

Universidade Estadual de Campinas



Curso - Conceitos de Orientação a Objetos

Autor: Denis Clayton Alves Ramos
Arquiteto de Sistemas CCUEC - Unicamp

Sumário

I. Introdução ao paradigma da orientação a objetos.....	5
1. Breve histórico.....	5
2. Vantagens inerentes a este paradigma.....	7
II. Introdução a UML.....	11
1. UML, como surgiu esse padrão?.....	11
2. Objetivos da UML.....	11
3. Principais diagramas da UML.....	12
4. Introdução ao diagrama de classes.....	13
III. Introdução à criação de classes.....	17
1. Classes e Objetos.....	17
2. Atributos.....	20

3. Operações (Métodos).....	22
4. Sobrecarga de métodos.....	23
5. Construtores.....	25
6. Encapsulamento.....	27
7. Empacotamento.....	27
IV. Relacionamentos entre classes.....	29
1. Associação.....	29
2. Associações Avançadas.....	30
V. Polimorfismo.....	36
VI. Classe Abstrata.....	41
VII. Interface (Herança Múltipla em JAVA).....	42
VIII. Padrões de projetos.....	45
1. O que é um padrão de projeto?.....	45
2. O que um padrão de projeto não é?.....	46
3. Catálogo de padrões de projetos.....	46

Pré-requisitos:

Conhecimentos em desenvolvimento de Sistemas de Informação

Conhecimento a adquirir:

Aprendizado dos conceitos fundamentais de orientação a objetos

Noções dos princípios da programação orientada a objetos em Java

Visão geral dos fundamentos de UML

Noções gerais sobre conceitos de padrões de projetos

I. Introdução ao paradigma da orientação a objetos

1. Breve histórico

O PROGRESSO DA ABSTRAÇÃO

Todas as linguagens de programação trazem inerente a sua forma de trabalhar e propósito **o recurso da abstração** para a resolução de problemas.

Uma *variável* é uma abstração (ou representação) de um tipo de dado cujo valor é referenciado por um endereço de memória.

Uma *procedure* é uma abstração para a resolução de uma parte específica de um problema.

Definição do verbo abstrair: **1.** Considerar isoladamente (coisas que estão unidas). **2.** Separar, apartar. **3.** Não levar em conta; pôr de parte. **4.** Alhear-se, distrair-se. **5.** Concentrar-se, absorver-se. (*Minidicionário Aurélio*)

A linguagem de programação *Assembly* é uma abstração reduzida do hardware no qual ela executa.

Muitas linguagens de programação que sucederam a criação da *Assembly* (*FORTRAM*, *BASIC* e *C*) são abstrações de alto nível da própria *Assembly*. São linguagens que apresentam uma sintaxe mais próxima da linguagem humana para facilitar a resolução de problemas.

Contudo, essas linguagens ainda requerem **fortemente** que a solução do problema seja abordada (abstraída) em termos da estrutura do computador como: ponteiros, registradores, estruturas de dados etc.

Enfim, o desenvolvedor é obrigado a estabelecer a *associação* entre o modelo de funcionamento do hardware (o ambiente onde o problema está sendo modelado – o computador) e o modelo do problema que está sendo solucionado (o ambiente onde o problema existe).

O esforço requerido para fazer esse mapeamento (associação entre os modelos discrepantes) resulta em **programas que são difíceis de codificar e mais difíceis ainda para se manter**.

Algumas linguagens introduziram abordagens próprias, isto é, romperam com a abordagem da solução centrada na estrutura da máquina.

LISP optou por uma visão particular do mundo para resolução de problemas – “Todos os problemas são em última instância listas”.

PROLOG transforma todos os problemas em cadeias de decisões.

Entretanto, essas linguagens eram restritas a domínios de problemas específicos, problemas fora de tais domínios demandavam soluções impraticáveis.

O paradigma de orientação a objetos foi um avanço pois viabiliza ferramentas para o desenvolvedor propor soluções a partir dos próprios elementos do domínio do problema.

Neste paradigma a solução pode ser representada de forma genérica o suficiente de maneira que o desenvolvedor não fica restrito a nenhum tipo de problema específico, como nos casos de *LISP* e *PROLOG*.

A abstração obtida por este paradigma é expressa na forma de “objetos”. A idéia é que os programas orientados a objetos sejam capazes de adaptar-se a especificidade do problema por meio da adição de novos tipos de objetos.

Dessa forma, quando o desenvolver lê o código que descreve a solução do problema, ele está lendo palavras que de fato expressam o problema. Essa abstração é mais flexível e poderosa do que a abstração proposta pelas linguagens anteriormente citadas.

Portanto, o paradigma de orientação a objetos permite que a solução do problema seja descrita em termos do problema em si, e não em termos do ambiente computacional onde a solução será executada.

Todavia, ainda existe uma conexão com o ambiente computacional: cada objeto assemelha-se um pouco com um pequeno computador – ele possui estado e operações. Contudo, essa conexão não parece ser tão ruim na medida que no mundo real todos os objetos também apresentam atributos (estado) e comportamentos (operações).

2. Vantagens inerentes a este paradigma

O paradigma de orientação a objetos não se trata tão somente de um recurso a mais das novas linguagens de programação. Este novo paradigma **afeta** desde a *Análise*, passando pelo *Projeto (Design)* e concretizando-se na *Implementação*.

Vários "papas" da engenharia de software como *Peter Coad*, *Edward Yourdon* e *Roger Pressman* abordaram extensamente a análise orientada a objetos como sendo um grande avanço no desenvolvimento de sistemas.

Os principais pontos que eles abordaram e definiram em suas publicações foram que:

- A orientação a objetos é uma tecnologia para a produção de **modelos que especifiquem o domínio do problema de um sistema**
- Quando construídos corretamente, sistemas orientados a objetos são **flexíveis a mudanças**, possuem estruturas bem conhecidas e provêem a oportunidade de criar e implementar componentes com alto grau de **reutilização**
- Modelos orientados a objetos são implementados utilizando uma linguagem de programação orientada a objetos. A engenharia de software orientada a objetos é muito mais que utilizar mecanismos de uma linguagem de programação, na verdade, constitui-se em saber utilizar da melhor forma possível todas as técnicas da modelagem orientada a objetos

A análise orientada a objetos:

- ✓ Determina o que o sistema deve fazer: Quais os atores envolvidos? Quais as atividades a serem realizadas?
- ✓ Decompõe o sistema em objetos: Quais são? Que tarefas cada objeto terá que fazer?

O design orientado a objetos:

- ✓ Define como o sistema será implementado
- ✓ Modela os relacionamentos entre objetos e atores
- ✓ Utiliza e reutiliza abstrações como classes, objetos, funções, frameworks, APIs, padrões de projeto

Vocabulário

API (Application Programming Interface): o propósito primordial de uma API é descrever como acessar um conjunto de funções. Por exemplo, uma API que descreve como aplicações podem se comunicar com SGBDs - estabelecer conexão, realizar pesquisas em tabelas etc. A API é apenas um conjunto de definições (interfaces ou classes

abstratas). Para se comunicar de fato com um SGBD existe a API JDBC (Java Database Connectivity) e cada SGBD provê uma implementação de JDBC (um software).

Framework: é uma estrutura de software para suporte, na qual outros projetos de software podem se organizar e se desenvolver. Um framework pode incluir bibliotecas de código, linguagem de script e outros softwares que cooperam no desenvolvimento e na cola de diferentes componentes de um projeto de software

PARADIGMA DA ORIENTAÇÃO A OBJETOS VERSUS PARADIGMA PROCEDURAL

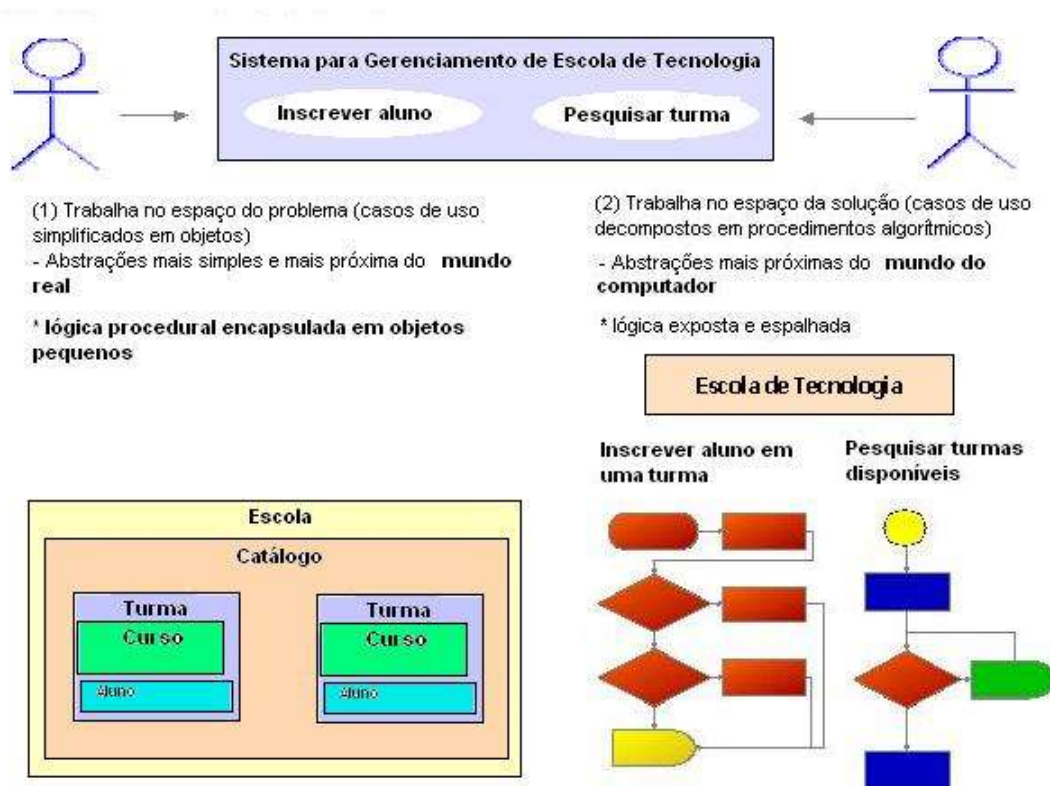


Figura 1 - Abstração de Casos de Uso em Análise OO (1) e Análise Procedural (2)

