Padrões de Projeto

Prof. Rafael Milbradt

Contato: rmilbradt@gmail.com

Ementa

- OBJETIVOS:
  - Conhecer técnicas avançadas em orientação a objetos.
  - Conhecer e aplicar padrões de projeto no desenvolvimento orientado a objetos.
  - Capacidade de desenvolver código reutilizável.
  - Conhecer técnicas de refatoração de código.

Ementa

UNIDADE 1 – ORIENTAÇÃO A OBJETOS AVANÇADA
- 1.1 - Revisão de conceitos de orientação a objetos.
- 1.2 – Concorrência.
- 1.3 - Tratamento de exceções.
- 1.4 – Criação de código dinâmico: reflexão e outras técnicas.
- 1.5 – Padronização de código fonte.

UNIDADE 2 – PADRÕES DE PROJETO
- 2.1 – Introdução.
- 2.2 - Tipos de padrões (análise, projeto, banco de dados, programação, entre outros).
- 2.3 - Padrões de projeto de criação.
- 2.4 – Padrões de projeto de estruturais.
- 2.5 – Padrões de projeto de comportamentais.

Ementa

UNIDADE 3 – REUTILIZAÇÃO DE SOFTWARE
- 3.1 – Boas práticas para a produção de software reutilizável.
- 3.2 – Reuso de componentes de software.
- 3.3 – Frameworks – Reutilização de software de terceiros.

UNIDADE 4 – PADRÕES DE PROJETO EM SISTEMAS PARA INTERNET
- 4.1 – Visão geral.
- 4.2 – Padrões da camada de apresentação.
- 4.3 – Padrões da camada de negócio.
- 4.4 – Padrões da camada de integração.

UNIDADE 5 – EVOLUÇÃO DE SISTEMAS ORIENTADOS A OBJETOS
- 5.1 – Princípios de evolução e refatoração de sistemas orientados a objetos.
- 5.2 – Identificando a necessidade de evolução e refatoração.
- 5.3 – Catálogos de refatoração de projeto orientado a objetos.

Bibliografia

BIBLIOGRAFIA BÁSICA
- GAMMA, E. **Padrões de Projeto: soluções reutilizáveis de software orientado a objetos.** Porto Alegre: Bookman, 2007**.**
- PREISS, B**. Estruturas de dados e algoritmos : padroes de projetos orientados a objetos com Java.** Rio de Janeiro: Elsevier, 2001.
- COIMBRA, E.; GUIZZO, G.; LAMB, J. R. **Padrões de Projeto Em Aplicações Web**. Ed. Visual Books, 2013.

BIBLIOGRAFIA COMPLEMENTAR
- BLOCH, J. **Effective Java.** Addison-Wesley, 2008.
- SANDERS, William. **Aprendendo padrões de projeto em PHP**. São Paulo: Novatec, 2013.
- MAHEMOFF, Michael. **Padrões de projetos Ajax.** Rio de Janeiro: AltaBooks, 2007.
- ALUR, D. **Core J2EE Patterns: Best Practices and Design Strategies.** Prentice Hall: 2ª ed., 2003.
- PREE, W. **Design patterns for object-oriented software development**. Addison-Wesley, 1995.
- Apostilas CAELUM;

Avaliações

- Avaliação 1 (Peso 10):
  - Avaliação teórica e prática;
- Avaliação 2 (Peso 10):
  - Avaliação teórica e prática;

- Exame:
  - Avaliação teórica e prática;

Revisão OO

- Algumas observações sobre a linguagem Java:
- Pilha (stack):
  - Alocação de espaço de memória utilizado no escopo de métodos;
  - Apenas referências e tipos primitivos vão para a pilha;
- Monte (heap):
  - Alocação de objetos (new) é feita sempre no monte;
  - Não existe um comando free(), ao invés disso a JVM usa uma estratégia de Garbage Collector.
- Referências vs. Ponteiros:
  - Afinal, qual a diferença?
  - Ponteiros são endereços de memória e permitem operações sobre estes:
    - Referenciar, derreferenciar, incrementar, decrementar, etc.
  - Referências na prática também são endereços de memória, porém a linguagem deixa tudo mais abstrato e fácil de entender:
    - Não existem operações diretamente sobre referências. Todas as operações sobre as referências na verdade são executadas sobre os próprios objetos;
    - Não preciso pensar se agora devo referenciar, derreferenciar, etc. Posso abstrair o próprio conceito da referência e imaginar estar sempre com o próprio objeto.

