em qualquer relacionamento no qual A participa. Uma vantagem disso é que o comportamento do caso de uso original é reutilizado pelos casos de uso herdeiros. Somente o comportamento que não faz sentido ou é diferente para um herdeiro precisa ser redefinido.

A **generalização entre atores** significa que o herdeiro possui o mesmo comportamento que o ator do qual ele herda. Além disso, o ator herdeiro pode participar em casos de uso em que o ator do qual ele herda não participa. Um exemplo: considere uma biblioteca na qual pode haver alunos e professores como usuários. Ambos podem realizar empréstimos de títulos de livros e reservas de exemplares. No entanto, somente o professor pode requisitar a compra de títulos de livros à biblioteca.

5.5. Diagrama de Caso de Uso

Um diagrama de casos de uso é a representação gráfica de um sistema (ou subsistema), através de seus casos de uso.

Notação Básica

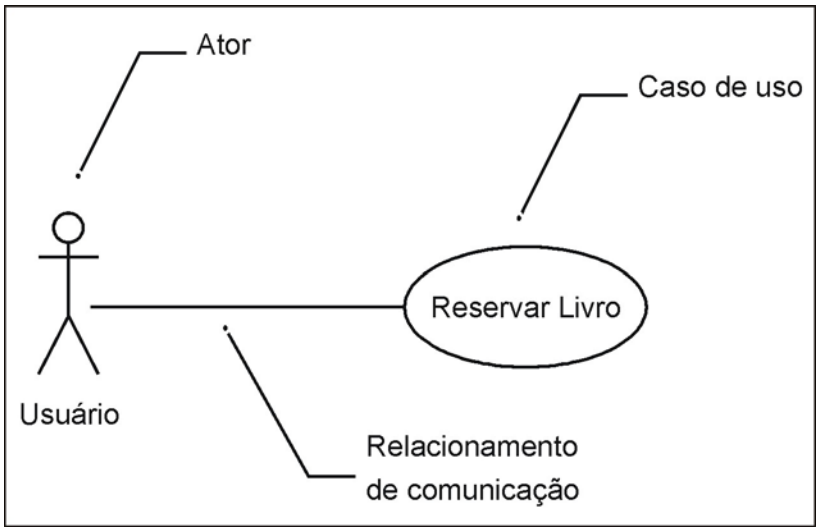


Figura 19.

Notação delimitando a fronteira do sistema

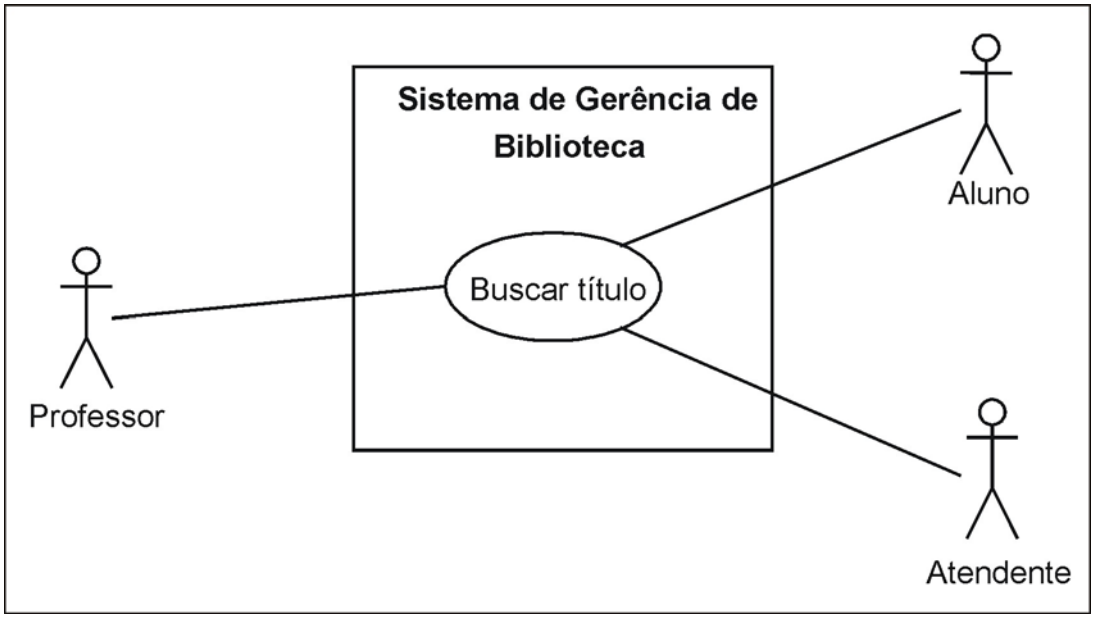


Figura 20.

Notação dos relacionamentos

Inclusão: seta pontilhada rotulada com <inclui> do caso de uso que inclui para incluído.

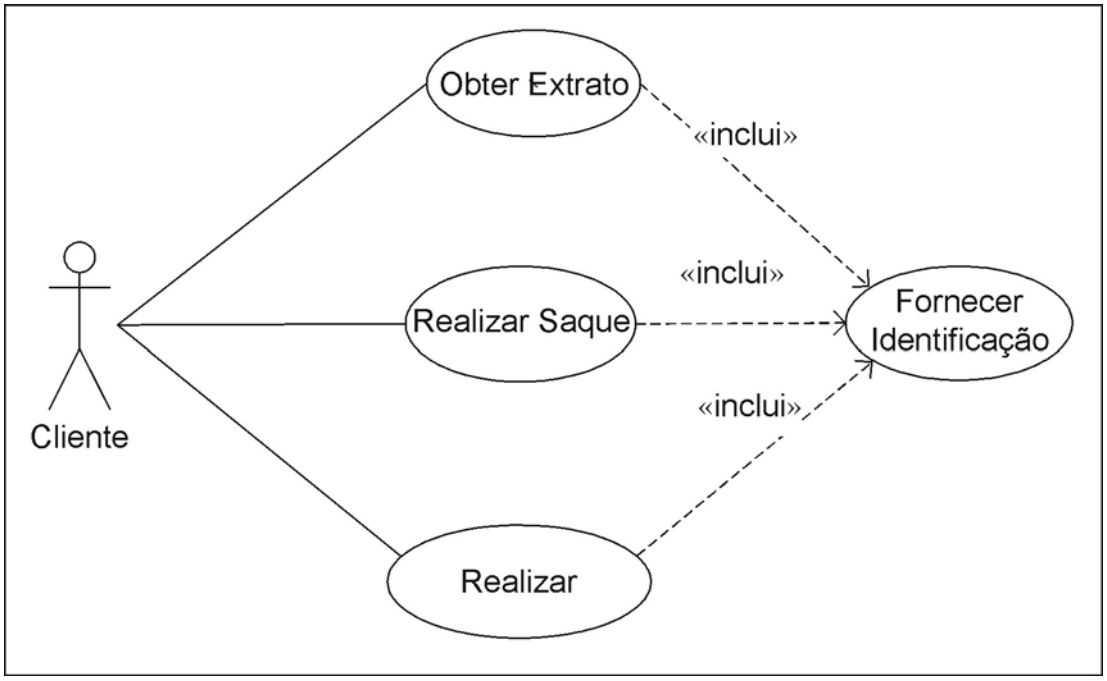


Figura 21.

Extensão: seta pontilhada e rotulada com <estende> do caso de uso que estende para o caso de uso estendido (aquele que oferece a opção de chamar).

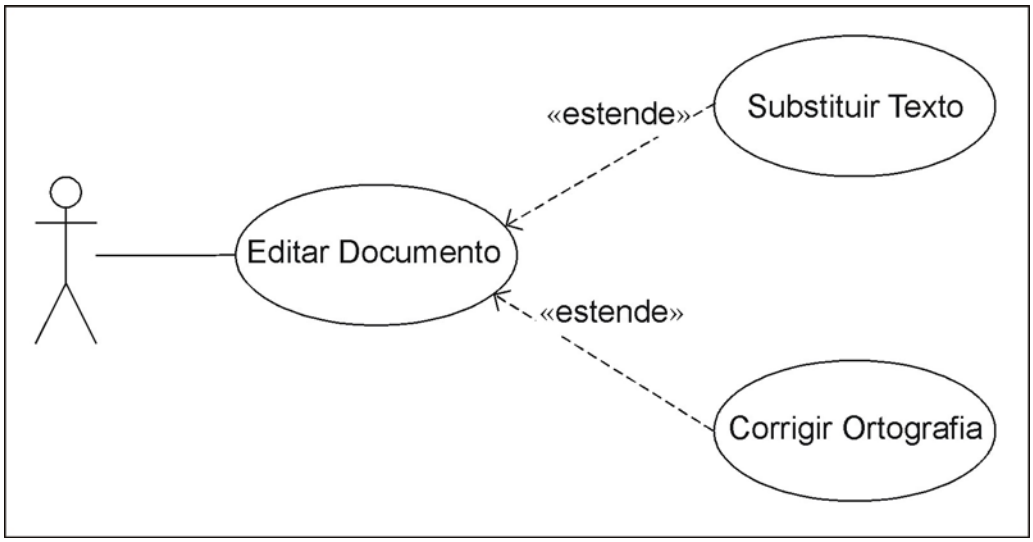


Figura 22.