II. Introdução a UML

1. UML, como surgiu esse padrão?

Grady Booch, Ivar Jacobson e James Rumbaugh trabalhavam isoladamente cada qual desenvolvendo sua própria metodologia para análise e projeto de software orientado a objeto nos anos 80 e começo dos anos 90. Cada um deles é considerado um renomado pesquisador no tema de orientação a objetos e metodologia de desenvolvimento de software orientado a objetos. A partir da segunda metade dos anos 90 surge a iniciativa de fundir as três metodologias mantendo o que havia de melhor em cada uma dessas.

A versão 1.0 da UML aglutinou grandes nomes do mercado como Oracle, HP e Rational na sua confecção realizada em 1997. A versão 1.1 foi submetida para OMG (*Object Management Group*) que adotou a UML como linguagem de modelagem de software padrão.

2. Objetivos da UML

Um modelo é uma simplificação da realidade e provê uma descrição completa desta sob uma perspectiva em particular. Modelos são construídos para prover um entendimento melhor de problemas complexos – construção de um prédio, sistema financeiro de Bolsas de Mercado Futuro.

A modelagem é importante porque pode auxiliar a equipe de desenvolvimento a *visualizar, especificar, construir e documentar* a estrutura e o comportamento da arquitetura de um sistema.

Assim, uma linguagem de modelagem padrão como a UML (*Unified Modeling Language*) garante que os vários membros da equipe de desenvolvimento possam comunicar suas decisões de forma **não ambígua** entre si.

Os objetivos da UML são:

- Modelar sistemas usando os conceitos da orientação a objetos
- Promover um vocabulário único entre os integrantes da equipe resultando em benefícios no entendimento e comunicação efetiva entre as partes
- Criar uma linguagem de modelagem útil tanto pelo homem quanto pela máquina (é possível gerar código a partir de alguns modelos UML; a engenharia reversa também é possível)

3. Principais diagramas da UML

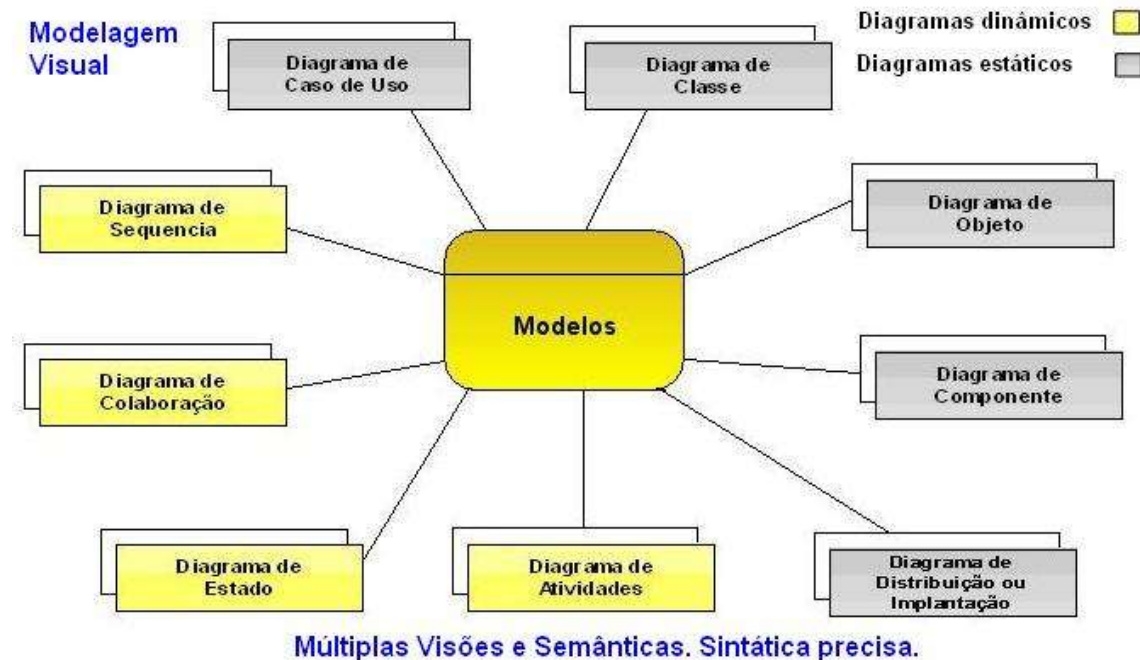


Figura 2 - diagramas da UML

Na construção de um sistema, diferentes diagramas são necessários para representar as diferentes visões deste sistema. A UML fornece uma notação adequada para cada uma destas visões. A notação UML inclui os seguintes modelos (visões):

- Diagrama de casos de uso (use-cases) para ilustrar as interações dos usuários dentro do sistema
- Diagramas de classe para ilustrar a estrutura lógica
- Diagrama de objetos para ilustrar os objetos e seus relacionamentos
- Diagrama de estado para ilustrar comportamentos
- Diagrama de componente para ilustrar a estrutura ou organização "física" do software
- Diagrama de distribuição ou implantação (deployment) para mostrar o mapeamento do software e a configuração de hardware para comportá-lo
- Diagrama de interação (colaboração e de seqüencia) para ilustrar comportamento
- Diagrama de atividade para mostrar os fluxos de eventos

4. Introdução ao diagrama de classes

Este diagrama tem como finalidade descrever as classes de objetos presentes no sistema sendo modelado. A partir das classes identificadas, suas características – atributos e comportamentos são mapeados bem como os relacionamentos existentes entre essas classes.

O diagrama de classes é classificado como estático pois especifica quais entidades interagem, mas não especifica o que acontece quando ocorre a interação.

Exemplo de diagrama de classes:

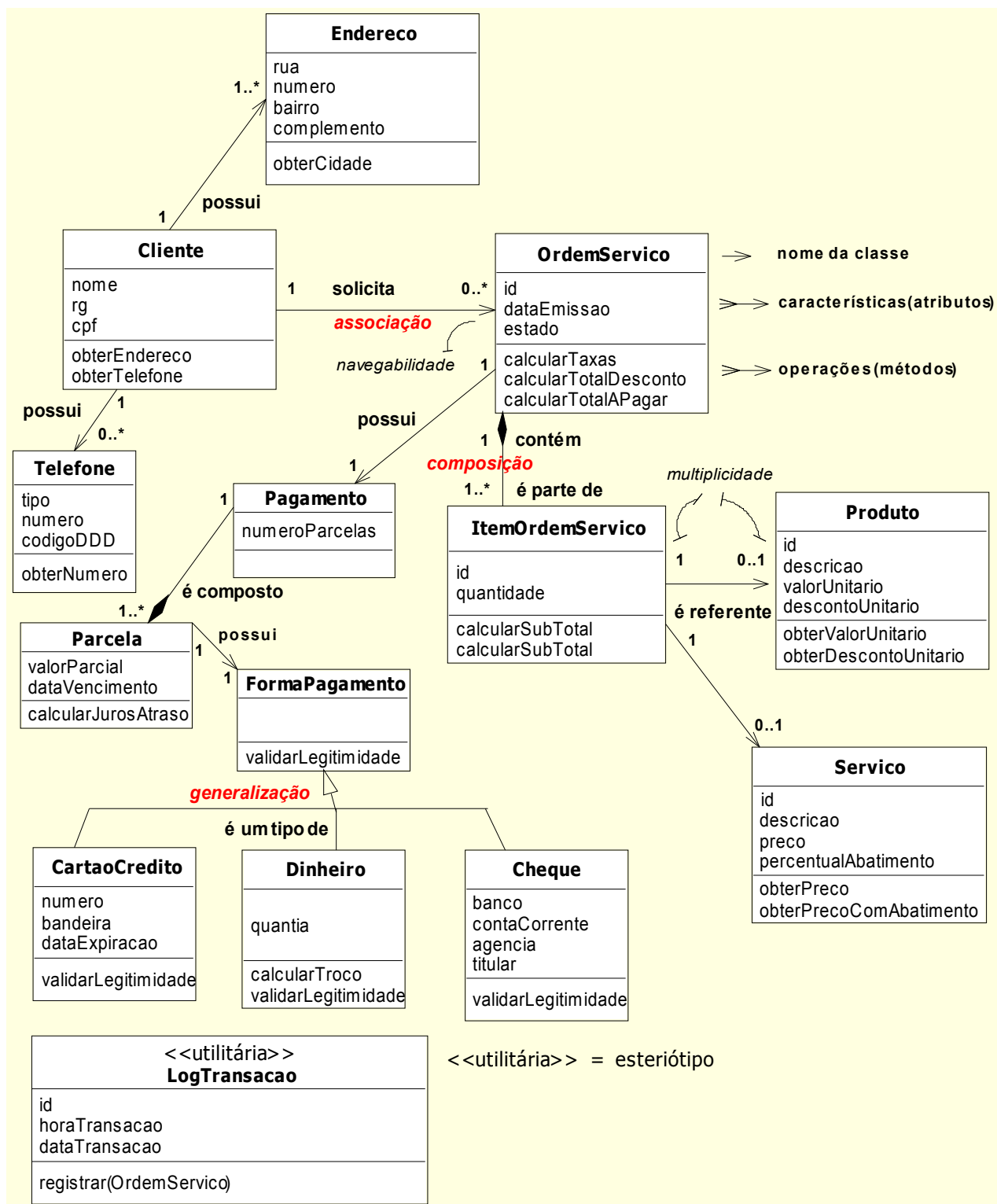


Figura 3 – Ordem de serviço de um cliente simplificada

A notação UML para classe pode ser observada a partir da entidade *OrdemServico* onde a primeira parte do retângulo é destinada para identificar o estereótipo da classe caso seja interessante e o nome da classe. A segunda parte destina-se a definição de atributos e a última é para os métodos.

III. Introdução à criação de classes

1. Classes e Objetos

Classe em orientação a objetos é definida tanto como uma **representação de uma entidade do mundo real** como uma **representação de uma entidade que faz parte do domínio da solução**.

O termo mundo real está sendo usado como sinônimo de *domínio do problema* ou *domínio do negócio* a ser representado e sistematizado – Cliente, Ordem de Serviço, Produto, Serviço, Forma de Pagamento etc.

Já o termo domínio da solução faz menção a aspectos inerentes as linguagens de programação para resolução de questões computacionais. Por exemplo, classe utilitária responsável por gravar Log das ordens de serviço. Esta classe não é parte do negócio sendo modelado, mas trata de um requisito não funcional do sistema.

Classe, em um sentido mais técnico, é um **tipo de dado estruturado especial**, pois agrega **características**: informações peculiares, denominadas atributos como **comportamentos** (métodos ou operações) que a identifica.

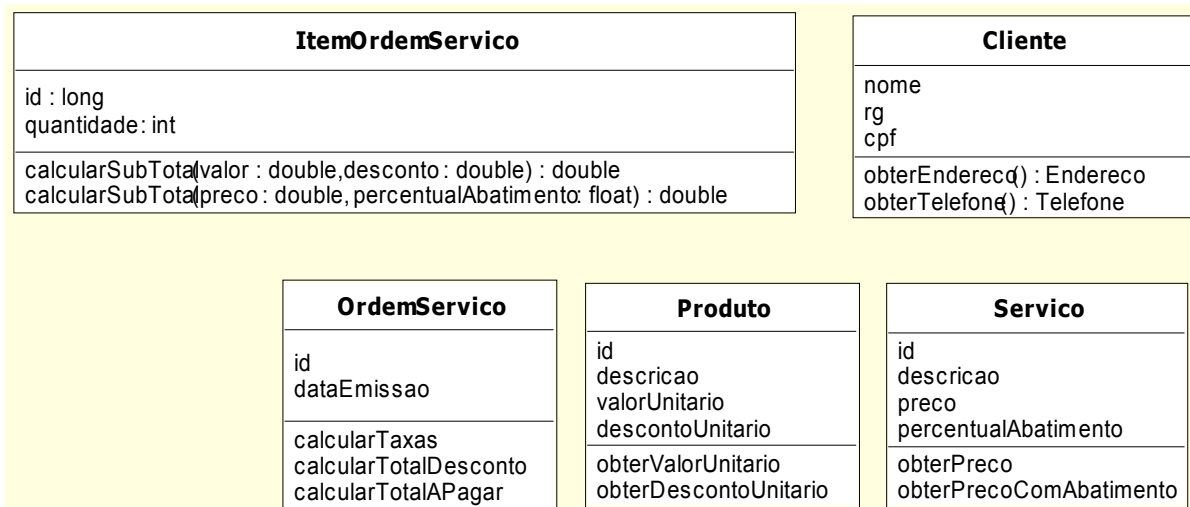


Figura 4 - Classes com suas características: atributos e comportamentos

Observe que cada classe acima representa um e somente um conceito apenas.

Por exemplo, a classe cliente poderia ter atributos relacionados a endereço e telefone.

Contudo, mantê-los como características intrínsecas à entidade Cliente não resulta em uma separação ideal de informações e conceitos.

Dessa forma, como mostrado na figura 3, duas entidades (classes) – Endereço e Telefone – estabelecem associações com a classe Cliente resultando uma modelagem mais organizada em termos de separação de conceitos.

Além disso, essa modelagem facilita trabalhar o aspecto da multiplicidade. Essa modelagem permite que o cliente indique mais de um telefone e/ou mais de um endereço.

Qual a relação entre classes e objetos?

Classes servem de **molde** ou **especificação** para objetos.

Assim, os objetos do tipo de uma determinada classe são obrigados a implementar a especificação (molde) desta, ou seja, devem apresentar as mesmas características: informações (atributos) e comportamentos (operações).

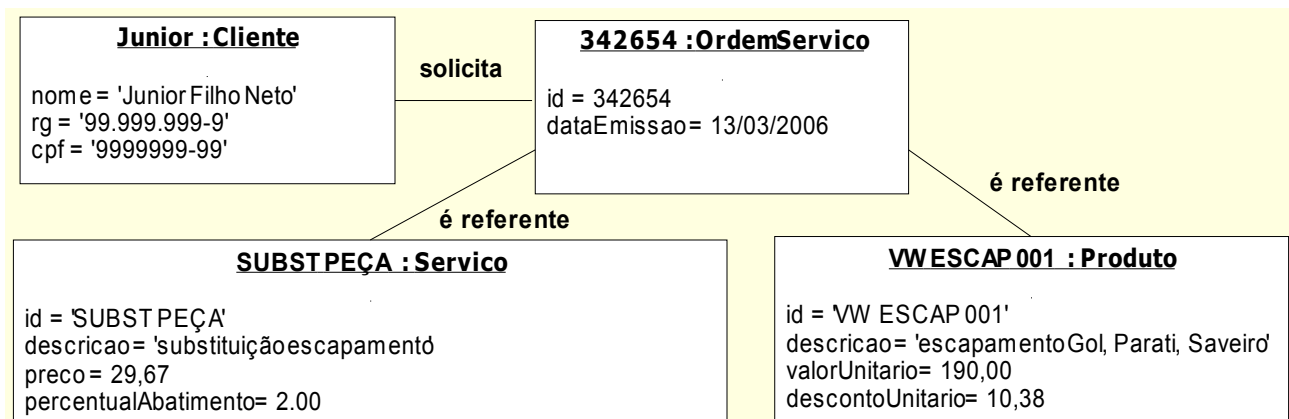


Figura 5 – Diagrama de Objetos

Objetos **possuem características** que eles conhecem (atributos) e **sabem realizar determinadas ações** (comportamentos ou operações). Os valores assumidos pelos atributos de um objeto determinam seu **estado**.

Objetos são denominados **instâncias de classes** e possuem **vida independente** entre si, apesar de **compartilharem o mesmo “molde”** (Classe).

Todos os objetos possuem identidade e, portanto, são distinguíveis.

Identidade significa que os objetos são diferenciados uns dos outros pela sua própria existência e não pelas propriedades descritivas que estes possam ter.

Objetos podem ser diferenciados uns dos outros pelo conjunto de valores que seus atributos podem assumir, porém, mais precisamente, são identificados de maneira unívoca pela sua referência de memória; cada objeto recebe um endereço distinto na pilha de memória.

Um objeto pode representar algo **concreto** como um avião, um computador ou pode representar também um **conceito** como um processo químico, uma transação bancária, uma ordem de compra.

Objetos servem a dois propósitos:

- promover o entendimento do mundo real; e
- viabilizar uma representação deste que seja prática para sua implementação pela computação

Vocabulário

O termo **classe de objetos** descreve um grupo de objetos de mesma classe, ou seja, tais objetos compartilham entre si as mesmas propriedades ou características, *os mesmos relacionamentos com outros objetos; enfim, compartilham a mesma semântica.*

Objetos e classes de objetos são freqüentemente encontrados como **substantivos** em descrições de problemas.

O termo **instância de objeto** refere-se a um objeto em particular de uma determinada classe.

Tendo essas definições, a notação UML do diagrama de objetos (vide figura 5) consiste em :

- cada retângulo representa uma instância de objeto
- o nome da instância deve ser sublinhado
- para efeito de clareza, a classe de objetos a qual essa instância pertence pode ser especificada a direita do nome do objeto.

