Universidade Federal de Jataí
Curso de Ciência da Computação
Disciplina de Programação Orientada a Objetos

Notas de Aula
Conceitos e Definições

Programação Orientada a Objetos: Uma Abordagem com Java

Prof. Marcos Wagner de Souza Ribeiro

2021

1 Introdução

1.1 Programação Orientada a Objetos

Nesta disciplina serão apresentados os conceitos de programação orientada a objetos. Para tanto será necessário o uso de uma linguagem de programação que siga esses conceitos. A linguagem escolhida foi JAVA. A linguagem Java apresenta uma sintaxe muito semelhante às linguagens C e C++. Porém inicialmente é preciso falar um pouco sobre Orientação a Objetos e Paradigmas de Programação.

a) Paradigmas de Programação

Fonte: https://www.treinaweb.com.br/blog/linguagens-e-paradigmas-de-programacao

No passado escrevia-se programas utilizando apenas linguagens de *baixo nível*. A escrita é engessada, complexa e muito específica, sendo pouco acessível para os desenvolvedores no geral. Esse tipo de linguagem exige muito conhecimento de quem a programa (inclusive relacionado à forma com que o processador opera uma instrução-máquina).

Recentemente foi liberado o código-fonte utilizado no computador que guiou a missão *Apollo* que teve como principal objetivo levar o homem à lua (na tão famigerada corrida espacial entre a União Soviética e os EUA), o *Apollo Guidance Computer*: *https://github.com/chrislgarry/Apollo-11*
Se você navegar no repositório acima encontrará diversos códigos-fonte com instruções como essas:

```
PROGLARM        CS      DSPTAB +11D
                MASK    OCT40400
                ADS     DSPTAB +11D

MULTEXIT        XCH     ITEMP1          # OBTAIN RETURN ADDRESS IN A
                RELINT
                INDEX   A
                TC      1

MULTFAIL        CA      L
                AD      BIT15
                TS      FAILREG +2

                TCF     MULTEXIT
```

São instruções da linguagem *AGC Assembly Language*, uma variante da *Assembly*, que por sinal, é de *baixo nível*.

Um programa escrito em uma dessas linguagens, chamadas de *baixo nível*, é composto por uma série de *instruções de máquina* que determinam quais operações o processador deve

executar. Essas instruções são convertidas para a linguagem que o processador entende, que é a linguagem binária (sequência de bits `0` e `1`), que é categorizada como *First-generation programming language (1GL)*, em livre tradução: *linguagem de programação de primeira geração*.

a.1) Linguagens de alto nível

Com a popularidade dos computadores criou-se um "problema": alta demanda por software e, consequentemente, por programadores. Talvez você esteja pensando que isso não é exatamente um problema, e sim uma coisa boa, uma tendência, um novo mercado. Faz sentido, até certo ponto. O problema era encontrar mão de obra qualificada para codificar àquelas instruções tão complicadas.

Com isso, novas linguagens surgiram e, cada vez mais, aproximavam-se da linguagem humana. Isso abriu "fronteiras" para que uma enorme gama de novos desenvolvedores se especializassem. Tais linguagens são denominadas como sendo de *alto nível*. As linguagens modernas que hoje conhecemos e usamos são de *alto nível*:
- C, PHP, Java, Rust, C#, Python, Ruby etc.

> *"Quanto mais próxima da linguagem da máquina, mais baixo nível é a linguagem. Quanto mais próxima da linguagem humana, mais alto nível ela é."*

a.2) Paradigmas das linguagens de programação

Quando uma linguagem de programação é criada, a partir das suas características, ela é categorizada em um ou mais paradigmas.

A definição do dicionário Aurélio para "paradigma":

1. *Algo que serve de exemplo geral ou de modelo.*
2. Conjunto das formas que servem de modelo de derivação ou de flexão.
3. Conjunto dos termos ou elementos que podem ocorrer na mesma posição ou contexto de uma estrutura.

O *paradigma* de uma linguagem de programação é a sua identidade. Corresponde a um conjunto de características que, juntas, definem como ela opera e resolve os problemas. Algumas linguagens, inclusive, possuem mais de um paradigma, são as chamadas *multi paradigmas*.
Alguns dos principais paradigmas utilizados hoje no mercado:

- Funcional
- Lógico
- Declarativo
- Imperativo
- Orientado a objetos
- Orientado a eventos

Paradigma funcional

O foco desse paradigma está na avaliação de funções. Como na matemática quando temos, por exemplo, uma função f(x): f(x) = x + 2
x é um parâmetro (o valor de entrada) e, após a expressão ser avaliada, obtêm-se o resultado.
Se o valor de entrada for 2, o resultado da avaliação da nossa função será 4.
Algumas das linguagens que atendem a esse paradigma: *F#* (da Microsoft), *Lisp*, *Heskell*, *Erlang*, *Elixir*, *Mathematica*.