Generalização entre atores

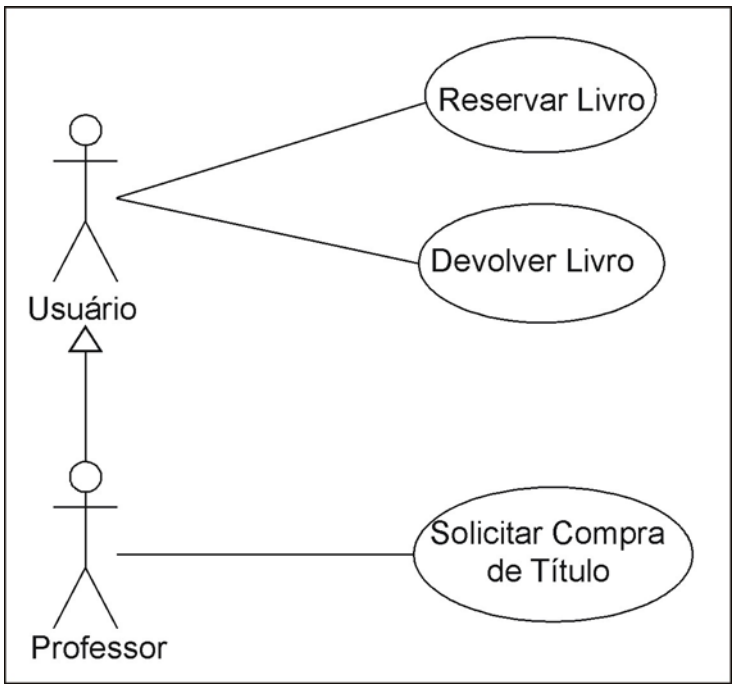


Figura 23.

Generalização entre casos de uso

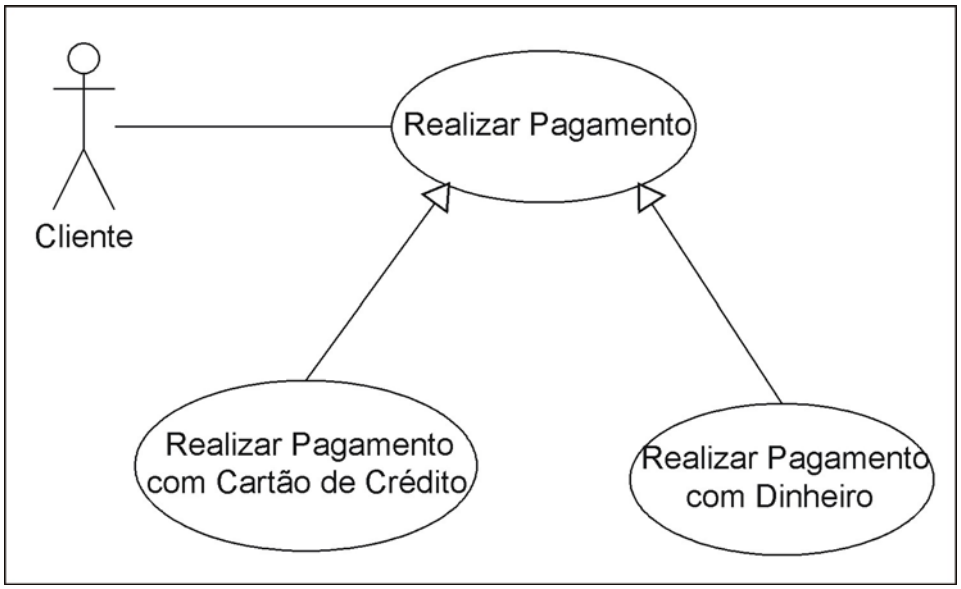


Figura 24.

5.6. Regras do Negócio

São políticas, condições ou restrições que devem ser consideradas na execução dos processos existentes em uma organização. Descrevem a maneira pela qual a organização funciona.

Estas regras são identificadas e documentadas no chamado modelo de regras do negócio. A descrição do modelo de regras do negócio pode ser feita utilizando-se texto informal, ou alguma forma de estruturação.

Alguns exemplos de regras do negócio:

- O valor total de um pedido é igual à soma dos totais dos itens do pedido acrescido de 10% de taxa de entrega.
- Um professor só pode estar lecionando disciplinas para as quais esteja habilitado.
- Um cliente do banco não pode retirar mais de R$ 1.000 por dia de sua conta.
- Os pedidos para um cliente não especial devem ser pagos antecipadamente.

Regras do negócio normalmente têm influência sobre um ou mais casos de uso. Os identificadores das regras do negócio devem ser adicionados à descrição do caso de uso. Utilizando a seção “regras do negócio” da descrição do caso de uso.

Exercícios

01. Identificar os casos de usos e regras de negócio dos enunciados a seguir. Desenhe os diagramas de caso de uso incluindo os relacionamentos.

    a. Uma loja de CDs possui discos para venda e locação. Um cliente pode comprar ou locar uma quantidade ilimitada de discos. Para locar é obrigatório que o cliente esteja cadastrado na loja,

ou seja, tenha preenchido uma ficha de cadastro que deve ser renovada a cada 6 meses. As vendas de CDs podem ser efetuadas à vista com 10% de desconto, ou sem desconto através de cheque pré-datado, descontado 30 dias após a compra. As locações somente podem ser pagas à vista, no ato da devolução dos discos, que tem de acontecer 2 dias após a locação. O valor da locação de cada disco é R$ 1,00. A loja possui um funcionário cuja função é atender os clientes durante a venda e locação dos discos. Suas principais tarefas são: receber e conferir o pagamento efetuado pelos clientes; emitir recibo de venda e locação (este último em duplicata); anotar em uma caderneta o valor de cada venda, assim também como o nome dos discos vendidos; conferir o estado dos CDs devolvidos (caixa, disco e encarte).

b. O vendedor de uma loja de eletrodomésticos, ao efetuar uma venda, encaminha o cliente para o caixa para a realização do pagamento do eletrodoméstico adquirido. O sistema aceita que o pagamento seja feito à vista, no crediário da loja ou por cartão de crédito. Independente da forma de pagamento, o caixa deve verificar, através do sistema, se o cliente não consta do SPC. Após ter recebido o pagamento do cliente, o caixa deve emitir a nota fiscal ao consumidor e efetuar a baixa no estoque. No final do mês o sistema deve emitir um relatório de vendas realizadas para o gerente da loja.

02. Para cada software abaixo e para os objetivos especificados, acompanhe a interação descrevendo-a na forma de um caso de uso.

a. **Software**: Calculadora do windows
Objetivo: Realizar um cálculo

b. **Software**: Outlook
Objetivo: Enviar email

c. **Software**: Word
Objetivo: Localizar palavra

d. **Software**: Excel
Objetivo: Incluir função

e. **Página**: www.orkut.com
Objetivo: Editar perfil

Bibliografia

BEZERRA, Eduardo. *Princípios de análise e projeto de sistemas com UML*. Editora Campus. Rio de Janeiro 2002.

LARMAN, Craig. *Utilizando UML e Padrões*. Bookman Companhia Editora, 2ª. Edição. Porto Alegre, 2004.

Leitura complementar: Dividir para integrar

Crônica 34 de Metódio Prudente (personagem do Prof. Wilson de Pádua Paula Filho)

Publicada online em http://www.wppf.uaivip.com.br/metodio/Prudente.pdf

Divide et impera, dizia Maquiavel aos pupilos da política. Engenheiros aplicam tal máxima na construção de sistemas complexos: divida-se o problema em problemas mais simples, tanto quanto necessário. Resolvam-se os miniproblemas, dos quais resultará uma coleção de mini-soluções. E integrem-se as mini-soluções, para construir o sistema total.

Feita a divisão, os políticos se defrontam com a estratégia de integração. César pode dividir a Gália em três partes (*Gallia est omnis divisa in partes tres*...), mas terá que decidir em que ordem conquistará os divididos gauleses. Considerando-se que os belgas têm fama de serem os mais ferozes (para desgosto de Abraracourcix), César os atacará em primeiro lugar ou em último? Ou conquistará um pouco de território de cada vez, alternando entre as partes suas incursões?

Os engenheiros enfrentam, na integração, várias restrições diferentes: disponibilidade de recursos materiais e humanos, interesses do cliente, e as onipresentes leis da física. Estas, por exemplo, quase sempre ditarão aos engenheiros civis a integração *bottom-up*, de baixo para cima. Só no universo de Salvador Dalí um terceiro andar poderia flutuar no ar, à espera de seus inferiores.

Outro dilema da integração: horizontal versus vertical. Ao construir um conjunto com múltiplos edifícios, o engenheiro poderia construir um prédio inteiro de cada vez (**integração vertical**), ou construir em paralelo o mesmo andar em todos os edifícios (**integração horizontal**). A conveniência de cada uma depende das demais restrições: a integração vertical sempre tem a vantagem de oferecer mais cedo edifícios prontos, ou seja, subconjuntos resolvidos do escopo do problema. Entretanto, a integração vertical exige baixo acoplamento entre as partes, e sempre existe uma unidade mínima, dentro da qual só a integração horizontal funciona. Dadas as leis da resistência dos materiais, assim como outras considerações de engenharia, dificilmente se conseguiria partir verticalmente a construção de um único edifício; fazendo primeiro, por exemplo, apenas o apartamento mais a leste de cada andar, ou, quem sabe, apenas a cozinha de cada um desses apartamentos.