Observação: os valores dos atributos de cada instância de objeto na figura 5 foram colocados para ilustração.

Exercício:

1) Construa uma representação parcial da sua árvore genealógica onde cada grau de parentesco representa uma pessoa da sua família. Utilize o diagrama de objetos conforme notação UML.

2. Atributos

Atributos **armazenam os valores** das informações do objeto.

Cada atributo é de um tipo de dado específico. As especificidades das implementações dos tipos de dados diversificam, em geral, conforme a linguagem de programação e/ou plataforma operacional em uso.

Por exemplo, em Java os tipos primitivos são definidos assim (independente da plataforma operacional na qual a aplicação esteja rodando):

Tipo primitivo (palavra reservada)	Tamanho / Formato	Descrição
byte	8 bits	Inteiro do comprimento de um byte
short	16 bits	Inteiro curto
int	32 bits	Inteiro
long	64 bits	Inteiro longo
float	32 bits IEEE 754	Ponto flutuante de precisão simples
double	64 bits IEEE 754	Ponto flutuante de precisão dupla
boolean	true ou false	Verdadeiro (true) ou Falso (false)
char	16 bits (caractere Unicode)	Um caractere

Além dos tipos primitivos a API de Java define uma série de tipos (objetos) próprios para representar por exemplo cadeias de caracteres (`java.lang.String`), array de bytes (`java.io.InputStream`) etc.

Também é possível definir (modelar) novos tipos de dados (objetos) os quais não são próprios da linguagem, mas estão associadas às necessidades da aplicação sendo construída.

Convenções para nomes de atributos:

Os nomes dos atributos são escritos em letras minúsculas, no singular e sem acentuação.

Por exemplo: nome, rg, cpf, descricao, preco.

Quando o nome do atributo for composto por duas ou mais palavras, a partir da segunda palavra, cada palavra nova deve iniciar com letra maiúscula; não deve haver nenhum caractere de separação.

Por exemplo: valorUnitario, descontoUnitario, dataEmissao.

Atributos de classe versus atributos de instância de objetos

Em Java, a palavra reservada *static* (estático) na frente da declaração de um atributo indica que este é um atributo de classe. Isso implica que esse atributo é compartilhado por todos os objetos dessa classe.

Assim, se um objeto alterar o valor desse atributo estático todos os demais objetos vão ter acesso ao novo valor.

Quando a palavra reservada *static* é omitida na declaração de um atributo significa que o mesmo é um atributo de instância, ou seja, o mesmo é criado para cada objeto instanciado e cada objeto pode alterar apenas seu próprio atributo de instância.

Atributos como constantes

A palavra reservada *final*, em Java, indica que o atributo assim definido não pode ter seu valor alterado. O valor atribuído na declaração de um atributo estático é imutável durante todo o ciclo de vida do objeto ao qual o atributo em questão faz parte.

3. Operações (Métodos)

Uma classe concentra um conjunto de responsabilidades para si e estas definem o comportamento dos objetos dessa classe. Essas **responsabilidades são materializadas na forma de operações** definidas pela classe.

Uma operação (método) deve tratar de uma e somente uma responsabilidade apenas.

Como todo objeto “conhece” sua classe assim tem acesso à implementação dos métodos definidos na classe da qual deriva.

A assinatura de um método consiste do nome deste, da quantidade de parâmetros e o tipo de cada parâmetro.

Convenções para nomes de métodos:

- os nomes dos métodos devem ser definidos com letra minúscula
- quando o nome do método for composto por duas ou mais palavras, a partir da segunda palavra, cada palavra nova deve iniciar com letra maiúscula; não deve haver nenhum caractere de separação
- usar o infinitivo na nomenclatura dos métodos: calcularTotalDesconto, validarLegitimidade, obterPrecoComAbatimento

Sintaxe para definição de métodos:

<tipo de dado do retorno> nomeDoMetodo () ou

<tipo de dado do retorno> nomeDoMetodo (<tipo de dado do parametro1>

nomeParametro1,...)

ItemOrdemServico
id : long quantidade: int
calcularSubTotal(valor : double,desconto : double) : double calcularSubTotal(preco : double, percentualAbatimento: float) : double

Figura 6 – Exemplo de assinatura de métodos

Relacionamentos e assinatura de operações

A assinatura de uma operação pode indicar um relacionamento.

Se o(s) argumento(s) de uma operação ou o retorno dela for uma classe do domínio do negócio, isso indica um relacionamento entre a classe que define a operação em questão e a classe que participa da operação como argumento ou retorno dela.

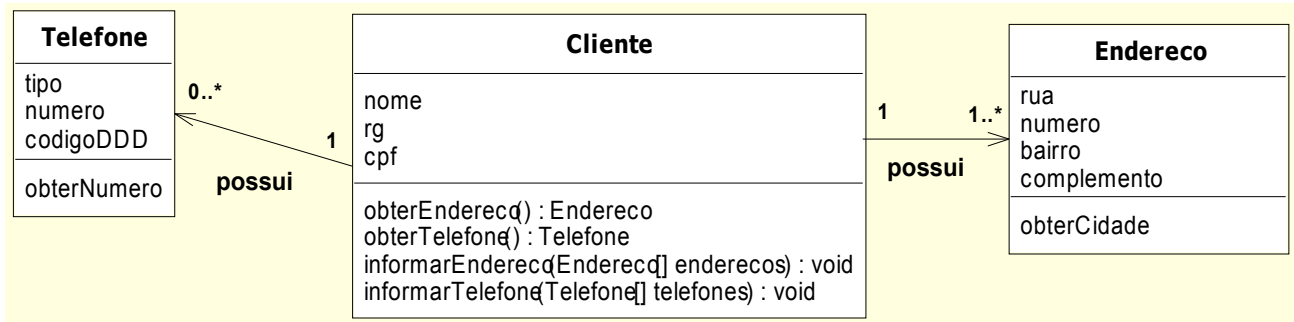


Figura 7 – Relacionamentos da classe Cliente

Métodos de classe versus métodos de instância de objetos

Em Java, a palavra reservada *static* (estático) na frente da declaração de um método indica que este é um método de classe. Isso implica que esse método pode ser acionado diretamente a partir da classe, não é necessário instanciar um objeto dessa classe para poder acionar o método estático.

4. Sobrecarga de métodos

Pode-se ter diversos métodos declarados com o mesmo nome dentro de uma mesma classe. Entretanto, tais métodos não podem ter os mesmos tipos de parâmetros na mesma ordem de declaração.

Na figura 6 vemos um exemplo de sobrecarga de métodos também denominada como *overriding*.

calcularSubTotal(valor : double, desconto : double) : double

calcularSubTotal(preco : double, percentualAbatimento : float) : double

Observação: nos métodos sobrecarregados os parâmetros podem apresentar o mesmo nome mantendo apenas a variação do tipo de dado.

Dessa forma, o segundo método poderia apresentar a seguinte assinatura:

calcularSubTotal(valor : double, desconto : float) : double

O primeiro método implementa o cálculo do subtotal para produtos a partir de um cálculo de subtração. É subtraído desconto (descontoUnitario) de valor (valorUnitario). Já o segundo método implementa o cálculo do subtotal para serviços a partir de um cálculo envolvendo percentual (percentualAbatimento).

Na sobrecarga, os métodos sobrecarregados, compartilham a mesma responsabilidade, mas cada um adota uma abordagem diferente para se obter a mesma solução.

Um método de instância de uma subclasse com a assinatura e tipo de retorno igual a um método de instância da superclasse sobrescreve este método da superclasse.

Uma subclasse não pode sobrescrever um método que é declarado como *final* na superclasse.

A habilidade de uma subclasse sobrescrever um método permite que o comportamento definido na superclasse para esse método seja alterado conforme as necessidades da subclasse.

A classe *java.lang.Object*, a classe raiz da hierarquia de classes da API de Java, define um método de instância denominado *toString*. Este método retorna uma string (texto) de representação de uma instância de objeto dessa classe.

Segue um exemplo de sobrecarga deste método:

```
package financeiro;
public class ItemOrdemServico {
    //atributos
    private long id;
    private int quantidade;
    public String toString() {
        return ("ItemOrdemServico = " + String.valueOf(id) + ", " + String.valueOf(quantidade));
    }
    ...
}
```

Como toda classe em Java é uma subclasse de *java.lang.Object*, o método *toString* da superclasse foi sobrescrito em *ItemOrdemServico* para retornar os valores de seus dois atributos.

5. Construtores

São métodos especiais chamados no processo de instanciação (**criação**) de um objeto. A execução de tais métodos garante a inicialização correta de um objeto.

Todo construtor tem o mesmo nome da classe e diferente dos demais métodos, não define tipo de retorno.

Também pode haver sobrecarga de construtores conforme segue (trecho do código de uma aplicação que usa a definição da classe *Cheque* para criar duas instâncias de objetos dessa classe – *chequeSemIdentificacao* e *chequeComIdentificacao*):

//1ª forma de instanciação definida pela classe Cheque

```
FormaPagamento chequeSemIdentificacao = new Cheque();
```

//2ª forma de instanciação definida pela classe Cheque, uso do construtor sobrecarregado

```
FormaPagamento chequeComIdentificacao = new Cheque(agenzia, banco, contaCorrente, titular);
```

Na primeira forma é acionado o construtor padrão que a API de Java implementa atribuindo valores nulos por exemplo para objetos do tipo *java.lang.String* como: *agenzia*, *banco* e

titular. Como o atributo *contaCorrente* é do tipo primitivo *int*, o valor 0 é atribuído na chamada do construtor padrão.