É possível desenvolver de forma "funcional" mesmo em linguagens não estritamente funcionais. Por exemplo, no *PHP*, que é uma linguagem multi paradigma, teríamos:

```
<?php
$sum = function($value) {
    return $value + 2;
};
echo $sum(2); // 4
```

Paradigma lógico

Também é conhecido como "restritivo". Muito utilizado em aplicações de inteligência artificial. Esse paradigma chega no resultado esperado a partir de avaliações lógico-matemáticas. Se você já estudou lógica de predicados, confortável se sentirá em entender como uma linguagem nesse paradigma opera.

Principais elementos desse paradigma:

- **Proposições**: base de fatos concretos e conhecidos.
- **Regras de inferência**: definem como deduzir proposições.
- **Busca**: estratégias para controle das inferências.

Exemplo:

- **Proposição:** Chico é um gato.
- **Regra de inferência**: Todo gato é um felino.
- **Busca**: Chico é um felino?

A resposta para a **Busca** acima precisa ser **verdadeira**. A conclusão lógica é:
Se Chico é um gato e todo gato é felino, então Chico é um felino.

A idéia básica da programação em lógica é oferecer uma maneira que permita chegar a uma conclusão a partir de uma premissa.

A linguagem mais conhecida que utiliza esse paradigma é a **Prolog**. Esse paradigma é pouco utilizado em aplicações comerciais, seu uso se dá mais na área acadêmica.

Paradigma declarativo

O paradigma *declarativo* é baseado no *lógico* e *funcional*. Linguagens *declarativas* descrevem **o que** fazem e não exatamente **como** suas instruções funcionam.

Linguagens de marcação são o melhor exemplo: *HTML*, *XML*, XSLT, *XAML* etc. Não obstante, o próprio *Prolog* - reconhecido primariamente pelo paradigma *lógico* - também é uma linguagem declarativa. Abaixo alguns exemplos dessas linguagens.

HTML:

```
<article>
  <header>
    <h1>Linguagens e paradigmas de programação</h1>
  </header>
</article>
```

SQL:

```
SELECT nome FROM usuario WHERE id = 10
```

Paradigma imperativo

Você já ouviu falar em "*programação procedural*" ou em "*programação modular*"? De modo geral, são imperativas.

Linguagens clássicas como *C*, *C++*, *PHP*, *Perl*, *C#*, *Ruby* etc, "suportam" esse paradigma. Ele é focado na mudança de estados de variáveis (ao contrário dos anteriores).

Exemplo:

```
if(option == 'A') {
        print("Opção 'A' selecionada.");
}
```

A impressão só será realizada se o valor da variável `option` for igual a `A`.

Paradigma orientado a objetos

Esse é, entre todos, talvez o mais difundido. Nesse paradigma, ao invés de construirmos nossos sistemas com um conjunto estrito de procedimentos, assim como se faz em linguagens "fortemente imperativas" como o Cobol, Pascal etc, na orientação a objetos utilizamos uma lógica bem próxima do mundo real, lidando com objetos, estruturas que já conhecemos e sobre as quais possuímos uma grande compreensão.

OO é sigla para orientação a objetos

O paradigma orientado a objetos tem uma grande preocupação em esconder o **que não é** importante e em realçar **o que é** importante. Nele, implementa-se um conjunto de classes que

definem objetos. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento entre eles.

Esse é o paradigma mais utilizado em aplicações comerciais e as principais linguagens o implementam: *C#*, *Java*, *PHP*, *Ruby*, *C++*, *Python* etc.

Programação orientada a objetos e programação estruturada

Como a maioria das atividades que fazemos no dia a dia, programar também possui modos diferentes de se fazer. Quando começamos a utilizar linguagens como Java, C++, C#, Python e outras que possibilitam o paradigma orientado a objetos, é comum errarmos e aplicarmos a programação estruturada achando que estamos usando recursos da orientação a objetos.

Na programação estruturada, um programa é composto por três tipos básicos de estruturas:

- sequências: são os comandos a serem executados
- condições: sequências que só devem ser executadas se uma condição for satisfeita (exemplos: if-else, switch e comandos parecidos)
- repetições: sequências que devem ser executadas repetidamente até uma condição for satisfeita (for,while, do-while etc)

Essas estruturas são usadas para processar a entrada do programa, alterando os dados até que a saída esperada seja gerada. Até aí, nada que a programação orientada a objetos não faça, também, certo?

A diferença principal é que na programação estruturada, um programa é tipicamente escrito em uma única rotina (ou função) podendo, é claro, ser quebrado em subrotinas. Mas o fluxo do programa continua o mesmo, como se pudéssemos copiar e colar o código das subrotinas diretamente nas rotinas que as chamam, de tal forma que, no final, só haja uma grande rotina que execute todo o programa.