O engenheiro de *software*, bem menos restrito pela física, pode levar muito mais longe a integração vertical. Ela é praticada em todos os processos que privilegiam o **desenvolvimento dirigido por casos de uso**: a unidade de integração passa a ser uma colaboração, arranjo de instâncias de classes e relacionamentos que implementam um caso de uso. Como, por definição, cada caso de uso representa uma função independente, sua implementação representa um acréscimo de funcionalidade visível e significativo para o usuário e o cliente. A estratégia também reduz os riscos, desde que os primeiros casos de uso exercitem as soluções para os riscos mais graves.

Resta o problema do acoplamento. Sempre existem intersecções entre as colaborações que realizam os casos de uso, e tanto mais quanto mais orientada a objetos for a solução.

6. Objetos e Classes

6.1. Conceitos

Olhe em sua volta no mundo real. Para onde quer que você olhe, você vê – objetos! Pessoas, animais, plantas, carros, aviões, construções, computadores, etc.. Seres humanos pensam em termos de objetos. Temos a maravilhosa habilidade da abstração que nos permite visualizar imagens em uma tela como objetos, tais como pessoas, aviões, árvores, montanhas, em vez de ver pontos coloridos isolados. Podemos, se desejarmos, pensar em termos de praias em vez de grãos de areia, florestas em vez de árvores e casas em vez de tijolos.

E é isto que a orientação a objetos explora para o desenvolvimento de *software*. Neste capítulo apresentaremos os principais conceitos da orientação a objetos. A orientação a objetos é um modo natural de pensar sobre o mundo e de escrever programas de computador. Este capítulo pretende ajudá-lo a pensar orientado a objetos, de modo que você possa facilmente colocar em uso os conceitos na programação orientada a objetos (OO).

Em OO, o *software* desenvolvido é organizado como uma coleção de unidades que saibam se tratar por si só (“lego”).

A orientação a objetos não se baseia nos mesmos conceitos que a programação imperativa (ou estruturada); no entanto, não é uma completa ruptura com tudo o que algum dia aprendemos sobre *software*. Na verdade, é uma etapa evolucionária, que dá mais ferramentas para enfrentar o desafio de desenvolver softwares cada vez mais complexos.

6.1.1. Objeto

A Orientação a Objetos lida, naturalmente, com o conceito de Objeto, que é um componente do mundo real. Tudo que podemos ver é um objeto (coisas concretas). Por exemplo: temos que lâmpada da sua sala de aula é um objeto, cada carteira é um objeto, cada pessoa é um objeto, etc.

Em um sentido geral, um objeto é algo que ocupa espaço no mundo real ou conceitual e tem alguma utilidade.

Todo objeto deve ter algo que o diferencie de outro objeto encontrado em seu contexto. Um objeto do mundo real ocupa lugar no espaço e assim os diferenciamos (embora tenham o mesmo formato e sirvam para exatamente a mesma coisa). A esta característica que permite diferenciar um objeto de outro damos o nome de **identidade**.

Um objeto não é apenas algo que costuma ocupar espaço no mundo real; também é algo com o qual é possível fazer coisas, que possuem uma utilidade. O conjunto de operações que um objeto consegue realizar é chamado de **comportamento** do objeto.

Os objetos, apesar de terem as mesmas características, podem possuir um valor diferente para uma delas. A lâmpada 1 pode estar acesa e a lâmpada 2, idêntica a lâmpada 1, pode estar apagada. Assim como o carro 1, pode estar parado e o outro carro 2 pode estar movimentando-se. Assim, um objeto também tem um **estado**, que, nesse sentido, envolve todas as propriedades do objeto juntamente com os valores atuais de cada uma dessas propriedades. O estado do objeto, portanto, é dinâmico. Assim ao visualizar seu estado, você está realmente especificando o valor de seu estado em um determinado momento no tempo e no espaço.

Como estamos querendo desenvolver software, nossos objetos serão entidades de software que possuem:

- **Identidade**: lugar que ocupa na memória, representado por seu nome.
- **Comportamento**: operações que podem ser executadas pelo objeto.
- **Estado**: valor das propriedades do objeto.

6.1.2. Classe

Como na vida real, em um software podem existir diversos objetos do mesmo tipo. Imagine o sistema de gerenciamento de veículos no Detran. Quantos objetos do tipo carro este sistema deve manipular por dia? Se fosse feita uma descrição para cada um, o sistema jamais iria funcionar.

Assim, podemos descrever um mesmo tipo de objetos uma única vez, e cada vez que um novo objeto precise ser criado, é só utilizar a mesma fôrma. Esta fôrma é chamada **classe**.

Logo, uma classe é a descrição de um conjunto de objetos que possuem os mesmos atributos, operação e semântica. Dizemos que um objeto é uma **instância** de uma classe, quando ele é criado a partir dela, seguindo sua especificação.

Vamos supor que você queira construir uma casa. Quando o construtor finalmente lhe entregar as chaves de sua casa e você e sua família estiverem entrando pela porta da frente, nesse momento você está lidando com algo concreto e específico. Não é mais apenas uma grande casa de três quartos com porão, mas é a “minha casa de três quartos com um porão, localizada na Rua das Flores, 835”. Existe uma diferença fundamental entre uma casa de três quartos com porão e minha casa de três quartos. A primeira é uma **abstração** que representa um certo tipo de casa com várias propriedades; a segunda é uma instância concreta dessa abstração, representando alguma coisa que é manifestada no mundo real, com valores reais para cada uma dessas propriedades.

A abstração denota a essência ideal de uma coisa; a instância denota uma manifestação concreta.

A Classe funciona como um “modelo” para a criação de objetos em tempo de execução dos programas. Poderíamos dizer que,na codificação existem somente as Classes, não os Objetos. Já em tempo de execução, o que existe são os Objetos, e não as Classes.

Da mesma forma que dizemos que uma variável é de um determinado tipo (inteiro, booleano, real, caractere, etc.) numa linguagem de programação, dizemos que um objeto é de uma determinada classe.

Uma classe descreve uma categoria de objetos por meio de atributos e métodos.

❑ Atributo

Para descrever uma carteira podemos usar as propriedades altura, comprimento, largura, cor, material. Para descrever um livro podemos usar o título, autor, número de páginas. Estas informações são chamadas atributos.

Os atributos definidos em uma classe definem qual o estado que o objeto pode assumir.

Por exemplo, se os atributos de uma classe Livro são título, autor, número de páginas, os objetos definidos a partir desta classe podem ser:

Identidade: livro1Título: “Memórias Póstumas de Brás Cubas”Autor: “Machado de Assis”Número de páginas: 208	Identidade: livro2Título: “O Nome da RosaAutor: “Umberto Eco”Número de páginas: 562	Identidade: livro3Título: “Java como programar”Autor: “Harvey M. Deitel”Número de páginas: 1386

Tabela 3.

Portanto, atributos são as propriedades definidas nas classes que definem quais as informações o objeto poderá guardar.

❑ Métodos

Todo objeto pertencente à mesma classe exibe o mesmo comportamento. Isto significa que em uma classe é necessário descrever quais as possíveis operações que o objeto pode realizar. Estas operações são chamadas métodos.

Todas as classes exibem comportamentos (por exemplo, uma bola rola, salta, incha e esvazia; um bebê chora, dorme, engatinha, passeia e pisca; um carro acelera, freia e muda de direção; uma toalha absorve água, etc.).

Os seres humanos aprendem sobre objetos estudando seus atributos e observando seus comportamentos. Objetos diferentes podem ter atributos semelhantes e podem exibir comportamentos semelhantes. Comparações podem ser feitas, por exemplo, entre bebês e adultos e entre seres humanos e chimpanzés. Carros, caminhões, pequenas caminhonetes vermelha e patins têm muito em comum.

Um método é a implementação de um serviço que pode ser solicitado a algum objeto da classe para exibir um comportamento. Uma classe pode ter qualquer número de métodos ou até não ter nenhum método.

6.1.3. Encapsulamento

No mundo real, não sabemos como um objeto funciona interiormente, ou seja, como uma operação é implementada. Por exemplo, quando ligamos a TV, não temos de saber como a corrente elétrica é captada e como o sinal da antena é transformado em imagem. Quando usamos uma geladeira, não precisamos saber como a compressão e expansão do gás no motor geram a diminuição da temperatura. Se precisássemos ativar cada um dos comportamentos intermediários para obter um serviço de todos os objetos que utilizamos na vida real, a vida seria muito mais complexa. As operações realizadas para fazer funcionar a TV ou a geladeira estão encapsuladas, ou seja, ocultas do usuário do objeto. Em objetos de software, nós, os analistas, projetistas e programadores, definimos quais os comportamentos de um objeto que desejamos ocultar e aqueles que desejamos tornar acessíveis. Para isso definimos a visibilidade dos atributos e métodos.

As visibilidades permitidas são:

Público	**Atributos**: podem ter seus valores alterados por qualquer outro objeto pertencente a qualquer classe. **Métodos**: podem ser chamados por qualquer objeto de qualquer outra classe
Privado	**Atributos**: só podem ter seus valores alterados pelo próprio objeto. **Métodos**: só podem ser chamados por métodos do próprio objeto.
Protegido	Como o privado, mas podem ser acessados por subclasses (como veremos no Capítulo 9).

Tabela 4.