Revisão OO

- Algumas observações sobre a linguagem Java:
- Passagem de parâmetro por valor:
  - Nesta passagem de parâmetros os valores são copiados para dentro do escopo do método;
  - Não existe efeito colateral;
- Passagem de parâmetro por referência:
  - Neste tipo de passagem de parâmetros é passada apenas a referência do objeto para o escopo da função.
  - Alterações no objeto passado por parâmetro serão percebidas no escopo do objeto que chamou o método (efeito colateral).
- Objetos imutáveis:
  - Algumas classes de tipos básicos da linguagem são imutáveis como: String, Long, Integer, Character, Double, Float, entre outros...
  - Isto significa que estes objetos, uma vez criados, nunca mais poderão ser alterados;
  - Reduz situações indesejadas, causadas por efeito colateral;
  - Evite concatenar Strings longas com o operador "+" (use StringBuilder ou StringBuffer).

Revisão OO

- Algumas observações sobre a linguagem Java:
- Linguagem totalmente orientada a objetos?
  - Não, pois ainda existem os tipos primitivos;
  - Foram deixados por motivos de desempenho e também para manter a proximidade gramatical com a linguagem C/C++;
  - int, short, long, double, float, char, byte e boolean.
  - Nenhum método pode ser invocado sobre os tipos primitivos, já que estes não são objetos (exceto autoboxing);
  - Por consequência também não existe referência para tipos primitivos e tampouco passagem por referência;
  - Não podem assumir valor "null";
  - Evite utilizá-los como propriedades para representar dados em SGBD, tendo em vista que no SGBD poderá existir "null";

Revisão OO

- Algumas observações sobre a linguagem Java:
- Mais sobre as Strings...
  - Mais de 80% da memória utilizada por uma aplicação típica em Java será composta por objetos do tipo String;
  - Muitas destas Strings serão iguais, porém objetos diferentes cada um ocupando seu espaço de memória;
  - Mas se as Strings são imutáveis, Strings iguais não poderiam usar referências para o mesmo objeto?
  - Iria facilitar a comparação:
    - Usar == é infinitamente mais rápido do que usar .equals();
    - Java > 8 usa StringPool.
  - É possível fazer isto através do método intern();
    - Retorna uma referência única para aquela String;

Revisão OO

- Algumas observações sobre a linguagem Java:
- Vamos testar?
  - Passagem de parâmetros;
  - Imutabilidade;
  - String intern;

Revisão OO

- A ideia principal por detrás da orientação a objetos é permitir adicionar comportamento aos dados;
- Em linguagens procedurais podemos criar tipos complexos de dados (structs);
  - Em programação OO podemos adicionar comportamento nestes tipos complexos de dados;
- Quando temos os tipos complexos com comportamento, temos uma Classe.
  - Instâncias desta classe são o que chamamos de Objetos.
- Dentro de POO ainda temos uma série de conceitos adicionais que iremos revisar a seguir...

Revisão OO

- **Método:** é o código que define um comportamento existente em uma classe;
  - É o análogo de procedimento e função na programação procedural, porém os métodos somente podem ser invocados em um objeto.
  - Cada classe de objetos possui seus próprios métodos.
  - Exceção: métodos podem ser estáticos, caso em que eles não são invocados sobre um objeto e sim sobre a própria classe;
    - Porém, a própria classe também é um objeto...
    - Deve-se evitar o uso de métodos estáticos, pois eles são de certa forma uma quebra do paradigma OO;

Revisão OO

- **Construtor:**
  - Cada classe possui pelo menos um construtor, que é um tipo especial de método que permite a criação de objetos da referida classe.
  - Em Java o construtor padrão (sem parâmetros) existe de forma implícita, porém pode ser alterado.
    - Caso seja criado outro construtor (com parâmetros) o construtor padrão deixa de existir de forma implícita.
  - Em alguns casos uma classe pode não possuir construtor:
    - Veremos adiante na implementação de um pattern.

Revisão OO

- **Encapsulamento:**
  - Encapsular significa implementar uma classe com todos os dados e regras necessárias para operar os mesmos dentro de um único compartimento (a própria classe), porém deixando estas regras e dados “escondidos” para os outros objetos que usarão esta funcionalidade;
  - Por que “esconder”?
    - Se os objetos que usam determinado objeto não conhecem a sua lógica, eles não são construídos de forma acoplada a implementação desta lógica;
    - Um objeto pode usar o serviço de outro, porém sem conhecer detalhes internos daquele objeto, para prover o seu serviço. Quando o objeto precisar alterar a sua implementação não haverá impacto aos demais, desde que o serviço continue o mesmo.
    - O encapsulamento permite desacoplar implementações, ou seja, objetos usam outros porém de forma desacoplada ou independente.

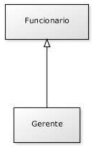
PROGRAME PARA A INTERFACE E NÃO PARA A IMPLEMENTAÇÃO!