Além disso, o acesso às variáveis não possuem muitas restrições na programação estruturada. Em linguagens fortemente baseadas nesse paradigma, restringir o acesso à uma variável se limita a dizer se ela é visível ou não dentro de uma função (ou módulo, como no uso da palavra-chave static, na linguagem C), mas não se consegue dizer de forma nativa que uma variável pode ser acessada por apenas algumas rotinas do programa. O contorno para situações como essas envolve práticas de programação danosas ao desenvolvimento do sistema, como o uso excessivo de variáveis globais. Vale lembrar que variáveis globais são usadas tipicamente para manter estados no programa, marcando em qual parte dele a execução se encontra.

A programação orientada a objetos surgiu como uma alternativa a essas características da programação estruturada. O intuito da sua criação também foi o de aproximar o manuseio das estruturas de um programa ao manuseio das coisas do mundo real, daí o nome "objeto" como

uma algo genérico, que pode representar qualquer coisa tangível.

Esse novo paradigma se baseia principalmente em dois conceitos chave: classes e objetos. Todos os outros conceitos, igualmente importantes, são construídos em cima desses dois.

O que são classes e objetos?

Imagine que você comprou um carro recentemente e decide modelar esse carro usando programação orientada a objetos. O seu carro tem as características que você estava procurando: um motor 2.0 híbrido, azul escuro, quatro portas, câmbio automático etc. Ele também possui comportamentos que, provavelmente, foram o motivo de sua compra, como acelerar, desacelerar, acender os faróis, buzinar e tocar música. Podemos dizer que o carro novo é um objeto, onde suas características são seus atributos (dados atrelados ao objeto) e seus comportamentos são ações ou métodos.

Seu carro é um objeto seu mas na loja onde você o comprou existiam vários outros, muito similares, com quatro rodas, volante, câmbio, retrovisores, faróis, dentre outras partes. Observe que, apesar do seu carro ser único (por exemplo, possui um registro único no Departamento de Trânsito), podem existir outros com exatamente os mesmos atributos, ou parecidos, ou mesmo totalmente diferentes, mas que ainda são considerados carros. Podemos dizer então que seu objeto pode ser classificado (isto é, seu objeto pertence à uma classe) como um carro, e que seu carro nada mais é que uma instância dessa classe chamada "carro".

Assim, abstraindo um pouco a analogia, uma classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe. Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de instanciação da classe, como se estivéssemos usando esse molde (classe) para criar um objeto.

1.2 A plataforma Java

Fonte: Programação orientada a objetos : Curso técnico de informática / Victorio Albani de Carvalho, Giovany Frossard Teixeira. – Colatina: IFES, 2012.

Plataformas podem ser descritas como a combinação do sistema operacional e o hardware em que rodam. Nesse contexto, a maioria das plataformas de desenvolvimento existentes possui uma restrição marcante: cada programa é produzido para uma plataforma (Sistema Operacional + hardware) específica. A plataforma Java difere dessas plataformas pelo fato de desagregar o hardware de si, ou seja, trata-se de uma plataforma de software que roda em cima de outras plataformas baseadas em hardware.

Essa independência de hardware obtida pela plataforma Java deve-se à utilização do conceito de máquina virtual: a Java Virtual Machine (JVM). A JVM é um software que funciona sobre o sistema operacional, sendo responsável pelo processo de tradução de um programa Java

para uma plataforma específica. Assim, um programa feito em Java pode rodar em qualquer SO de qualquer arquitetura, desde que exista uma JVM implementada para ele.

1.3 Ambientes de desenvolvimento Java

Um programa Java precisa passar por um processo de compilação para ser analisada a existência de erros de sintaxe. Esse processo de compilação traduz o código-fonte escrito pelo programador para uma linguagem intermediária chamada Java bytecodes. Esse processo de tradução dos códigos fontes para Java bytecodes é feito por um programa chamado compilador. Então, é necessário que outra ferramenta chamada interpretador se responsabilize por interpretar esses bytecodes para o sistema operacional. Essa ferramenta que interpreta bytecodes é a máquina virtual Java (JVM). O conjunto de ferramentas necessárias para desenvolver, compilar e rodar aplicativos Java é disponibilizado em um kit conhecido como Java Development Kit (JDK). Assim, para começar a programar em Java você deve realizar o download do JDK e instalá-lo. Ao realizar o download do JDK, escolha a versão correta para seu sistema operacional. Com o JDK instalado, você pode começar a programar em Java utilizando um simples editor de texto para editar seus programas, como, por exemplo, o bloco de notas. Assim, você teria de editar seus programas, salvá-los com extensão .Java, compilá-los e então executá-los. Para facilitar e agilizar esse processo, existem disponíveis vários Ambientes de Desenvolvimento – Integrated Development Environment (IDE), que dão suporte à linguagem Java. Um IDE é um programa de computador que reúne ferramentas de apoio ao desenvolvimento de software com o objetivo principal de agilizar o processo de codificação. Há vários IDEs para programação Java. Os dois mais amplamente utilizados são o NetBeans e o Eclipse. Nessa disciplina utilizaremos o NetBeans.