O encapsulamento de *software* é um conceito quase tão antigo quanto o próprio *software*. No princípio da década de 40, alguns programadores notaram que o mesmo modelo de instrução apareceria diversas vezes dentro do mesmo programa. Alguns estudiosos logo concluíram que esse tipo de modelo repetido poderia ser transportado para um canto do programa e mesmo requisitado, sob a forma de um único nome a partir de vários pontos diferentes do programa principal.

Apesar de ser possível definir um atributo como público, tal prática não é recomendável pois permite a modificação de um atributo por outro objeto, violando assim o princípio de coesão (que diz que cada módulo, ou seja, em OO, cada classe só deve conter o que tem relação com o conceito que representa). Se um outro objeto1 do tipo Classe1 modifica um valor de um outro objeto2 do tipo Classe2, das duas uma: ou o objeto1 contém uma operação que deveria ser definida na Classe2 (e ser executada pelo objeto2); ou o atributo está definido na classe errada.

6.1.4. Diagrama de Classes em UML

Notação para Classe – UML

Nome da Classe
Atributos
Método

O topo do retângulo contém o nome da classe, em negrito e centralizado. O nome é derivado do domínio do problema e não pode ser ambíguo. Deve ser escrito com letra inicial maiúscula e no singular. Recomenda-se que não tenha sufixo e nem prefixo.

Sintaxe de especificação de atributo

<visibilidade> <nome_do_atributo>:<tipo_do_atributo>=<valor_inicial>

Onde:

visibilidade: + para público e – para privado;

nome_do_atributo: palavra que especifica uma característica, iniciada por letra minúscula e sem espaços;

tipo_do_atributo: tipo que qualifica o atributo. Pode ser, por exemplo, *String* (para texto), int (para números inteiros), *double* (para números reais), *boolean* (para verdadeiro e falso);

valor_inicial: valor que o atributo deve ter no começo.

Sintaxe de especificação de método

<visibilidade> <nome_do_método>(<lista_de_parâmetros>):<tipo_do_retorno>

Onde:

visibilidade: + para público e – para privado;

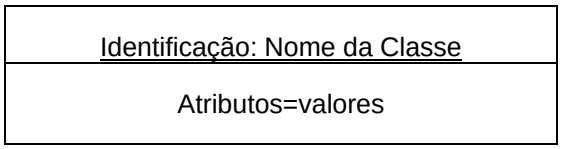
nome_do_método: palavra que especifica uma ação, iniciada por letra minúscula e sem espaços (pode-se usar letra maiúscula para separar palavra dentro do nome do método);

lista_de_parâmetros:<nome_parametro1> : <tipo_do_parametro1>,
<nome_parametro2> : <tipo_do_parametro2>

Um parâmetro é uma informação externa (que não está na classe) que um método pode precisar para executar sua função corretamente. A regra para nome e tipo de atributo também vale para o nome e tipo do parâmetro.

tipo_do_retorno: tipo que qualifica a informação devolvida por um método. Pode ser, por exemplo, *String* (para texto), int (para números inteiros), *double* (para números reais), *boolean* (para verdadeiro e falso).

Notação UML – Objeto



A notação de objetos utiliza-se da mesma notação da classe, pois o objeto nada mais é do que a instância da classe. É um exemplo do estado da classe num determinado momento.

Exemplo de classe e objeto em UML

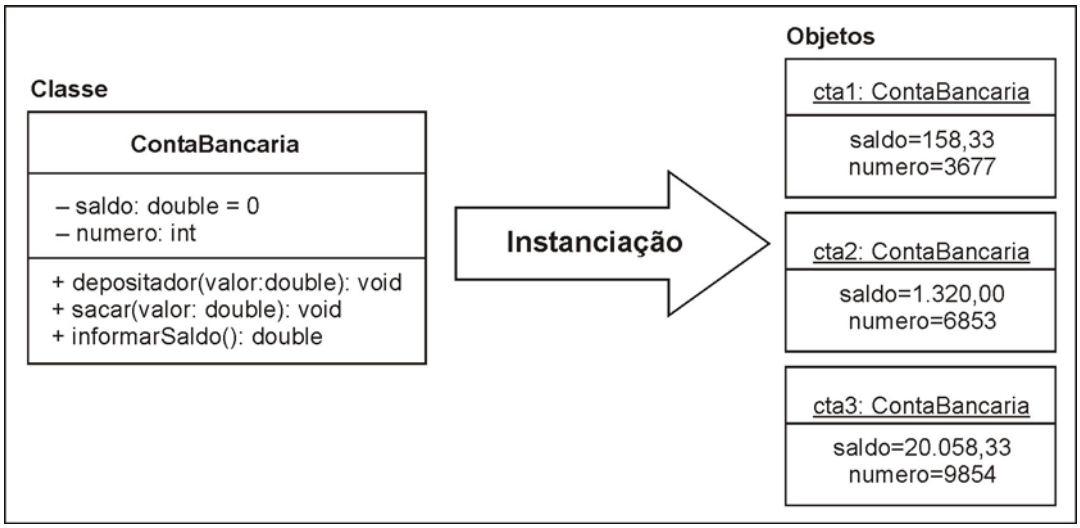


Figura 25.

Existem duas diferenças entre objetos da mesma classe. Cada objeto tem um identificador diferente e, a determinada hora, cada objeto provavelmente terá um estado diferente (o que significa valores diferentes armazenados em seus atributos).

6.2. Associação entre Classes

Um objeto no mundo real nunca está isolado. Está sempre associado a outros. Por exemplo, uma biblioteca possui vários exemplares de livros, um carro possui quatro pneus, uma porta possui uma fechadura, um aluno está matriculado em um curso.

Em um sistema, os objetos também estão associados a outros. Para representar que dois tipos de objetos podem estar associados, ligamos as duas classes por uma linha.

Exemplo (Diagrama de classe representando que um cliente possui uma conta bancária).

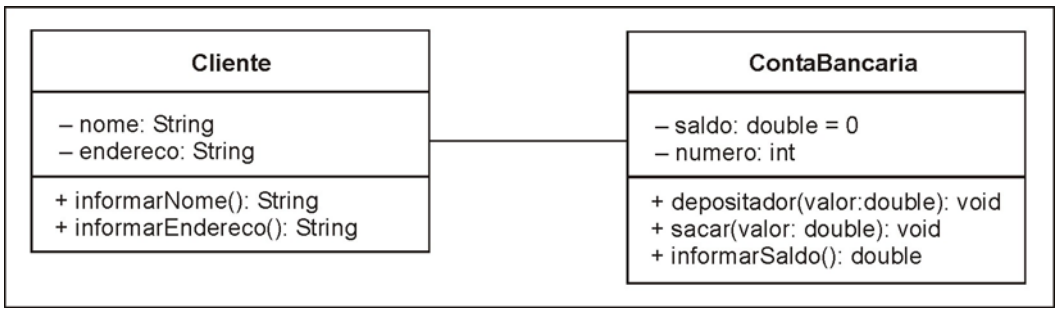


Figura 26.

No Capítulo 7, vamos aprender como detalhar a especificação de uma associação.

6.3. Responsabilidades de Classes

No desenvolvimento de sistemas orientados a objeto, pode ser difícil encontrar classe somente fazendo uma analogia com o mundo real. O mundo real é muito mais complexo do que um sistema, e a abordagem, utilizando simulação, dificulta o processo de abstração.

Na prática, uma responsabilidade é alguma coisa que um objeto conhece ou faz (sozinho ou não).

- Um objeto Cliente conhece seu nome, seu endereço, seu telefone, etc.
- Um objeto Pedido conhece sua data de realização e sabe fazer o cálculo do seu total.

Se um objeto tem uma responsabilidade com a qual não pode cumprir sozinho, ele deve requisitar colaborações de outros objetos.

Costuma-se categorizar os objetos de um sistema de acordo com o tipo de responsabilidade a ele atribuída:

- objetos de entidade;
- objetos de controle;
- objetos de fronteira.

Esta categorização foi proposta por Ivar Jacobson em 1992 (vide bibliografia) em uma técnica denominada Análise de Robustez.

Objetos de entidade

Um objeto de entidade é um repositório para alguma informação manipulada pelo sistema. Esses objetos representam conceitos do domínio do negócio. Normalmente esses objetos armazenam informações persistentes. Há várias instâncias de uma mesma classe de entidade coexistindo no sistema.

Atores não têm acesso direto a estes objetos. Objetos de entidade comunicam-se com o exterior do sistema por intermédio de outros objetos. Objetos de entidade normalmente participam de vários casos de uso e têm um ciclo de vida longo.

Exemplo: Um objeto Pedido pode participar dos casos de uso Realizar Pedido e Atualizar Estoque. Este objeto pode existir por diversos anos ou mesmo tanto quanto o próprio sistema.

Responsabilidades de fazer típicas de objetos de entidade:

- Informar valores de seus atributos a objetos de controle.
- Realizar cálculos simples, normalmente com a colaboração de objetos de entidade associados através de agregações.
- Criar e destruir objetos parte (considerando que o objeto de entidade seja um objeto todo de uma agregação).