Revisão OO

- **Encapsulamento:**
  - É implementado em Java através dos modificadores de acesso.
    - public: acesso irrestrito a atributos ou métodos;
    - private: não permite acessos externos a atributos ou métodos;
    - protected: permite acesso apenas às subclasses;
    - default (sem nenhum modificador): público apenas para as classes do mesmo pacote, para as demais é privado!
  - Um padrão de projeto muito usado (Java Bean) é manter os atributos “private” e implementar getters e setters públicos para o acesso a estes atributos:
    - Porém de forma geral o uso indiscriminado de getters e setters pode quebrar o encapsulamento.
    - Alternativas:
      - É possível fazer um método read-only não implementando o set;
      - Não esqueça dos efeitos colaterais! É possível alterar atributos não imutáveis apenas com o get!
      - Nestes casos o get pode retornar cópias do objeto;
        - » Vamos estudar padrões de projeto para resolver isto!
      - É possível deixar um set como private e ele mesmo assim ser acessado por outro objeto:
        - » Sim, se o outro objeto usar reflexão!
        - » P. ex.: Hibernate

Revisão OO

- **Herança:**
  - É uma relação onde uma classe é um subtipo de outra determinada classe:




```
class Funcionario {
    String nome;
    String cpf;
    double salario;
     // métodos devem vir aqui
}
```

```
class Gerente {
    String nome;
    String cpf;
    double salario;
    int senha;
    int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // outros métodos
}
```

  - Se um Gerente é um Funcionário a classe gerente não deveria repetir todo o código de Funcionário.

Revisão OO

- **Herança:**
  - Se um Gerente é um Funcionário a classe gerente não deveria repetir todo o código de Funcionário.

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // setter da senha omitido
}
```

Revisão OO

- **Herança:**
  - Sempre que quisermos reaproveitar código devemos usar herança?
    - **Não!!! Este é um erro muito frequente de quem usa POO;**
    - Sugestão de Leitura:
      - » http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-heranca/
      - » BLOCH, J. **Effective Java.** Addison-Wesley, 2008.
    - (maus) Ex
```
class Stack extends Vector
class Properties extends Hashtable
```
    - Uma pilha é um tipo de vetor???
    - Se um gato possui raça e patas, e um cachorro possui raça, patas e tipoDoPelo, logo "Cachorro extends Gato"? Pode parecer engraçado, mas é o **mesmo** caso que os anteriores: herança por preguiça, por comodismo. A relação "**é um**" não se encaixa aqui, e vai gerar problemas. Como?

Revisão POO

- **Herança:**
  - Claramente um Properties não deveria ser um Hashtable, afinal ela não mapeia objetos a objetos, mas acaba nos fornecendo um método put(Object, Object), que se usarmos sem passar Strings vai causar problemas.
  - Como resolver isso depois que já nos comprometemos com a herança? A única solução é colocar um aviso grande no javadoc para ninguém usar determinados métodos herdados!
  - O mesmo vale para a Stack e o Vector. Uma Stack não é um Vector, definitivamente. Se você pensar direitinho, uma pilha tem comportamento **oposto** da implementação da Vector!
  - Sempre que usarmos Herança por comodismo aberrações deste tipo vão ocorrer.

Revisão POO

- **Herança:**
  - Mau uso da herança é uma visível quebra do encapsulamento.
  - Joshua Block, em Effective Java, sugere que as classes sejam implementadas pensando que podem ser futuramente herdadas ou então a herança deve ser proibida. Como?
    - Através de "final";
      - Em métodos, não permite que sejam sobrescritos;
      - Em classes, não permite que sejam herdadas;
      - Em atributos, não permite que sejam novamente atribuídos após a inicialização.

Revisão POO

- **Herança:**
  - Mas como é o certo?
  - Joshua Block, em Effective Java, sugere: "*prefira composição em vez de herança*".
    - Um Properties é composto por uma Hashtable e outras propriedades para implementar o seu serviço. Properties **tem um** Hashtable, muito diferente de dizer que uma Properties **é um** Hashtable.
    - Desta forma a implementação do Properties com uma Hashtable fica totalmente encapsulada dentro da classe. Quem usar o serviço não deve acessar diretamente o Hashtable.
  - Em linhas gerais: **evite o uso de herança.**
  - Mas isto é um problema apenas do Java, certo?
    - Não. Este problema está relacionado com a POO em si.

Revisão POO

- **Reescrita de método:**
  - É a forma como uma subclasse pode mudar o comportamento da superclasse;

```
class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 0.10;
    }
    // métodos
}
```

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return this.salario * 0.15;
    }
    // ...
}
```


Revisão POO

- **Reescrita de método:**
  - É possível invocar o método da superclasse, dentro da classe filha:

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return super.getBonificacao() + 1000;
    }
    // ...
}
```

Revisão POO

- **Polimorfismo:**
  - No exemplo anterior se um Gerente **é um** Funcionário como podemos referenciar o Gerente?
    - Como Funcionário ou como Gerente!
    - Se fizermos isto o que ocorre?

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
funcionario.getBonificacao();
```