O NetBeans IDE é um ambiente de desenvolvimento integrado gratuito e de código aberto. Esse IDE é executado em muitas plataformas, como Windows, Linux, Solaris e MacOS, sendo fácil de instalar e usar. O NetBeans IDE oferece aos desenvolvedores todas as ferramentas necessárias para criar aplicativos profissionais de desktop, empresariais, web e móveis multiplataformas. Assim, todo o processo de edição, compilação e execução dos programas será feito dentro do NetBeans.

1.4 Um programa em Java

```
public class Teste {
      public static void main(String[] args) {
            System.out.println("Olá! Primeiro teste");
      }
}
```

A principal diferença que podemos notar já de início é que todo programa em Java inicia-se com a definição de uma classe. Uma classe é definida pela palavra reservada class, seguida pelo nome da classe (no exemplo “Teste”). Por convenção, todo nome de classe inicia-se com uma

letra maiúscula. Um programa Java é formado por uma ou mais classes.

Assim como em C todo programa Java tem sua execução iniciada pelo método main, alguma das classes do programa Java deverá conter um método main. Essa classe é chamada de classe principal. As palavras “public” e “static” que aparecem na definição do método main serão explicadas em aulas futuras. Já a palavra “void”, assim como em C, indica que o método não possui retorno. O argumento “String args[ ]” é um vetor de Strings formado por todos os argumentos passados ao programa na linha de comando quando o programa é invocado. O comando “System.out.println ( )” é utilizado para imprimir algo na tela. Ele é equivalente ao “printf()” da linguagem C.

1.5 Variáveis e constantes

Assim como em linguagem C, programas feitos em Java devem ter suas variáveis declaradas e inicializadas antes de serem utilizados. Em Java toda variável é identificada por um nome e tem um tipo definido no ato da declaração, que não pode ser alterado.

A sintaxe para declaração de variáveis em Java é idêntica à que utilizamos em C:
Sintaxe: <tipo_da_variável> <nome_da_variável>;

2 Orientação a Objetos

2.1 Conceitos Básicos

Hoje, a maioria das linguagens de programação são **orientadas a objetos** como **Java**, **C#**, **Python** e C++ e, apesar de terem algumas diferenças na implementação, todas seguem os mesmos princípios e conceitos. Muitos programadores, apesar de utilizarem linguagens orientadas a objetos, não sabem utilizar alguns dos principais conceitos desse **paradigma orientado a objetos** e, por isso, desenvolvem sistemas com alguns erros conceituais e acabam escrevendo mais código que o necessário, não conseguindo reutilizar o código como seria possível.

Os principais conceitos do paradigma orientado a objetos são **Classes e Objetos, Associação , Encapsulamento, Herança e Polimorfismo.**

a) Classe e Objeto

Uma classe é uma forma de definir um tipo de dado em uma **linguagem orientada a objeto**. Ela é formada por dados e comportamentos.

Para definir os dados são utilizados os atributos, e para definir o comportamento são utilizados métodos. Depois que uma classe é definida podem ser criados diferentes objetos que utilizam a classe. A **Listagem 1** mostra a definição da classe Empresa, que tem os atributos nome, endereço, CNPJ, data de fundação, faturamento, e também o método imprimir, que apenas mostra os dados da empresa.

```
public class Empresa {

	private String nome;
	private String cnpj;
	private String endereco;
	private Date dataFundacao;
	private float faturamento;

	public void imprimir() {
		System.out.println("Nome: " + nome);
		System.out.println("CNPJ: " + cnpj);
		System.out.println("Endereço: " + endereco);
		System.out.println("Data de Fundação: " + dataFundacao);

	}

	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public String getCnpj() {
		return cnpj;
```

```
	}
	public void setCnpj(String cnpj) {
		this.cnpj = cnpj;
	}
	public String getEndereco() {
		return endereco;
	}
	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}
	public Date getDataFundacao() {
		return dataFundacao;
	}
	public void setDataFundacao(Date dataFundacao) {
		this.dataFundacao = dataFundacao;
	}
	public float getFaturamento() {
		return faturamento;
	}
	public void setFaturamento(float faturamento) {
		this.faturamento = faturamento;
	}

}
```