Os objetos de entidade são modelados por classes de entidade. As classes de entidade são representadas como abaixo:

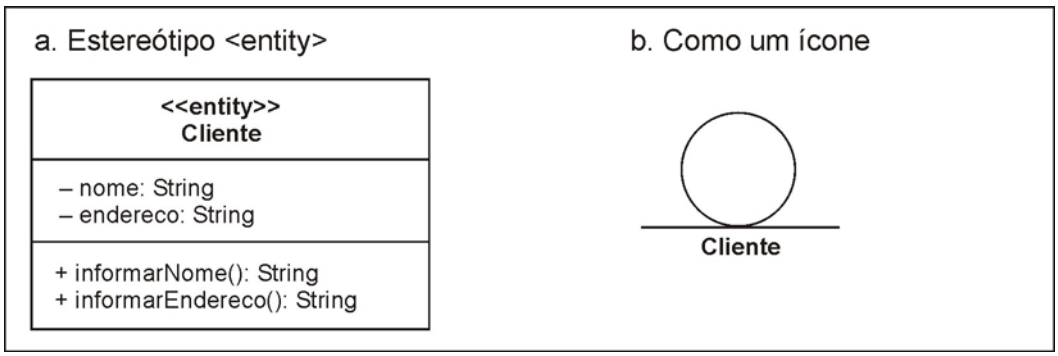


Figura 27.

A utilização de um ou de outro depende do tipo de visualização que se deseja obter por responsabilidade ou pela estrutura da classe.

Objetos de interface ou fronteira (*boundary*)

Esses objetos traduzem os eventos gerados por um ator em eventos relevantes ao sistema. Também são responsáveis por apresentar os resultados de uma interação dos objetos em algo inteligível pelo ator. Um objeto de fronteira existe para que o sistema se comunique com o mundo exterior.

Por conseqüência, estes objetos são altamente dependentes do ambiente.

Objetos de fronteira realizam a comunicação do sistema com atores, sejam eles outros sistemas, equipamentos ou seres humanos.

Há três tipos principais de objetos de fronteira:

– os que se comunicam com o usuário (atores humanos);
– os que se comunicam com outros sistemas;
– os que se comunicam com dispositivos atrelados ao sistema.

Tipicamente têm as seguintes responsabilidades de fazer:

– Notificar aos objetos de controle de eventos gerados externamente ao sistema.
– Notificar aos atores do resultado de interações entre os objetos internos.

Responsabilidades de conhecer de classes de fronteira para interação humana representam informação manipulada através da interface com o usuário. A construção de protótipos pode ajudar a identificar essas responsabilidades.

Responsabilidades de conhecer para classes de fronteira que realizam comunicação com outros sistemas representam propriedades de uma interface de comunicação.

As classes de fronteira modelam objetos de fronteira:

- Modela as partes do sistema que dependem dos atores - coletam requisitos vindos das fronteiras do sistema.
- Geralmente representam abstrações de janelas, formulários, painéis, interfaces de comunicação, interfaces de impressão, sensores e terminais.

- Cada classe de fronteira deve se relacionar com pelo menos um ator e vice-versa.

As classes de fronteira são representadas como abaixo:

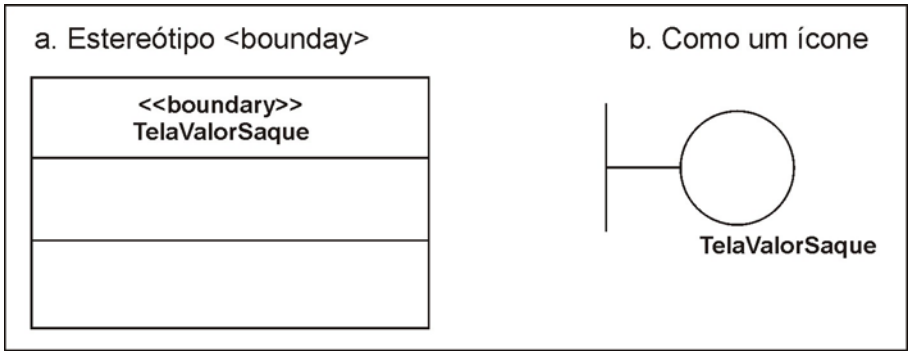


Figura 28.

Neste curso, não veremos a construção de interface gráficas em Java e por isso nossas classes de fronteira não serão especificadas.

Objetos de controle

Os objetos de controle são a “ponte de comunicação” entre objetos de fronteira e objetos de entidade. São responsáveis por controlar a lógica de execução correspondente a um caso de uso. Decidem o que o sistema deve fazer quando um evento externo relevante ocorre.

– Eles realizam o controle do processamento.
– Agem como gerentes (coordenadores, controladores) dos outros objetos para a realização de um caso de uso (ou mais casos de uso, quando são casos de uso muito simples).

Os objetos de controle encapsulam a lógica de controle do caso de uso. Traduzem eventos externos em operações que devem ser realizadas pelos demais objetos. Ao contrário dos objetos de entidade e de fronteira, objetos de controle são tipicamente ativos. Consultam informações e requisitam serviços de outros objetos.

Estes objetos possuem responsabilidades de fazer típicas:

- Realizar monitorações, a fim de responder a eventos externos ao sistema (gerados por objetos de fronteira).
- Coordenar a realização de um caso de uso através do envio de mensagens a objetos de fronteira e objetos de entidade.
- Assegurar que as regras do negócio estão sendo seguidas corretamente.
- Coordenar a criação de associações entre objetos de entidade.

A responsabilidade de conhecer deste tipo de objeto, geralmente, é a de manter valores acumulados, temporários ou derivados durante a realização de um caso de uso. Podem também ter o objetivo de manter o estado da realização do caso de uso.

Os objetos de controle têm vida curta: normalmente existem somente durante a realização de um caso de uso.

As classes de controle são representadas como abaixo:

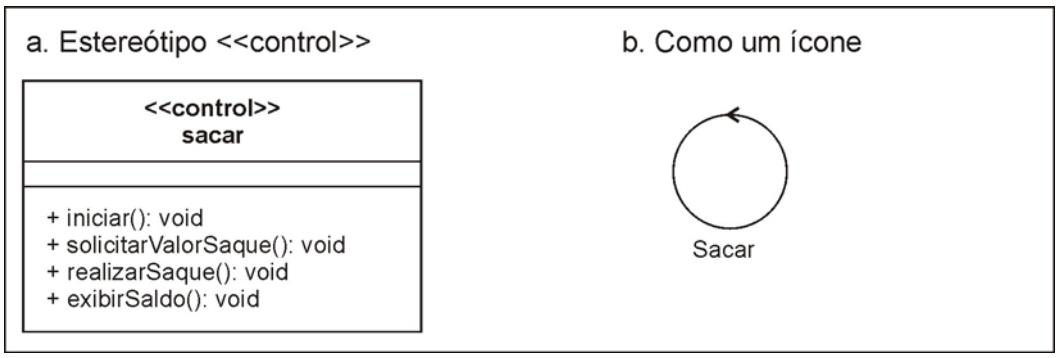


Figura 29.

A categorização das responsabilidades tem como conseqüência a especialização de cada objeto em um dos três tipos de tarefa:

– comunicar-se com atores (fronteira);
– manter as informações do sistema (entidade);
– coordenar a realização de um caso de uso (controle).

Essa categorização facilita a adaptação do sistema a eventuais mudanças. Pois, se cada objeto tem funções específicas dentro do sistema, eventuais mudanças no sistema podem ser menos complexas e mais localizadas.

Logo, uma modificação em uma parte do sistema tem menos possibilidades de resultar em mudanças em outras partes.

Tipo de mudança	**Onde mudar**
Mudanças em relação à interface gráfica, ou em relação à comunicação com outros sistemas.	Fronteira
Mudanças nas informações manipuladas pelo sistema	Entidade
Mudanças em funcionalidades complexas (lógica do negócio)	Controle

Tabela 5.

Exemplo: vantagem de separação de responsabilidades em um sistema para uma loja de aluguel de carros.

- Se o sistema tiver de ser atualizado para que seus usuários possam utilizá-lo pela *Internet*, a lógica da aplicação não precisaria de modificações.
- Considerando a lógica para calcular o valor total das locações feitas por um cliente: se esta lógica estiver encapsulada em uma classe de controle, somente esta classe precisaria de modificação.

A construção de um sistema de *software* que faça separação das responsabilidades de apresentação (fronteira), de lógica da aplicação (controle) e de manutenção dos dados (entidade) facilita também o reúso dos objetos no desenvolvimento de sistemas de *software* semelhantes.

6.4. Análise de um Caso de Uso

Tipicamente, um caso de uso necessita dos três tipos de objeto para ser realizado. Os três tipos de objeto colaboram para obter o objetivo final do caso de uso.