Já na segunda forma, os valores referenciados pelos parâmetros *agencia*, *banco*, *contaCorrente* e *titular* passarão a ser referenciados também pelos atributos do objeto *Cheque* sendo criado.

É necessário definir o seguinte trecho de código na classe *Cheque* para a segunda forma de instanciação mostrada acima.

```
package financeiro;
public class Cheque extends FormaPagamento {
    //atributos
    private String banco;
    private int contaCorrente;
    private String agencia;
    private String titular;

    public Cheque() {
        super ();
    }
    //construtor sobrecarregado
    public Cheque(String agencia, String banco, int contaCorrente, String titular) {
        super();
        this.agencia = agencia;
        this.banco = banco;
        this.contaCorrente = contaCorrente;
        this.titular = titular;
    }
    ...
}
```

A palavra reservada **this** indica que está sendo referenciado os atributos do objeto corrente. É uma forma de facilitar a leitura do código diferenciando os atributos do objeto dos parâmetros do

construtor sobrecarregado que foram nomeados da mesma forma.

A palavra reservada **super** será explicada mais adiante.

6. Encapsulamento

É a capacidade de restringir o acesso às características de uma classe utilizando modificadores de acesso. Um modificador de acesso define a visibilidade de um atributo ou método de uma classe em relação a demais classes / objetos de um sistema.

A tabela a seguir contém as palavras reservadas em Java para os modificadores de acesso definidos pela API de Java. A ordem ascendente dos modificadores representa o nível com maior acessibilidade para o nível de menor acessibilidade.

Modificador de acesso	Efeito na visibilidade ou acessibilidade de um atributo ou método
public	Acesso às características com esse modificador é totalmente livre
protected	Acesso às características com esse modificador é restrito a própria classe ou subclasses desta
package	Acesso às características com esse modificador é restrito a própria classe ou as classes que compõem o mesmo pacote
private	Acesso às características com esse modificador é permitido apenas de dentro da própria classe que as define

7. Empacotamento

Linguagens orientadas a objetos trabalham com o conceito de pacotes que representam uma outra forma de encapsulamento além daquela estabelecida por meio de modificadores de acesso.

Também é uma forma de estruturar ou organizar os conceitos que classes representam ao agrupá-las em pacotes por assunto (conceito) ou por outras motivações.

Por exemplo, a classe *Cheque* faz parte do pacote (palavra reservada **package**) **financeiro** que abrange todas as classes relacionadas ao assunto. Constituem esse pacote todas as subclasses de *FormaPagamento*.

A associação do conceito de pacotes com a idéia de uma árvore de diretórios facilita seu entendimento.

O pacote **financeiro** seria o diretório raiz e dentro dele há organizações como a subdivisão nos subdiretórios **formapagamento** e **ordenservico** para separar classes relacionadas ao assunto financeiro porém com propósitos distintos dentro desse assunto.

A organização de um sistema em pacotes com semântica clara e visando a modularização também é um recurso de orientação a objetos para a criação de componentes ou bibliotecas de componentes que resulta em reuso.

IV. Relacionamentos entre classes

Os relacionamentos definidos entre classes se propagam para as instâncias de objetos dessas classes.

1. Associação

Associação é uma forma de estabelecer relacionamentos entre classes. É descrita na UML como “**relacionamentos estruturais entre objetos de tipos diferentes**”.

Associações podem representar uma **conexão física** ou **conceitual** entre as instâncias de classes.

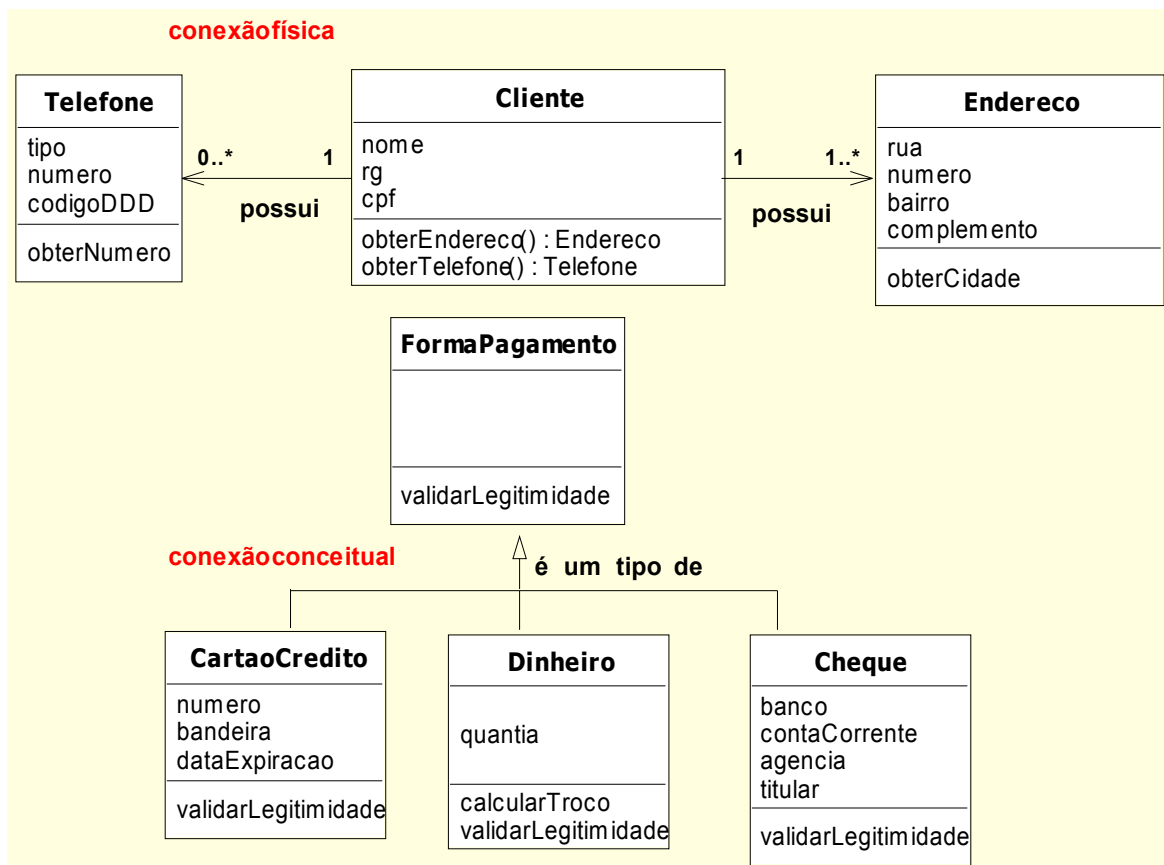


Figura 8 – Associações com conexão física e conexão conceitual

Notação UML:

- associação é representada por uma linha que liga as classes envolvidas no relacionamento
- pode conter uma seta de direção de leitura (navegabilidade)
- nas extremidades dessa linha é representada a multiplicidade da associação (0, 1, *)

Associações são freqüentemente encontradas na forma de **verbos** em descrições de problemas.

Tipos de navegabilidade:

- ✓ navegabilidade bilateral (mais comum) - cada entidade da associação referencia uma à outra
- ✓ navegabilidade unilateral - apenas uma das partes da associação referencia a outra

O nome de uma associação binária ou bilateral é lido em geral em um determinado sentido, mas a associação bilateral existe em ambos os sentidos.

Exemplo:

- ✓ Funcionário *cadastra* Ordem de serviço
- ✓ Ordem de serviço é *cadastrada* pelo Funcionário


2. Associações Avançadas

2.1 Agregação e Composição

Agregação é uma forma especializada de associação na qual **um objeto é parte ou compõe outra entidade**.

A notação UML que caracteriza o relacionamento de agregação é: 

Composição é uma associação mais forte na qual **o objeto que é parte ou compõe outra entidade (mandatória) tem sua existência vinculada a esta**. Caso a entidade mandatória seja excluída, as demais partes que a compõe também devem ser excluídas.

A notação UML que caracteriza o relacionamento de composição é: 

Palavras comuns para expressar relacionamentos de agregação e composição são: “parte de” e “contém”.

Na figura 3, a associação entre a classe *OrdemServico* e a classe *ItemOrdemServico* é uma composição. Pois ao se cancelar uma ordem de serviço não faz mas sentido manter o(s) item(ns) dessa ordem de serviço.

Como diferenciar os relacionamentos de agregação e associação?

Os testes seguintes podem ajudar a determinar se uma associação deve ser modelada como uma agregação:

- ✓ O termo “parte de” se aplica para descrever o relacionamento das entidades?
- ✓ Existe alguma operação na entidade mandatória que se aplica automaticamente para as entidades que a compõe? Por exemplo, a operação de exclusão da entidade mandatória deve ser precedida pela execução da mesma operação de exclusão das entidades relacionadas.

Agregações incluem explosões de uma entidade em entidades constituintes daquela.