Listagem 1. Classe Empresa

Com a classe definida, podem ser criados diversos objetos do tipo Empresa, por isso a **Listagem 2** mostra como criar esses objetos, bastando declarar uma variável com o tipo Empresa e com a palavra reservada new criar um novo objeto. Depois podemos definir os dados para os atributos da classe Empresa e, por fim, chamar o método definido.

```
public class Main {

	public static void main(String[] args) {


		Empresa empresa1 = new Empresa();
		empresa1.setNome("Loja 1");
		empresa1.setCnpj("12343232");
		empresa1.setDataFundacao(new Date());
		empresa1.setEndereco("Rua abc, 100");
		empresa1.setFaturamento(50000);
		empresa1.imprimir();


		Empresa empresa2 = new Empresa();
		empresa2.setNome("Loja 2");
		empresa2.setCnpj("12354432");
		empresa2.setDataFundacao(new Date());
		empresa2.setEndereco("Rua abc, 200");
		empresa2.setFaturamento(50000);
		empresa2.imprimir();


		Empresa empresa3 = new Empresa();
		empresa3.setNome("Posto de Gasolina");
		empresa3.setCnpj("12345434");
		empresa3.setEndereco("Rua afd, 500");
		empresa3.setFaturamento(10000);
		empresa3.setDataFundacao(new Date());
```

```
            empresa3.imprimir();

      }
}
```

Listagem 2. Definição dos objetos do tipo Empresa

Encapsulamento

O conceito do encapsulamento consiste em “esconder” os atributos da classe de quem for utilizá-la. Isso se deve por dois motivos principais. Um é para que alguém que for usar a classe não a use de forma errada como, por exemplo, em uma classe que tem um método de divisão entre dois atributos da classe - se o programador java não conhecer a implementação interna da classe, ele pode colocar o valor zero no atributo do dividendo, mas se a classe estiver corretamente encapsulada podemos impedir que o programador faça isso. Esse tipo de implementação é feito via os métodos `get` e `set`. A **Listagem 3** mostra o código da classe `Divisao` citada como exemplo.

```
public class Divisao {

      private int num1;
      private int num2;

      public void divisao() {
            System.out.println("A divisao e: " + (num1 / num2));
      }

      public int getNum1() {
            return num1;
      }

      public void setNum1(int num1) {
            this.num1 = num1;
      }

      public int getNum2() {
            return num2;
      }

      public void setNum2(int num2) {
            if (num2 == 0) {
                  num2 = 1;
            } else {
                  this.num2 = num2;
            }
      }

}
```

Listagem 3. Classe Divisao

O outro motivo é de manter todo o código de uma determinada classe encapsulada dentro dela mesmo como, por exemplo, se existe uma classe Conta, talvez seja melhor não permitir que um programador acesse o atributo saldo diretamente, nem mesmo com os métodos `get` e `set`, mas

somente por operações, como saque, depósito e saldo. A **Listagem 4** mostra a implementação da classe Conta, onde só é possível acessar a atributo saldo pelas operações disponibilizadas na classe e os outros atributos podem ser acessados via métodos `get` e `set`.

```
public class Conta {

	private String agencia;
	private String numero;
	private float saldo;

	public void saque(float valor) {
		if (saldo < valor) {
			System.out.println("O saldo é insuficiente!");
		} else {
			saldo -= valor;
		}
	}

	public String getAgencia() {
		return agencia;
	}

	public void setAgencia(String agencia) {
		this.agencia = agencia;
	}

	public String getNumero() {
		return numero;
	}

	public void setNumero(String numero) {
		this.numero = numero;
	}

	public void deposito(float valor) {
		saldo += valor;
	}

	public void saldo() {
		System.out.println("O saldo é: " + saldo);
	}

}
```

Listagem 4. Classe Conta

Associação de Classes

A associação de classes indica quando uma classe tem um tipo de relacionamento "tem um" com outra classe como, por exemplo, uma pessoa tem um carro e isso indica que a classe Pessoa tem uma associação com a classe Carro. Esse tipo de relacionamento entre classes é importante, porque define como as classes interagem entre elas nas aplicações.

A **Listagem 5** mostra como implementar uma associação um para um. A associação é feita entre as classes Pessoa e Carro. A classe Carro tem os atributos modelo, placa, ano e valor; e a classe Pessoa tem os atributos nome, endereço, telefone e dataNascimento. Além disso, ela tem um relacionamento com a classe Carro.

```
public class Carro {

	private String modelo;
	private String placa;
	private int ano;
	private float valor;

	public String getModelo() {
		return modelo;
	}

	public void setModelo(String modelo) {
		this.modelo = modelo;
	}

	public String getPlaca() {
		return placa;
	}

	public void setPlaca(String placa) {
		this.placa = placa;
	}

	public int getAno() {
		return ano;
	}

	public void setAno(int ano) {
		this.ano = ano;
	}

	public float getValor() {
		return valor;
	}

	public void setValor(float valor) {
		this.valor = valor;
	}

}



public class Pessoa {

	private String nome;
	private String endereco;
	private String telefone;
	private Date dataNascimento;

	// Relacionamento com a classe Carro
	private Carro carro;

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public String getEndereco() {
		return endereco;
	}
```

```
	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}

	public String getTelefone() {
		return telefone;
	}

	public void setTelefone(String telefone) {
		this.telefone = telefone;
	}

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public Carro getCarro() {
		return carro;
	}

	public void setCarro(Carro carro) {
		this.carro = carro;
	}

}
```