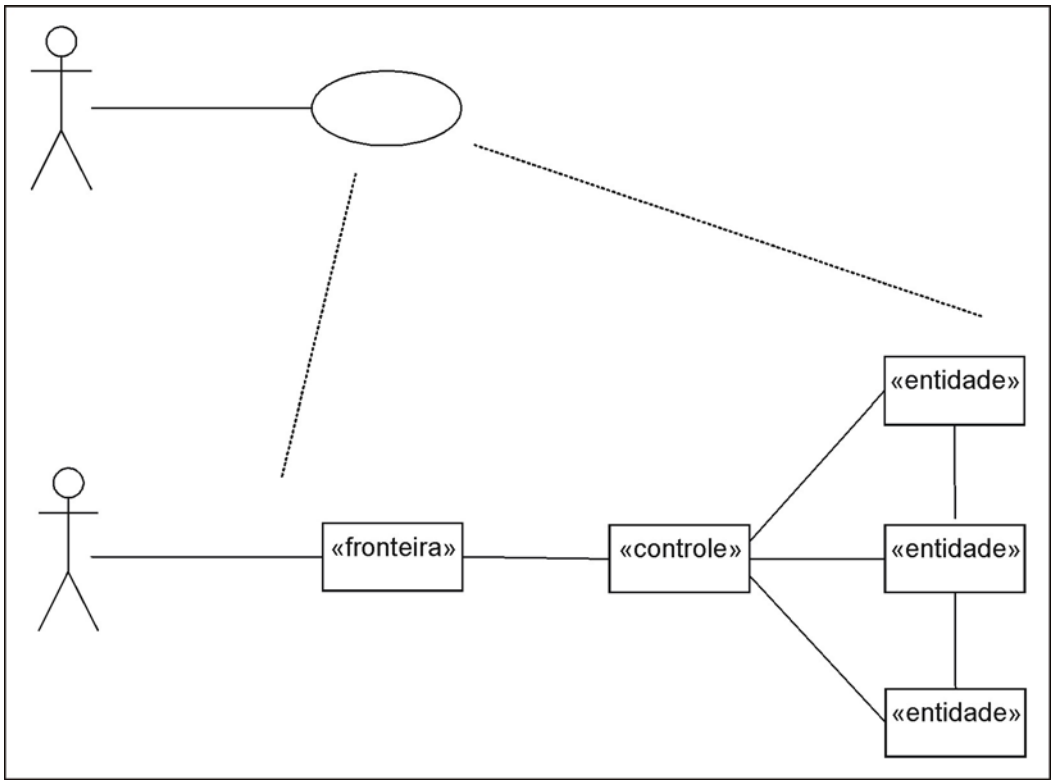


Figura 30.

O nosso trabalho é descobrir as classes e como os respectivos objetos colaboram para que o caso de uso seja realizado.

Para descobrir as classes:

1. Identificar as classes de interface

- Leia a descrição do caso de uso e conte quantas vez aparecem telas diferentes no caso de uso.
- Para cada tela, crie uma classe de fronteira.

2. Identificar classe de controle

- Crie uma classe de controle com o nome do caso de uso justaposto. Exemplo: se o nome do caso de uso é “Efetuar matrícula”, o nome da classe de controle é EfetuarMatricula.

3. Identificar classes de entidades

- Os substantivos que aparecem no texto do caso de uso são destacados.
- São também consideradas locuções equivalentes a substantivos.
- Remover sinônimos.
- Analisar as regras de negócio associadas com o caso de uso.
- Verificar os relacionamentos entre classes.
- Remover associações redundantes.
- Atribuir responsabilidades a cada classe e verificar as dependências entre as classes.
- Eliminar ou criar outras classes aplicando os princípios para manter alta coesão e o baixo acoplamento.

Dicas

- Associar responsabilidades com base na especialidade da classe.
- Distribuir a inteligência do sistema.
- Agrupar as responsabilidades conceitualmente relacionadas.
- Evitar responsabilidades redundantes.

Exemplo de análise de caso de uso

Seja o caso de uso abaixo, de um sistema acadêmico:

[CSU01] Fornecer Disponibilidade

Sumário: Professor fornece a sua grade de disponibilidade (dia e horários) e disciplinas que deseja lecionar no próximo semestre letivo.

Ator primário: Professor.

Ator secundário: N/A.

Precondições: O professor e as disciplinas estão cadastrados no sistema.

Fluxo Principal

1. O professor escolhe a opção “Fornecer disponibilidade”.
2. O sistema apresenta a lista de disciplinas disponíveis para as quais o professor está habilitado e uma grade de dias e horários da semana em branco.
3. O professor seleciona as disciplinas que deseja lecionar e preenche a grade com os horários que deseja lecionar.
4. O sistema registra as informações fornecidas e exibe a mensagem: “Dados gravados com sucesso.”

Fluxo alternativo

1A. No passo 3, o professor escolhe a opção de apresentar a mesma grade do semestre atual. O sistema apresenta a configuração requisitada. Volta ao passo 3.

Fluxo Exceção

1E. Se o professor não selecionou uma disciplina ou não preencheu as grades de disponibilidade, o sistema emite uma notificação ao professor e volta ao passo 2.

Pós-Condições: O sistema registrou a disponibilidade do professor para o próximo semestre letivo.

Iniciando a análise pelas responsabilidades já podemos identificar as seguintes classes:

1. **Classes de fronteira**
   - TelaGradeHorario
   - TelaConfirmação
   - TelaExceção
2. **Classe de controle**
   - CasdatrarDisponibilidade (mesmo nome do caso de uso)
3. **Iniciando a análise das classes de entidade, vamos marcar todos os substantivos (do fluxo principal)**
   - O **professor** escolhe a opção “Fornecer **disponibilidade**”.
   - O sistema apresenta a lista de **disciplinas** (nome e código de cada uma) disponíveis e uma grade de **dias** e **horários** da semana em branco.
   - O professor seleciona as disciplinas que deseja lecionar e preenche a grade com os horários que deseja lecionar.
   - O sistema registra as informações fornecidas.

Analisando cada uma das classes candidatas:

Professor: esta classe faz sentido, pois é sobre quem estamos falando no caso de uso. Suas informações sobre disponibilidade é que serão gravadas. Outras informações sobre professor já estão no sistema, tais como nome, cpf, endereço, formação.

Disciplina: esta classe também faz sentido, já que estas devem estar gravadas no sistema e possuem várias características: nome, código, carga horária e descrição (embora as últimas duas não estejam explícitas no caso de uso).

Disponibilidade: esta classe também faz sentido, pois é onde vamos registrar cada dia e horário que um professor pode dar aula. Se escolhêssemos colocar esta informação em Professor quebraríamos o princípio de coesão, já que Disponibilidade é uma informação relacionada a Professor mas não é essencial ao Professor (como nome, por exemplo), pois ela pode estar ou não registrada no sistema.

Assim temos as seguintes classes (visão por responsabilidade):

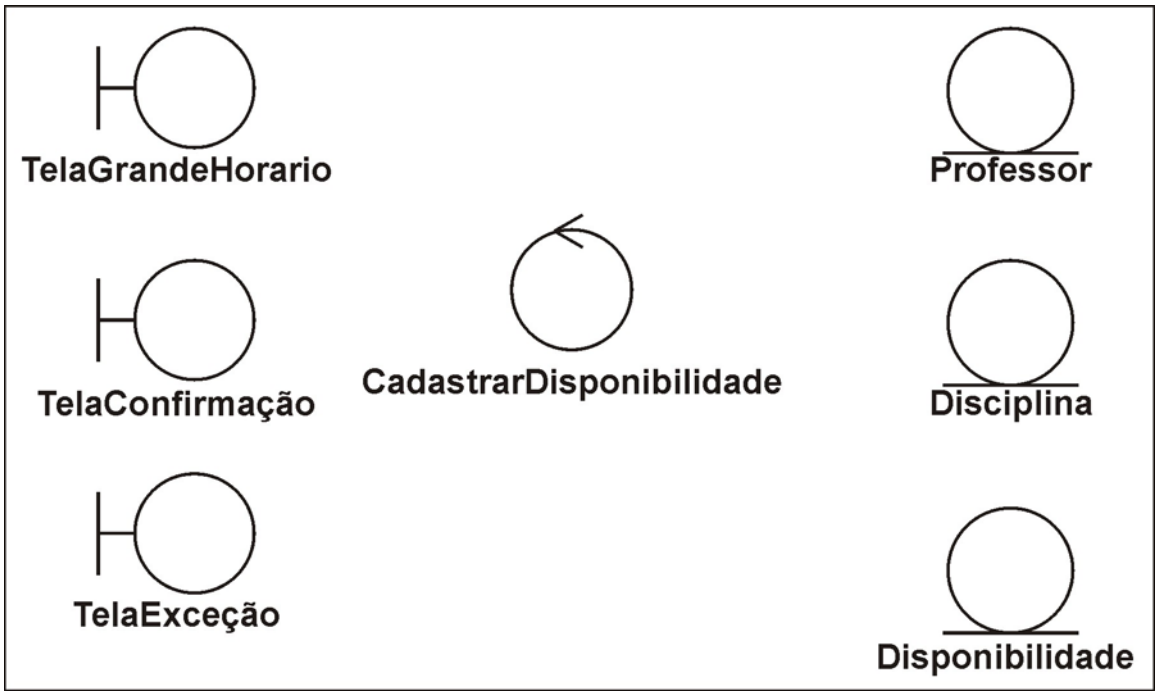


Figura 31.

Agora precisamos analisar a ligação (relacionamento de associação) entre as classes.

A classe de controle CadastrarDisponibilidade é quem controla o fluxo de execução do caso de uso, logo ela sabe a hora de mostrar cada tela e de salvar cada classe. Assim as classes de fronteira são ligadas com a classe de controle.