Exemplo:

Uma Empresa é uma agregação de suas Divisões ou Órgãos de trabalho que por sua vez são constituídos de Departamentos.

Agregação versus Generalização

São formas de relacionamento distintos.

Na agregação dois objetos distintos estão envolvidos no relacionamento; um desses objetos é parte do outro.

Generalização relaciona objetos de mesma classe, isto é, relaciona subclasses à superclasse. Na generalização, um objeto é simultaneamente uma instância tanto da superclasse como de uma subclasse envolvida na generalização.

2.2 Modelando uma associação na forma de uma classe

Por vezes pode ser interessante modelar uma associação como uma classe. Essa associação, além de estabelecer uma ligação entre duas entidades apresenta também um ou mais atributos e operações inerentes ao conceito que ela representa.

Exemplo:

Originalmente, a classe *OrdemServico* possui uma associação direta com as classes *Produto* e *Servico* que são itens de uma determinada ordem de serviço. Mas, como essa associação apresenta a necessidade de ter o atributo quantidade, para cada item da ordem de serviço é necessário especificar sua quantidade, tal associação foi modelada como uma classe.

Sendo modelada como uma classe, uma responsabilidade lhe pode ser empregada - calcular o subtotal de cada item.

2.3 Associação recursiva

Caso o exemplo de modelagem do sistema de ordem de serviço seja expandido para incorporar a estrutura organizacional da empresa a qual esteja vinculado, é possível que seja necessário tratar de objetos do tipo *Departamento*.

Por exemplo, o departamento de marketing é subdividido nas áreas de marketing para produtos e para serviços.

Esse cenário implica uma associação recursiva de *Departamento* consigo mesmo para representar o conceito de subdepartamentos.

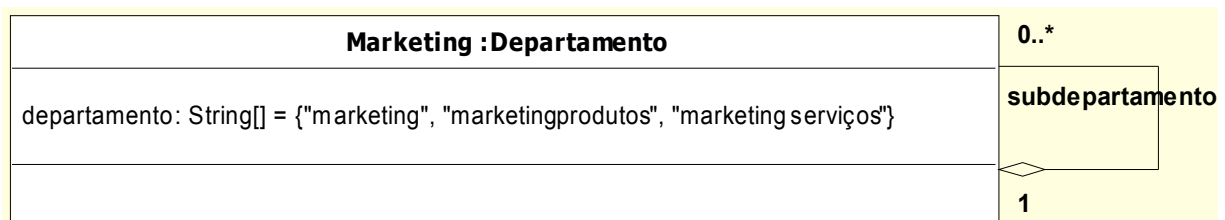


Figura 9 – Associação recursiva entre departamentos e subdepartamentos

O diagrama acima é o diagrama de objetos da UML que mostra instâncias de classes e é útil para explicar pequenas partes do diagrama de classes com relacionamentos complicados.

2.4 Generalização, Especialização e Herança

Generalização, especialização e herança são abstrações poderosas para **compartilhar similaridades entre classes enquanto se preserva as especificidades das mesmas**. São formas de **relacionar classes por meio de hierarquias**.

Generalização é o relacionamento entre uma determinada classe e uma ou mais versões refinadas (especializadas) desta.

A classe sendo especializada é denominada *superclasse* (*classe ancestral ou classe base*) e as classes refinadas ou especializadas são denominadas *subclasses* (ou *classes descendentes*).

A palavra reservada em Java para indicar o mecanismo de herança é **extends**.

```
class CartaoCredito extends FormaPagamento { ... }
```

Exemplo: (vide figura 3)

A classe *FormaPagamento* pode ser considerada como uma generalização das classes *CartaoCredito*, *Dinheiro* e *Cheque*.

Na superclasse concentra-se as características (atributos e operações) relacionadas a qualquer forma de pagamento, enquanto nas subclasses são especializadas as características conforme o papel ou função que cada uma representa.

Dessa forma, a subclasse *CartaoCredito* acrescenta características próprias como: *numero*, *bandeira* e *dataExpiracao*. Enquanto que a subclasse *Cheque* acrescenta: *banco*, *contaCorrente*, *agencia*, *titular*. A subclasse *Dinheiro*, por sua vez, acrescenta as características: *quantia* e *calcularTroco*.

Todas as subclasses, como são a própria entidade *FormaPagamento*, podem acessar as características definidas no escopo desta que no caso se limita ao comportamento *validarLegitimidade*.

UM OBJETO PODE SER TRATADO COMO UM OBJETO DO SEU TIPO OU DO SEU TIPO BASE

```
FormaPagamento cartao = new CartaoCredito();
```

```
FormaPagamento cheque = new Cheque();
```

```
FormaPagamento dinheiro = new Dinheiro();
```

As instâncias de objeto *cartao*, *cheque* e *dinheiro* estão sendo tratadas como instâncias da classe *FormaPagamento* em função de terem sido declaradas como sendo desse tipo.

Mas cada instância de objeto, é na verdade, do tipo mais especializado na hierarquia de herança da classe *FormaPagamento*, conforme a classe indicada após a palavra reservada *new*.

Dessa forma, neste momento elas podem ser tratadas unicamente na sua forma base o que implica acesso restrito apenas ao comportamento *validarLegitimidade*.

Para obter acesso as características específicas, é necessário fazer um *cast* (conversão) dessas instâncias para o tipo de suas subclasses. Por exemplo:

```
CartaoCredito cartaoCredito = (CartaoCredito) cartao;
```

A instância de objeto *cartaoCredito* é tão somente uma referência para a instância *cartao*.

Porém, é visível na forma da subclasse *CartaoCredito*. O que garante acesso tanto as suas próprias características como as da classe base.

Os termos herança, generalização e especialização representam o mesmo conceito e são usados como sinônimos.

Generalização e especialização são dois enfoques diferentes de um mesmo relacionamento, visto respectivamente hora a partir da superclasse e hora a partir da subclasse.

Herança, por sua vez, refere-se ao mecanismo de compartilhamento de atributos e operações por meio de hierarquias.

Não há limites para o número de classes envolvidas numa hierarquia de herança.

Palavras comuns para expressar o relacionamento de herança são: “é um” e “é um tipo de”.

Na modelagem a generalização facilita identificar o que é similar e o que é diferente entre classes de objetos. Já na implementação a herança de operações é o veículo pelo qual se faz o reuso de código.

Como a herança não se trata de um relacionamento entre objetos de classes diferentes, não se aplica nomear o relacionamento e especificar a multiplicidade deste.

Operação de cast - conversão entre tipos de mesma hierarquia de herança

```
FormaPagamento formaPagto = new CartaoCredito();
```

```
// cast denominado downcasting ou especialização (da superclasse para subclasse)
```

```
CartaoCredito cartaoCredito = (CartaoCredito) formaPagto;
```

```
// cast denominado upcasting ou promoção (da subclasse para superclasse)
```

```
formaPagto = (FormaPagamento) cartaoCredito;
```

Exercício:

2) O exemplo de modelagem do sistema de ordem de serviço deve ser expandido para incorporar qual funcionário gerou ou cadastrou a ordem de serviço. Essa expansão deve prever que um funcionário está vinculado a um determinado departamento.

Crie uma entidade Pessoa e a especialize para atender essa nova demanda e corrija o que for necessário em relação à modelagem existente.

V. Polimorfismo

Polimorfismo é a terceira característica essencial em uma linguagem orientada a objetos, após abstração de dados e herança.

É um recurso que provê uma forma de separar “o quê” – a definição de um comportamento, do “como” – a implementação deste.

Através do polimorfismo as subclasses de uma mesma classe base podem expressar suas diferenças no comportamento de métodos que podem ser acionados através da classe base.

Polimorfismo é também chamado de *dynamic binding*, *late binding* ou *runtime binding*.

Considere o seguinte exemplo:

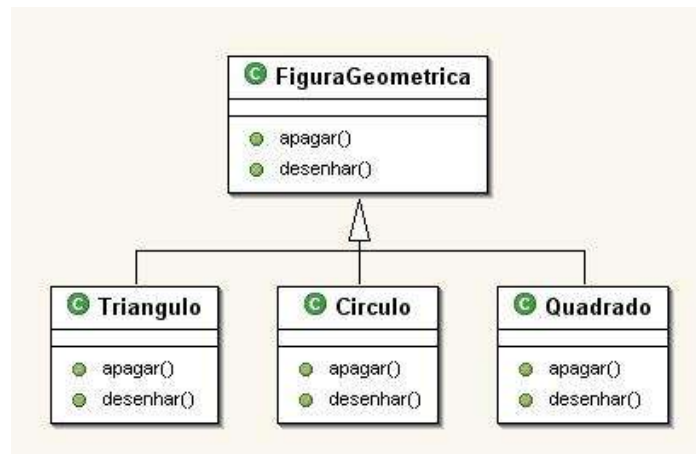


Figura 10 – Hierarquia de herança para figuras geométricas

```
FiguraGeometrica figura = new Quadrado();
```

Um objeto quadrado é criado e a referência gerada é imediatamente promovida para a classe base *FiguraGeometrica*. Apesar de parecer um pouco estranho, esse trecho de código está correto uma vez que *Quadrado* é um tipo de *FiguraGeometrica* em função da herança estabelecida.