Revisão POO

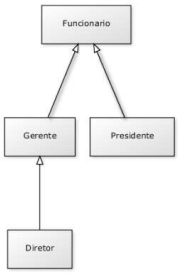
- **Polimorfismo:**

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
funcionario.getBonificacao();
```

  - A invocação do método sempre será decidida em tempo de execução: a JVM vai buscar a implementação do método getBonificacao() para o tipo real do objeto, independente da forma como está sendo referenciado.

Revisão POO

- **Classes Abstratas:**
  - No exemplo do Funcionário:
    - Pode ser útil referenciarmos objetos como Funcionario e aproveitar-se do polimorfismo;
    - Porém pode ser que o Funcionário não exista. Cada Funcionário, mesmo sendo Funcionário, deve ter seu cargo: Gerente, Diretor, Presidente, etc.



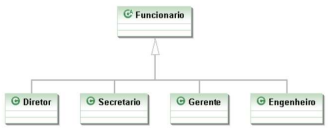
Revisão POO

- **Classes Abstratas:**
  - Qual a vantagem das classes abstratas:
    - Mesmo que não exista um Funcionário, comportamentos que são comuns a todos os funcionários podem ser implementados na classe abstrata.
    - Comportamentos específicos de cada Funcionário são implementados nas classes herdeiras:
      - São declarados como abstratos na classe abstrata, o que vai forçar a classe que estender aquela a implementar estes método

```
abstract class Funcionario {

    abstract double getBonificacao();

    // outros atributos e métodos

}
```


Revisão POO

- **Interfaces:**
  - No exemplo anterior:
    - Imagine que todos os funcionários precisam ter um método que os autentica no sistema;
    - Fácil, colocamos ele como abstrato na classe Funcionário!
    - O sistema de login chama o método da classe Funcionário, referenciando todos como Funcionario.
    - Alguém teve a ideia de implementar um "portal do cliente", sendo que cada cliente também precisará de autenticação;
    - E agora? Cliente vai estender Funcionário?



Revisão POO

- **Interfaces:**
  - Podemos criar uma Interface "Autenticavel" na qual todas as classes que tem autenticação deverão implementar;
  - Interfaces são como classes Abstratas, porém:
    - São apenas um contrato do que deve ser implementado por quem herda;
    - Não permitem implementação;
    - No Java são a única forma de implementar herança múltipla:
      - Uma interface pode estender várias;
      - Uma classe concreta pode implementar várias interfaces, por outro lado só pode estender uma classe;
      - No Java 8 interfaces podem implementar métodos concretos com a palavra "default".

Revisão POO

- **Interfaces:**
  - Embora a primeira vista escrever interfaces pareça retrabalho, as interfaces podem ser muito úteis para ajuda a escrever código reaproveitável sem o uso da Herança, que pode acarretar em problemas maiores:
  - As duas regras de ouro em orientação a objetos:
    1. "evite herança, prefira composição"
    2. "programe voltado a interface e não a implementação"
  - A 2ª regra reforça o conceito mais importante da POO: **encapsulamento**.

Revisão POO

- Exercício 1:
  - Modelar e implementar um sistema de representação de um imóvel juntamente com o cálculo de área total construída do imóvel, bem como o seu volume interno;
  - O imóvel deverá ter um conjunto de peças, sendo que cada uma deverá calcular a sua área com base nas formas geométricas que compõem a mesma;
  - Deverão existir as seguintes formas:
    - (semi)Círculo, quadrado, losango, retângulo e triângulo;
    - Cada forma deve contar com as medidas necessárias para implementar um método que calcula a sua área;
  - Cada peça deverá ter o seu pé-direito de forma que se possa calcular o volume da mesma;
  - Peças deverão ter identificação;
  - O imóvel deverá possuir identificação, tipo de uso (comercial, residencial), proprietário e endereço;

Revisão POO

- Exercício 2:
  - Crie uma classe banco que armazene um conjunto de contas e forneça métodos que permitam que sejam feitos criações de conta, exclusão de contas, saques (uma conta corrente só pode fazer saques desde que o valor não exceda o limite de saque -limite + saldo-), depósitos, emissão de saldo e extrato e transferência entre contas.
  - Uma conta possui um número, um saldo, um status que informa se ela é especial ou não, um limite e um conjunto de movimentações. Uma movimentação possui uma descrição, um valor e uma informação se ela é uma movimentação de crédito, débito ou de rendimento financeiro.
  - Além disto as contas podem ser do tipo: Poupança, Conta-corrente, Fundos de Renda Fixa ou Fundos de Renda Variável. Dentre estes apenas os rendimentos de renda fixa e variável são tributados pelo IR, nestes casos deverá existir um método que calcula o valor do imposto devido com base no rendimento financeiro do mês e a alíquota de 27,5%.