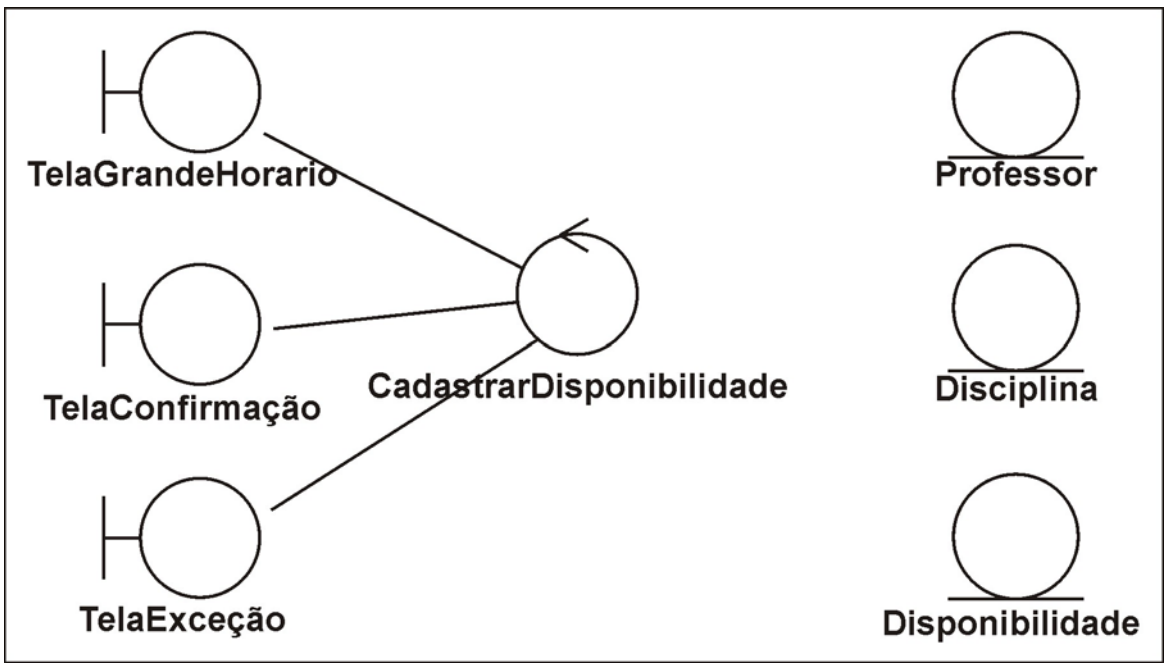


Figura 32.

Além disso, a classe de controle deve trazer a lista de disciplinas disponíveis. Logo deve ter acesso a estes objetos. Também deve saber quem é o professor que está acessando o sistema naquele momento. Além disso, o resultado do caso de uso é criar as disponibilidades (objetos) associados ao professor. Logo Disponibilidade e Professor devem estar ligadas. E mais, o professor escolhe as disciplinas que deseja lecionar. Logo, Professor e Disciplina também devem estar ligadas.

Assim o diagrama de classes com divisão de responsabilidades é:

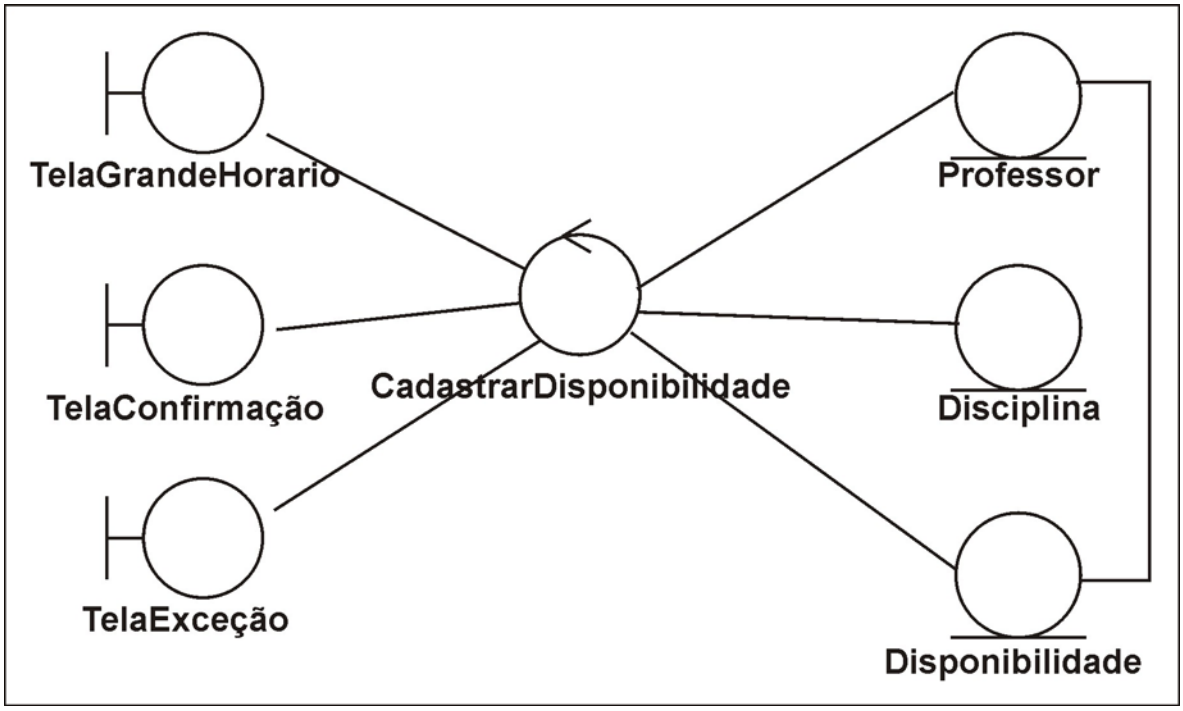


Figura 33.

Podemos detalhar as classes de entidades com seus atributos e métodos acessores:

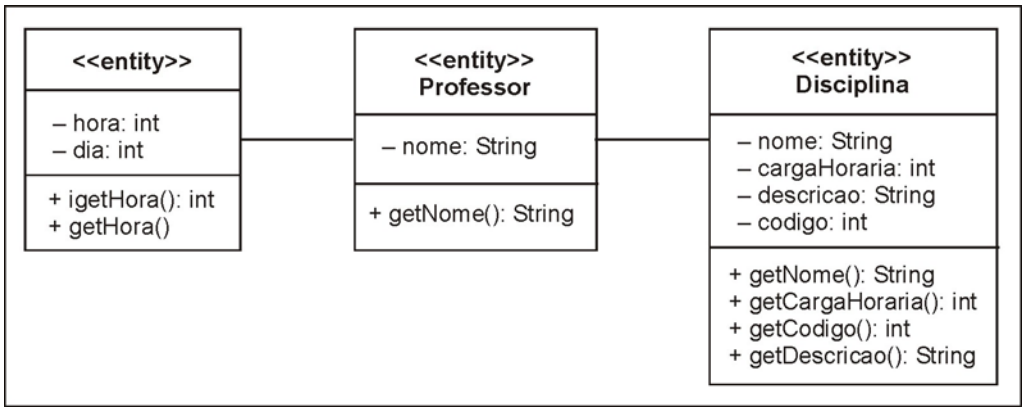


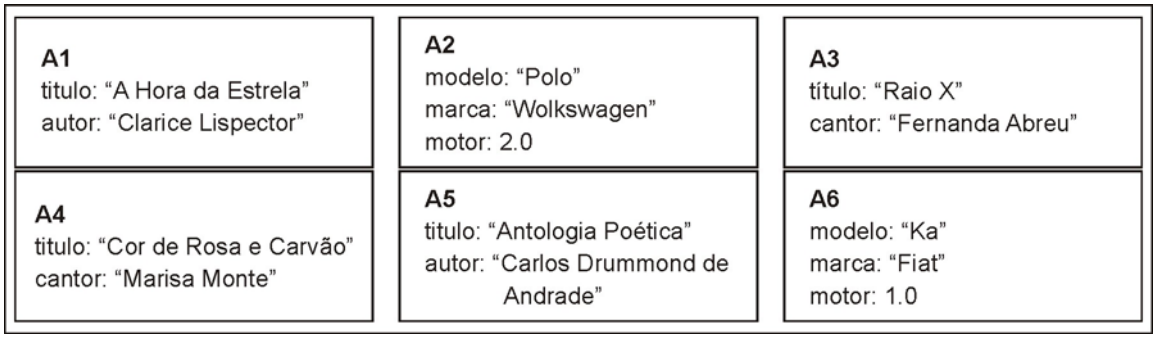
Figura 34.

Os métodos acessores permitem que possamos acessar o valor dos atributos. Como os atributos são privados, não podemos diretamente acessar o valor, por exemplo do nome de um professor. Assim, colocamos um método para que o nome seja devolvido pelo objeto.

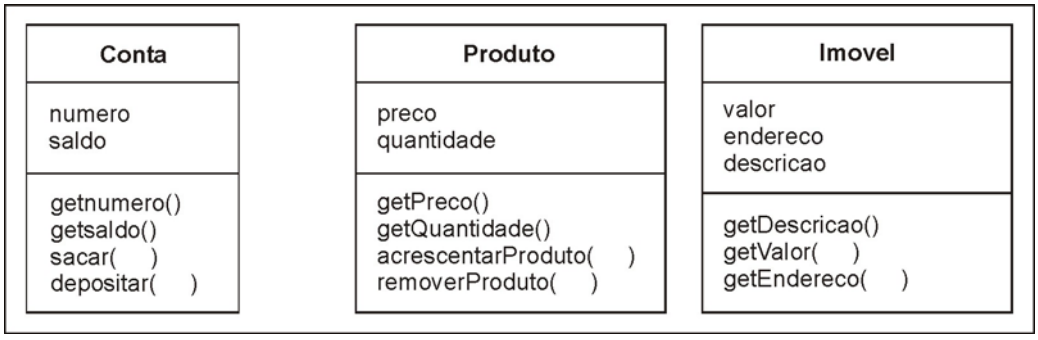
O nome dos métodos acessores possuem uma convenção: palavra get + <Nome do atributo>, ou seja, se o atributo é nome, o nome do método para acessá-lo é getNome(), além disso ele devolve o valor do nome que é do tipo String (texto). Logo, o método devolve um tipo String. Deve ser público para que os outros objetos possam acessá-lo. Ficando sua definição final assim: **+getNome():String.**

Exercícios

01. Dados os objetos sem tipo abaixo, crie classes que os agrupem.



02. Complete as classes abaixo com visibilidade, tipos, parâmetros e tipo de retorno para o método:



03. Analise os casos de uso abaixo, desenhando os diagramas de classe por responsabilidades e visualizando atributos e métodos.