Analise o código abaixo apenas da perspectiva de herança e polimorfismo.

```
1. package figurageometrica;
2. //classe base da hierarquia de herança, define dois comportamentos
3. public class FiguraGeometrica {
4.     public void desenhar() {}
5.     public void imprimirPropriedades() {}
6. }
7. //subclasse Circulo
8. public class Circulo extends FiguraGeometrica {
9.     int raio;
10.    public Circulo() { super(); } //construtor padrão
11.    public Circulo(int raio) { //construtor sobrecarregado
12.        super(); //aciona explicitamente o construtor da superclasse
13.        this.raio = raio;
14.    }
15.    public void desenhar() { System.out.println("Circulo.desenhar()"); }
16.    public void imprimirPropriedades() {
17.        System.out.println("Circulo com raio = " + this.raio);
18.    }
19. }
20. //subclasse Triangulo
21. public class Triangulo extends FiguraGeometrica {
22.     String classificacao;
23.     public Triangulo() { super(); }
24.     public Triangulo(int tipo) {
25.         super();
26.         //estrutura de decisão switch, dependendo do valor do param. tipo o atributo
27.         //classificacao recebe um valor
28.         switch (tipo) {
29.             default:
30.                 case 0: this.classificacao = " Eqüilátero ";
31.                 case 1: this.classificacao = " Isósceles ";
32.                 case 2: this.classificacao = " Escaleno ";
33.         }
34.     }
```

```

35.     public void desenhar() { System.out.println("Triangulo.desenhar()"); }
36.     public void imprimirPropriedades() {
37.         System.out.println("Triangulo do tipo " + this.tipo);
38.     }
39. }
40. //subclasse Quadrado
41. public class Quadrado extends FiguraGeometrica {
42.     int area;
43.     public Quadrado() { super(); }
44.     public Quadrado(int area) {
45.         super();
46.         this.area = area;
47.     }
48.     public void desenhar() { System.out.println("Quadrado.desenhar()"); }
49.     public void imprimirPropriedades() {
50.         System.out.println("Quadrado com area = " + this.area);
51.     }
52. }
53. //classe que contém método main para executar operações sobre instâncias de
54. //FiguraGeometrica
55. public class TestaFiguraGeometrica {
56.     public static void informarPropriedades(FiguraGeometrica figura) {
57.         figura.imprimirPropriedades();
58.     }
59.
60.     public static void main(String[] args) {
61.         ...
62.         //Trecho que cria instâncias das subclasses de FiguraGeometrica omitido,
63.         //considere que o array figura[i] contém em cada posição (i) uma instância de
64.         //cada subclasse de FiguraGeometrica
65.         ...
66.         //Faz chamadas ao metodo desenhar() usando polimorfismo
67.         for(int i = 0; i < figura.length; i++) {
68.             figura[i].desenhar();
69.             informarPropriedades(figura[i]);

```

```
70.         }
71.     }
72. }
```

Para a instância *figura*, se acionarmos o método *desenhar* qual será o resultado?

```
figura.desenhar(); // == ?
```

É esperado que o método *desenhar* de *FiguraGeometrica* seja acionado uma vez que a instância *figura* é do tipo da classe base.

Contudo, na verdade é acionado o método *desenhar* da subclasse *Quadrado* via polimorfismo.

```
figura.desenhar(); // == Quadrado.desenhar()
```

Considerando agora o trecho da classe *TestaFiguraGeometrica* na linha 69:

```
informarPropriedades(figura[i]);
```

Essa chamada ao método executa o corpo deste definido nas linhas 56 a 58.

Note que na assinatura deste método o parâmetro definido é do tipo *FiguraGeometrica*.

Contudo, o que está sendo passado como argumento efetivamente são subclasses de *FiguraGeometrica* contidas dentro do array *figura[i]*.

O resultado é a invocação dos métodos *imprimirPropriedades* (linha 57) das subclasses de *FiguraGeometrica*.

Isso decorre da linguagem Java implementar polimorfismo.

Em tempo de compilação, a linha 57 está correta porque a classe base *FiguraGeometrica* contém um método com mesmo nome conforme a linha 5.

Porém, em tempo de execução (*runtime*) o que é considerado é a classe da instância que efetivamente foi passada como parâmetro para o método *informarPropriedades*.

Assim, se o array *figura* conter na sequência instâncias de objetos do tipo *Circulo*, *Triangulo* e *Quadrado* será gerada a seguinte saída:

```
“Circulo com raio = “
```

```
“Triangulo do tipo “
```

“Quadrado com area = “

Observe como herança e polimorfismo trabalharam juntas para garantir a extensibilidade da classe *TestaFiguraGeometrica*. Mesmo que sejam incluídas novas subclasses de *FiguraGeometrica*, por exemplo cubo, cilindro e pirâmide, o método *informarPropriedades (FiguraGeometrica figura)* ainda continuará funcionando sem nenhuma alteração.

Este exemplo didático coloca em evidência as vantagens de extensibilidade e reuso obtidas por meio de orientação a objetos.

Portanto, polimorfismo é um rico recurso da programação orientada a objetos pois permite:

- ✓ organização avançada do código – a classe *FormaPagamento* estabelece um protocolo de comportamento, todas as suas subclasses devem implementar o comportamento de validação de legitimidade conforme suas necessidades
- ✓ melhor leitura do código através da separação de definições de suas implementações
- ✓ criação de programas **extensíveis** que podem expandir mesmo após seu projeto e criação inicial, quando se faz necessário acrescentar novas características

VI. Classe Abstrata

Classes abstratas são freqüentemente utilizadas para a **definição** de métodos que irão ser herdados pelas subclasses e estas são responsáveis por prover a implementação deste.

Classes abstratas organizam características comuns para várias outras classes, no caso suas descendentes.

Uma classe abstrata é uma classe que não pode ser instanciada, isto é, não há instâncias (objetos) dela. Pode haver objetos para suas subclasses, desde que estas não sejam abstratas (também denominadas de classes concretas).

Na figura 3, a entidade *FormaPagamento* se encontra em itálico porque é uma classe abstrata.

Além de ser a classe mãe da hierarquia de herança, é uma classe abstrata pois a operação *validarLegitimidade* é definida como abstrata e deve ser implementada pelas subclasses concretas.

O protocolo de comportamento entre classe base e subclasses é estabelecido: validar a legitimidade de uma forma de pagamento indicando se é válida (true) ou não (false).

Neste sentido, cada subclasse pode sobrescrever ou redefinir o comportamento *validarLegitimidade* definido na superclasse.

Ou seja, *validarLegitimidade* para *CartaoCredito* significa validar se há crédito para a realização da compra de produtos ou solicitação de serviço. Enquanto para *Cheque* essa mesma operação representa consultar junto a entidades credenciadas se o portador não possui nome sujo na praça com cheques não liquidados. Já para *Dinheiro* essa mesma operação pode representar algo mais sofisticado como acionar um mecanismo eletrônico de certificação da autenticidade da(s) cédula(s) usadas para pagar.

Algumas classes abstratas aparecem naturalmente no processo de modelagem de um sistema. Já outras classes precisam ser adicionadas artificialmente para servirem de mecanismo de promoção de reutilização do código.

VII. Interface (Herança Múltipla em JAVA)

A palavra reservada **interface** produz uma classe abstrata completa, no sentido de que a interface não possui implementações mas apenas definições de métodos.

Uma interface pode conter atributos mas os mesmos são tratados implicitamente como estáticos e como constantes.

Seu uso é voltado para criação de protocolos de comportamento, pois ela provê apenas uma forma e não uma implementação.

Classes que implementam uma interface (palavra reservada **implements**) são obrigadas a prover corpo para os métodos definidos naquela.

Nesse sentido uma interface define “o que” e a classe que a implementa estabelece “o como”.

Através das interfaces a linguagem Java consegue implementar um mecanismo de herança múltipla onde uma classe pode ser promovida (*upcast*) para mais de uma classe base.

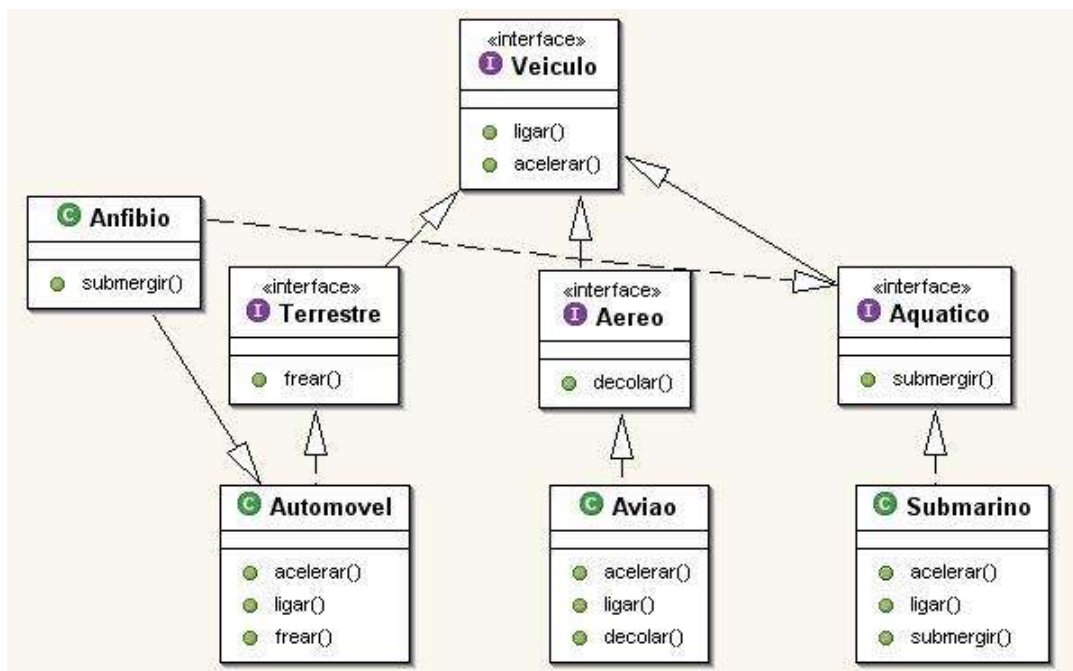


Figura 11 – Esquema de herança múltipla para classe Anfíbio

A classe *Anfibio* é um *Automovel* (veículo terrestre) quanto um veículo aquático.