Listagem 5. Associação entre as classes Pessoa e Carro

Além do relacionamento um para um, também existem o relacionamento um para N e N para N. Esses relacionamentos são modelados com um vetor de objetos de uma classe para outro como, por exemplo, se uma pessoa pode ter mais de um carro, é necessário mapear esse relacionamento com uma lista de carros na classe Pessoa. A **Listagem 6** mostra a alteração necessária na classe Pessoa, mas na classe Carro não é necessária nenhuma alteração.

```
public class Pessoa {

	private String nome;
	private String endereco;
	private String telefone;
	private Date dataNascimento;

	// Relacionamento com a classe Carro
	private List<Carro> carros = new ArrayList<Carro>();

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public String getEndereco() {
		return endereco;
	}
```

```
	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}

	public String getTelefone() {
		return telefone;
	}

	public void setTelefone(String telefone) {
		this.telefone = telefone;
	}

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public List<Carro> getCarros() {
		return carros;
	}

	public void setCarros(List<Carro> carros) {
		this.carros = carros;
	}

}
```

Listagem 6. Classe Pessoa com relacionamento um para `N`

Outros Relacionamentos

Objetos do mundo real relacionam-se uns com os outros de diversas formas: Um objeto motor ***é parte de*** um objeto carro Um objeto turma ***tem vários*** objetos alunos. Um objeto botão ***tem um*** objeto tratador de eventos

Tipos de associações entre objetos de software:

- ***Agregação***: estabelecem um vínculo entre objetos
- ***Composição***: relacionamento do tipo *todo/parte*.
- ***Uso***: um objeto usa a funcionalidade de outro sem estabelecer vínculo duradouro (referências)

Agregação

- Forma de composição em que o objeto composto apenas usa ou tem conhecimento da existência do(s) objeto(s) componente(s)
- Os objetos componentes podem existir sem o agregado e vice-versa.

Composição

- Forma de associação em que o objeto composto é

  responsável pela existência dos componentes O componente não tem sentido fora da composição.

Herança

A herança é um tipo de relacionamento que define que uma classe "é um" de outra classe como, por exemplo, a classe `Funcionario` que é uma Pessoa, assim um Funcionário tem um relacionamento de herança com a classe Pessoa. Em algumas linguagens, como `C`, é possível fazer herança múltipla, isto é, uma classe pode herdar de diversas outras classes, mas em Java isso não é permitido, pois cada classe pode herdar de apenas outra classe.

A **Listagem 7** mostra como implementar a herança entre classes: a classe Pessoa definida será utilizada como base para as outras classes. Será definida a classe `Funcionario`, com os atributos salario e matricula, e a classe Aluno com os atributos `registroAcademico` e curso. Como é possível observar, para implementar herança em Java usaremos a palavra reservada `extends`.

```
public class Funcionario extends Pessoa {

	private float salario;
	private String matricula;

	public float getSalario() {
		return salario;
	}

	public void setSalario(float salario) {
		this.salario = salario;
	}

	public String getMatricula() {
		return matricula;
	}

	public void setMatricula(String matricula) {
		this.matricula = matricula;
	}

}


public class Aluno extends Pessoa {

	private String registroAcademico;
	private String curso;
	private float mensalidade;

	public String getRegistroAcademico() {
		return registroAcademico;
	}

	public void setRegistroAcademico(String registroAcademico) {
		this.registroAcademico = registroAcademico;
	}
```

```
	public String getCurso() {
		return curso;
	}

	public void setCurso(String curso) {
		this.curso = curso;
	}

	public float getMensalidade() {
		return mensalidade;
	}

	public void setMensalidade(float mensalidade) {
		this.mensalidade = mensalidade;
	}

}
```

Listagem 7. Classes Funcionario e Aluno, filhas da classe Pessoa

Sobre a herança em Java é importante saber que existem quatro tipos de acesso aos atributos:

1. public: que permite que métodos e atributos sejam acessados diretamente de qualquer classe;
2. private: que permite que métodos e atributos sejam acessados apenas dentro da classe;
3. protected: que permite que métodos e atributos sejam acessados apenas dentro da própria classe e em classes filhas;
4. tipo de acesso padrão, que permite que métodos e atributos sejam acessadas por qualquer classes que esteja no mesmo pacote.