[CSU1] Sacar

Sumário: o cliente faz uma retirada em dinheiro de sua conta

Precondição: O caixa deve ter dinheiro para fornecer ao cliente e o cliente deve estar identificado no sistema.

Fluxo principal:

1. O cliente escolhe a opção “sacar”.
2. O sistema solicita o valor que deve ser retirado da conta do cliente
3. O cliente fornece o valor desejado
4. O sistema verifica se há saldo e se há dinheiro na máquina e libera o dinheiro.

Fluxo alternativo:

N/A.

Fluxo de Exceção:

1E. No passo 4, se a conta do cliente não tem saldo suficiente uma mensagem é dada ao cliente e a operação é cancelada.

2E. No passo 4, se a máquina não possui dinheiro suficiente uma mensagem é dada ao cliente e a operação é cancelada.

Pós-condição: o valor solicitado pelo cliente é fornecido e o saldo da conta é diminuído do valor

[CSU2] Depositar

Sumário: o cliente deposita um valor em dinheiro de sua conta

Precondição: N/A

Fluxo principal

1. O cliente escolhe a opção “Depositar”.
2. O sistema solicita o número da conta e número da agência para depósito.
3. O cliente fornece os dados solicitados.
4. O sistema solicita o valor que deve ser depositado da conta do cliente.
5. O cliente fornece o valor a ser depositado.
6. O sistema abre a gaveta para introdução do envelope e registra o depósito.

Fluxo alternativo

1A. Nos passo 3 e 4, o cliente pode cancelar a operação.

Fluxo de Exceção

1E. No passo 4, se a conta do cliente não tem saldo suficiente uma mensagem é dada ao cliente e a operação é cancelada.

2E. No passo 4, se a máquina não possui dinheiro suficiente uma mensagem é dada ao cliente e a operação é cancelada.

Pós-condição: o valor depositado pelo cliente é registrado e acrescentado ao saldo da conta.

[CS3] Reservar Vôo

Resumo: Este caso de uso mostra a seqüência de interações para um cliente de uma companhia aérea realizar a reserva de um vôo

Ator: Cliente

Precondições: cliente está autenticado no sistema.

Fluxo principal

1. O cliente escolhe a opção “Reservar vôo”.
2. O sistema exibe duas listas de cidades origem e destino, e solicita a data do vôo e número de lugares.
3. O cliente escolhe a cidade origem, a cidade destino, digita a data do vôo e o número de lugares.
4. O sistema exibe a lista de vôos, com preço, horário de partida, horário de chegada e número de escala e/ou conexões e solicita que o cliente selecione uma opção.
5. O cliente seleciona o vôo desejado.
6. O sistema exibe o número da reserva e mostra a data de validade da reserva.

Fluxo alternativo

1A. Nos passos 3-5, o cliente pode cancelar a operação.

Fluxo de Exceção

1E. No passo 4, o sistema não encontra nenhum vôo para a origem, o destino e a data especificada. O sistema mostra a mensagem “Não foram encontrados os resultados” e mostra a opção de fazer uma nova tentativa.

Pós-condição: o sistema tem uma nova reserva no nome do cliente.

Bibliografia

BEZERRA, Eduardo. Princípios de análise e projeto de sistemas com UML. Editora Campus. Rio de Janeiro 2002.

LARMAN, Craig. Utilizando UML e Padrões. Bookman Companhia Editora, 2ª. Edição. Porto Alegre, 2004.

JACOBSON, Ivar, Christenson, M. Jonsson, P. e Övergaard, G. Object-Oriented Software Engineering: A Use Case Driven Approach. MA: Addison-Wesley, 1992.

Artigo: Para que utilizar a Análise Orientada a Objetos ?

Prof. Sérgio Neves de Souza (Fonte: http://www.fiap.com.br/portal/int_cda_conteudo.jsp?ID=15145)

Temos visto muitos comentários acerca das possíveis vantagens de se trabalhar com Orientação a Objetos, e a pergunta é: Quais as reais vantagens de se trabalhar orientado a objetos? Porque, se esta pergunta não for respondida de forma a ficar bem claro na mente de quem se acha conhecedor do assunto, corremos o risco de mudar a batedeira para fazer o mesmo bolo (ou como diria minha querida avó, trocaremos seis por meia dúzia). O que vemos por aí é a exagerada ênfase que o mercado tem dado às ferramentas de desenvolvimento e muito pouca atenção às técnicas de desenvolvimento modernas, e isso não apenas nos referindo a especificações J2EE ou ambientes de desenvolvimentos como *.Net*. Vemos uma série de novos produtos e *frameworks* que irão “Facilitar o desenvolvimento”, e realmente o fazem, pois é muito mais fácil aprendermos, por exemplo, *Struts* do que aplicarmos as especificações MVC (que não são simples) em algum projeto. Para conhecer alguma novidade, basta ir a algum desses encontros de comunidades de desenvolvimentos e nos maravilharemos (ou nos assustaremos) com a quantidade de novidades que aparecem a cada encontro. Pena que a visão de como fazer, pensando-se em larga escala ou em outras possibilidades, vai se perdendo e fica cada vez mais clara a idéia da especialização, ou seja, conhecemos pouco sobre desenvolvimento e muito sobre ferramentas.

Chegamos, finalmente, no ponto em que o tema deste artigo é colocado em análise, e não é uma análise superficial; é necessário saber para onde estamos caminhando ou, ao menos, para onde está direcionado o nosso foco ou objetivo atual (diríamos até mesmo maturidade de desenvolvimento). Vamos iniciar por você, isso mesmo, você que está lendo este texto sabeque, ao aprender uma nova linguagem ou mesmo uma nova implementação na linguagem que já utiliza, está aprendendo a aplicar a análise orientada a objetos com isso? Nunca pensou nisso? Não sabe para que serve isso? Para deixar mais claro o assunto, vamos fazer uma pequena divisão. Existem:

1. Análise Orientada a Objetos (AOO)
2. Programação Orientada a Objetos (POO)

Se eu sei POO (realmente) e, nesse caso, não importa a linguagem, posso fazer isso utilizando meus conhecimentos da linguagem, suas implementações já existentes, os *frameworks* que facilitam o desenvolvimento e padrões de desenvolvimento para implementações um pouco mais elaboradas. O problema é que me restrinjo a simples implementação, se não conheço AOO, e por mais que eu conheça, a fundo, a linguagem com que trabalho, dificilmente terei uma visão global sobre o sistema como um todo. É como ter um prédio para construir e eu saber muito sobre como construir um quarto desse prédio. Neste caso, alguns vão dizer que podemos utilizar *designer patterns* para facilitar a implementação; concordo, desde que seja para padrões de desenvolvimento já existentes e problemas gerais comuns em aplicações. Fico pensando sobre o que não é padrão como as regras de negócio das empresas, que são específicas demais para serem padronizadas; e é justamente aí que mora o perigo, pois quando temos programadores demasiadamente especialistas em ferramentas e pouco afeitos a análise, vemos a especialização da codificação, onde o programador que desenvolve as regras de negócio torna-se, em muitas empresas, o intocável, pois só ele sabe mexer em tal programa e ninguém mais da empresa se atreve ou tenta tal cometimento. Quem não concorda comigo basta olhar como anda a área de desenvolvimento em muitas organizações, onde se criam pais e mães de sistemas, e ninguém mais pode pôr a mão neles, pois só quem fez é que sabe como é... e se o programador ficar gripado...a empresa pára!

Em vista disso é que batemos na tecla da revitalização da AOO, onde as regras, que não se prendem a padrões específicos, são devidamente documentadas em diagramas de fácil visualização, onde o desenvolvimento (com sua devida importância) obedece, isso sim, a padrões estabelecidos pela empresa, e não são criados a partir de um lampejo criativo do programador (nada contra a criatividade), estabelecendo regras que só ele entende e onde ficaria cada vez mais difícil substituí-lo. Perde a empresa por depender excessivamente de uma pessoa e perde a pessoa por tornar-se pai ou mãe de um sistema, sabe-se lá por quanto tempo.

E olha que nem entramos nas reais vantagens da OO, como a reutilização e não o reaproveitamento de código que muitos usam por aí, onde a idéia é criar-se componentes de programas cada vez menores, e isso é um problema à parte, pois o costume de muitos em programação estruturada cria entraves nesse tipo de abordagem. Boa parte de quem programa OO ainda acredita que criar componentes mínimos, para facilitar o reúso, deixa o sistema lento ou coisas parecidas, quando o ideal seria criar-se pequenos componentes, com o mínimo de tarefas. Assim, teríamos algo parecido com o jogo LEGO, onde poderíamos montar diferentes projetos de sistemas com os mesmos componentes, mas isto já é assunto para um outro tema.