Exercício:

3) Analise o texto abaixo e identifique as principais entidades, suas características e comportamentos. Identifique também os relacionamentos existentes entre as entidades e crie uma modelagem.

A Universidade de Tecnologia de Campinas (UniTechCamp) é uma instituição educacional experiente na formação de profissionais da área de TI e provê uma forma prática e garantida de seus clientes conquistarem know-how em novas tecnologias.

Abaixo constam os requisitos do sistema de informação que gerencia a composição das turmas de treinamento e a matrícula dos alunos.

Os instrutores são desenvolvedores habituados a desbravar tecnologias de maneira que sabem como montar o catálogo de treinamentos organizado em módulos estratégicos para a sedimentação eficaz do conteúdo.

Assim, os treinamentos apresentam a seguinte estrutura:

Os dados cadastrais obtidos dos clientes são: R.G., CPF, nome, sexo, data de nascimento, nacionalidade e estado civil. As informações endereço, telefone e e-mail são informações multivaloradas, o que possibilita a captação de mais de uma destas informações.

Além dessas informações, é cadastrado a empresa em que o cliente trabalha atualmente, sua área de atuação, a formação acadêmica dele e se ele deseja receber o e-news da UniTechCamp.

Quando um cliente se inscreve numa turma é gerado um contrato entre ele e a escola onde consta informações como: data contratual, forma de pagamento, valor do treinamento, desconto, código da turma em que se inscreveu e qual treinamento essa turma pertence. Também é gerado uma matrícula (código unívoco) que identifica o aluno a cada novo treinamento que vier a participar.

Existe também um cadastro dos instrutores da nossa escola. As informações coletadas são: nome, R.G., CPF, breve descrição da experiência profissional deste e sua(s) certificação(ões) em tecnologias. Quanto a essa última informação é registrado apenas o nome da certificação, a data e o local em que a obteve.

Um instrutor ministra cursos (treinamentos) para determinadas turmas fechadas. Cada turma está associada a um oferecimento o qual contém as seguintes informações: data início, data conclusão, horário início e de fim das aulas e horário do coffee break.

As informações dos treinamentos são: nome, ementa, carga horária, público alvo e data de criação desse treinamento.

O contrato dos clientes com a UniTechCamp apresenta as seguintes informações: código único, data contratual, forma de pagamento, treinamento comprado, dados da contratada, dados do contratante e número de matrícula deste. Este número de matrícula é um código único conferido ao aluno no primeiro treinamento realizado; nos próximos treinamentos a mesma matrícula deve ser associada ao contrato.

VIII. Padrões de projetos

1. O que é um padrão de projeto?

Christopher Alexander afirma: “cada padrão de projeto descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você pode usar esta solução mais de um milhão de vezes sem nunca fazê-lo da mesma maneira”.

Muito embora Alexander estivesse falando de padrões de projetos no contexto da construção civil, o que ele diz se aplica para padrões de projetos orientados a objetos.

As soluções para desenvolver software são expressas em termos de objetos, interfaces e classes (abstratas ou concretas) em vez de paredes e estruturas metálicas, mas no cerne de ambos os tipos de padrões está a solução para um problema em um determinado contexto.

Elementos essenciais de um padrão de projeto:

1. **nome:** é uma referência que podemos usar para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras. Nomear padrões estabelece um vocabulário único entre os desenvolvedores de software.
2. **definição do problema:** é descrito quando aplicar o padrão e em quais cenários. É explicado o problema e seu contexto. Por vezes, o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido adotar o padrão.
3. **descrição da solução:** descreve os elementos que compõem o padrão de projeto, seus relacionamentos e suas responsabilidades e colaborações. Não é descrito um padrão concreto ou solução específica, pois o objetivo é definir um gabarito passível de ser aplicado em muitas situações diferentes.
4. **conseqüências:** são os resultados e análises das vantagens e desvantagens (*trade-offs*) da aplicação do padrão. Essa análise é crítica na tomada de decisão quanto à adoção de padrões alternativos e para a compreensão dos custos e benefícios da aplicação do padrão.

2. O que um padrão de projeto não é?

Padrões de projetos não são projetos tais como listas encadeadas e tabelas de acesso aleatório que podem ser codificadas em classes e ser reutilizadas tais como estão. Nem projetos complexos, de domínio específico, para uma aplicação inteira ou subsistema.

Padrões de projetos são descrições de objetos e classes comunicantes que são customizadas para resolver um problema geral de projeto num contexto particular.

3. Catálogo de padrões de projetos

Como existem muitos padrões de projetos que variam no seu campo de aplicação devemos organizá-los. É feita uma classificação de padrões de maneira que é possível se referir a famílias de padrões de projetos.

O critério para a classificação de padrões de projetos é a sua finalidade. Esta finalidade reflete o que o padrão faz e é subdividida em: criação, estrutural ou comportamental.

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

Tabela com 23 padrões de projetos GoF

GoF – “Gang of Four” que editou o 2º livro citado na seção IX (Referências Bibliográficas).

Existem diversos catálogos de padrões de projetos, como padrões de projetos J2EE, padrões relacionados à Arquitetura de Sistemas e aqueles relacionados a Segurança de Sistemas dentre outros.

Exemplo de padrão de projeto – Model View Controller

Problema:

Aplicações podem ser construídas de maneira modularizada facilitando assim a incorporação de adequações ao longo do tempo.

Aquelas aplicações que em um mesmo trecho de código tratam da: (a.) validação da entrada de dados do usuário; (b.) tomada de decisões em cima das ações do usuário para decidir para qual funcionalidade da aplicação direcionar o fluxo de execução; e (c.) tratamento da interface do usuário são candidatas a manutenções recorrentes e custosas.

Considerando também que a interface com o usuário sofre alterações com frequência, não é desejável adequar toda aplicação toda vez que uma alteração desse tipo for demandada.

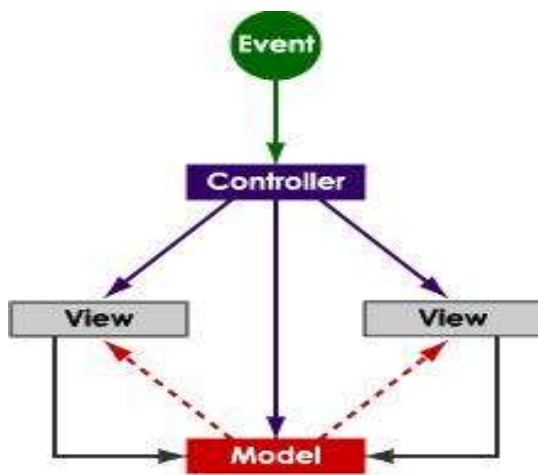
Solução:

O padrão MVC (Model – dados, View – interface com usuário, Controller – controlador do fluxo de execução e navegação) divide em partes distintas cada aspecto da aplicação mencionado acima conforme as iniciais do padrão.

Conseqüências:

Na implementação desse padrão são utilizadas diversas estratégias de implementação adaptadas a cada tipo de interface com o usuário – Web, Rich Client etc. Também utiliza outros padrões de projeto como Front Controller, Command, Composite View etc.

Dessa forma, existe uma curva de aprendizado para seu uso, mas depois de incorporado agrega produtividade e flexibilidade na construção de aplicações em partes distintas com baixo acoplamento.



Eventos são capturados pelo Controlador

Controlador muda o estado do modelo ou a visão

View obtém dados do modelo

Modelo atualiza a visão quando os dados mudam

Figura 12 – Esquema básico de funcionamento do padrão de projeto MVC

IX. Referências Bibliográficas

Rumbaugh, James - Object-oriented modeling and design, Prentice Hall, 1991.

Gamma, Erich 1; Helm Richard 2; Johnson, Ralph 3; Vlissides, John 4 - Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Eckel, Bruce - Thinking in java. 3a. Edição, 2000.

Terry Quatrani - Visual Modeling with Rational Rose and UML, Addison-Wesley, 1998.

Coordenação do Programa



Patrocinadores



Parceiros

A F P U