Como usamos a classe Pessoa com herança, uma possibilidade é mudar o tipo de acesso dos atributos da classe Pessoa para protegido. A **Listagem 8** mostra a alteração feita.

```
package com.devmedia.model;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Pessoa {

	protected String nome;
	protected String endereco;
	protected String telefone;
	protected Date dataNascimento;

	// Relacionamento com a classe Carro
	protected List<Carro> carros = new ArrayList<Carro>();

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}
```

```
	public String getEndereco() {
		return endereco;
	}

	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}

	public String getTelefone() {
		return telefone;
	}

	public void setTelefone(String telefone) {
		this.telefone = telefone;
	}

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public List<Carro> getCarros() {
		return carros;
	}

	public void setCarros(List<Carro> carros) {
		this.carros = carros;
	}

}
```

Listagem 8. Classe Pessoa com os atributos protegidos

Polimorfismo

O Polimorfismo é a possibilidade de em uma hierarquia de classes implementar métodos com a mesma assinatura e, assim, implementar um mesmo código que funcione para qualquer classe dessa hierarquia sem a necessidade de implementações específicas para cada classe. O principal objetivo do polimorfismo é diminuir a quantidade de código escrito, aumentando a clareza e a facilidade de manutenção.

A **Listagem 9** mostra um exemplo de polimorfismo, onde a classe Conta tem um método saque que apenas permite o saque caso o cliente tenha saldo em sua conta. A classe `ContaEspecial`, que é filha da classe Conta, permite que o cliente faça um saque caso tenha saldo ou tenha ainda limite no cheque especial, e a classe `ContaPoupanca` permite que o cliente faça um saque caso ainda tenha saldo na conta corrente ou saldo na conta poupança.

```
public class Conta {

	protected String agencia;
	protected String numero;
	protected float saldo;
```

```
	public void saque(float valor) {
		if (saldo < valor) {
			System.out.println("O saldo é insuficiente!");
		} else {
			saldo -= valor;
System.out.println("Saque efetuado com sucesso!");

		}
	}

	public String getAgencia() {
		return agencia;
	}

	public void setAgencia(String agencia) {
		this.agencia = agencia;
	}

	public String getNumero() {
		return numero;
	}

	public void setNumero(String numero) {
		this.numero = numero;
	}

	public void deposito(float valor) {
		saldo += valor;
	}

	public void saldo() {
		System.out.println("O saldo é: " + saldo);
	}

}

public class ContaEspecial extends Conta {

	private float limite;

	public void saque(float valor) {
		if (saldo + limite < valor) {
			System.out.println("O saldo é insuficiente!");
		} else {
			saldo -= valor;
			System.out.println("Saque efetuado com sucesso!");

		}
	}

	public float getLimite() {
		return limite;
	}

	public void setLimite(float limite) {
		this.limite = limite;
	}

}

public class ContaPoupanca extends Conta {

	private float saldoPoupanca;
```

```
      public void saque(float valor) {
            if (saldo + saldoPoupanca < valor) {
                  System.out.println("O saldo é insuficiente!");
            } else {
                  if (saldo < valor) {
                        valor -= saldo;
                        saldo = 0;
                        saldoPoupanca -= valor;
                  } else {
                        saldo -= valor;
                  }
            System.out.println("Saque efetuado com sucesso!");
            }
      }

    public void depositoPoupanca(float valor) {
            saldoPoupanca += valor;
      }

    private float getSaldoPoupanca() {
            return saldoPoupanca;
      }

      private void setSaldoPoupanca(float saldoPoupanca) {
            this.saldoPoupanca = saldoPoupanca;
      }

}
```

Listagem 9. Hierarquia de classes Conta

A **Listagem 10** mostra o uso das classes, apesar de serem definidos objetos dos três tipos de classes. O método fazSaque pode ser utilizado para qualquer um dos tipos de conta, não dependendo do tipo que é passado como parâmetro para ele, e isso é possível graças ao polimorfismo, pois o método saque das três classes tem a mesma assinatura e em tempo de execução a JVM sabe o método saque de qual classe executar.

```
public class MainConta {

      public static void main(String [] args) {

            Conta conta = new Conta();
            conta.setAgencia("1234");
            conta.setNumero("1244");
            conta.deposito(3000);

            ContaPoupanca contaPoupanca = new ContaPoupanca();
            contaPoupanca.setAgencia("3456");
            contaPoupanca.setNumero("1234");
            contaPoupanca.deposito(3000);
            contaPoupanca.depositoPoupanca(1000);

            ContaEspecial contaEspecial = new ContaEspecial();
            contaEspecial.setAgencia("3432");
            contaEspecial.setLimite(1000);
            contaEspecial.setNumero("1434");
            contaEspecial.deposito(3000);

            fazSaque(conta);
            fazSaque(contaPoupanca);
```

```
            fazSaque(contaEspecial);

      }

      public static void fazSaque(Conta conta) {

            float valor =
                        Float.parseFloat(
                        JOptionPane.showInputDialog("Digite o valor do saque")
                        );
            conta.saque(valor);
            conta.saldo();
      }

}
```

Listagem 10. Utilização das classes conta

3 Padrões de Projeto

Padrões de projeto podem ser vistos como uma solução que já foi testada para um problema. Desta forma, um padrão de projeto geralmente descreve uma solução ou uma instância da solução que foi utilizada para resolver um problema específico. Padrões de projetos são soluções para problemas que alguém um dia teve e resolveu aplicando um modelo que foi documentado e que você pode adaptar integralmente ou de acordo com necessidade de sua solução.

Em geral, um padrão tem quatro elementos essenciais:

1 – **Nome do padrão:** é uma referência que podemos usar para descrever um problema de projeto, suas soluções e consequências em uma ou duas palavras.

2 – **O problema:** explica o problema e seu contexto.

3 – **A solução:** descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações.

4 – **As conseqüências:** resultados e análises das vantagens e desvantagens (trades-off) da aplicação do padrão.

Podemos considerar que a idéia original de **Design Patterns** surgiu com Christopher Alexander (Engenheiro Civil) quando ele propôs a criação de catálogos de padrões para arquitetura no ano de 1978.

Com a palavra Christopher Alexander:

"*Um padrão descreve um problema que ocorre inúmeras vezes em determinado contexto, e descreve ainda a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações.*"

Descrevendo os padrões de projeto

As notações gráficas, embora sejam importantes e úteis, não são suficientes. Elas simplesmente capturam o produto final do processo de projeto como relacionamentos entre classes e objetos. Para reutilizar o projeto, nós devemos registrar decisões, alternativas e análises de custos e benefícios que levaram a ele. Também são importantes exemplos concretos, porque ajudam a ver o projeto em ação.

Nós descrevemos padrões de projeto usando um formato consistente. Cada padrão é dividido em seções de acordo com o **gabarito** a seguir. O gabarito fornece uma estrutura uniforme às informações, tornando os padrões de projeto mais fáceis de aprender, comparar e usar:

1. Nome e classificação do padrão
2. Intenção e objetivo
3. Também conhecido como
4. Motivação
5. Aplicabilidade
6. Estrutura
7. Participantes
8. Colaborações
9. Consequências
10. Implementação
11. Exemplo de código
12. Usos conhecidos
13. Padrões relacionados

Principais propriedades dos padrões de projetos

Dentre as principais propriedades dos padrões de projetos podemos citar:

1. Capturam o conhecimento e a experiência de especialistas em projeto de software.
2. Especificam abstrações que estão acima do nível de classes ou objetos isolados ou de componentes.
3. Definem um vocabulário comum para a discussão de problemas e soluções de projeto.
4. Facilitam a documentação e manutenção da arquitetura do software.

5. Auxiliam o projeto de uma arquitetura com determinadas propriedades.

6. Auxiliam o projeto de arquiteturas mais complexas.

Quando e como utilizar padrões de projetos

A primeira coisa que você tem que ter é bom senso. Faça a sua parte, implemente sua solução e veja se ela funciona. A seguir verifique se ela pode ser otimizada, se for o caso utilize o padrão de projeto que se ajusta ao seu caso para melhorar as deficiências verificadas no seu projeto.

Naturalmente isto será tão mais fácil se você tiver uma visão global do seu projeto e seu funcionamento.

Os padrões de projeto não são uma varinha mágica que vai tornar o seu projeto isento de falhas. Se for mal implementado eles podem até diminuir a compreensão do seu projeto e aumentar a quantidade de código. Portanto padrões de projeto não resolvem todos os problemas de design de projetos.

Os componentes de um padrão de projeto são :

1- **Nome** - descreve a essência do padrão.
2- **Objetivo** - descreve como o padrão atua.

3- **Problema** - descreve o problema.

4- **Solução** - descreve a solução.

5- **Conseqüências** - descreve os benefícios da utilização do padrão.

Requisitos de um bom sistema de padrões:

O sistema deve conter uma boa quantidade de padrões. A descrição dos padrões deve seguir um formato padronizado. O sistema deve ser estruturado, organizando os padrões seguindo critérios bem definidos.

Existem 23 padrões divididos em: Criação, Estruturais e Comportamentais. Além destes padrões também existem os padrões GRASP (General Responsibility Assignment Software Patterns).

		PROPÓSITO		
Escopo		De Criação	Estrutural	Comportamental
	Classe	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