UNIDADE III

Aplicações de Linguagem de Programação Orientadas a Objetos

Prof. Me. Lauro Tomiatti

Programação não é apenas código

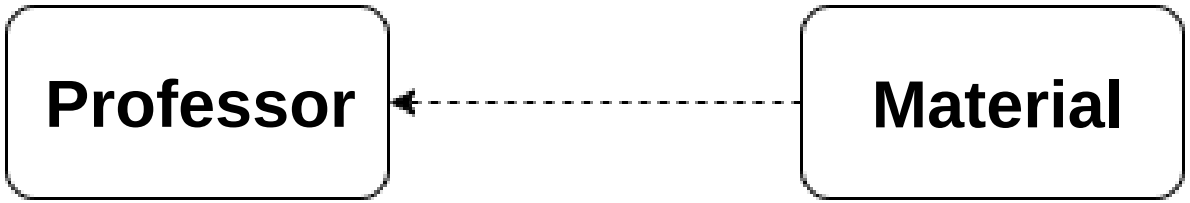
- Instruções não são tudo.
- Engenharia de *software* propõe projetar e guiar o desenvolvimento.
- Com isso, notamos especificações que se aplicam a diversos casos.

Relações entre objetos

- Toda estrutura composta de um projeto cria dependências e relacionamentos que não são necessariamente heranças ou implementações.

- O diagrama UML abaixo simboliza que a aula depende de um professor.

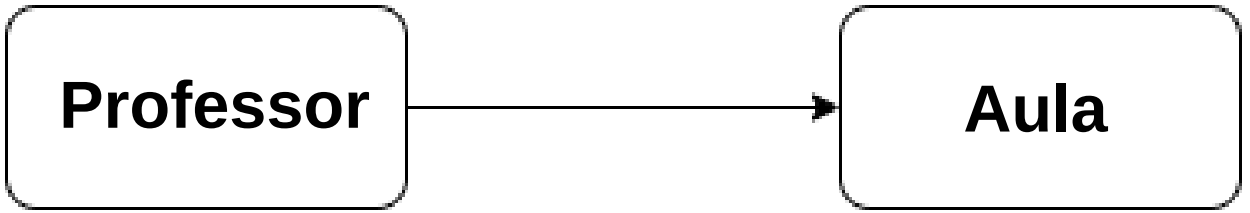
- A dependência é a mais básica e fraca das relações entre as classes, geralmente os diagramas não demonstram todas as dependências, pois podem poluí-los.



Fonte: Autoria própria

Associações

- A associação é um relacionamento no qual um objeto interage com outro objeto.
- As associações podem ser bidirecionais.
- Nesse relacionamento, um objeto sempre tem acesso ao objeto com o qual ele interage.



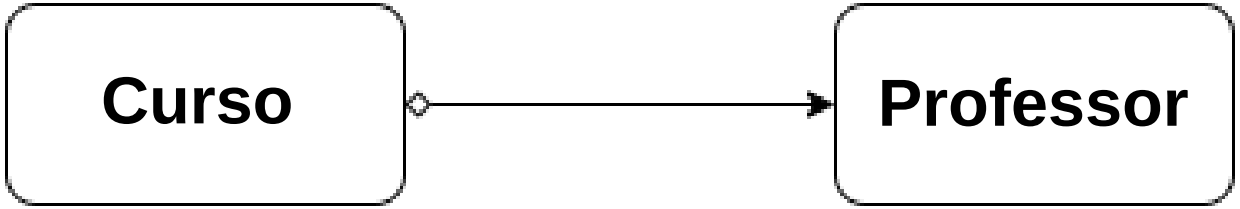
Fonte: Autoria própria

Agregação

- Agregação diz respeito à associação especializada que representa um para muitos, muito para muitos, entre outros, sobre diversos objetos.

- Geralmente, um objeto-chave possui uma coleção de outros objetos e serve como um contêiner para essa categoria.

- UML representa uma relação entre classes, isso significa que apesar de vermos apenas um professor, está implícito que no cenário de execução pode haver diversas instâncias de professor.



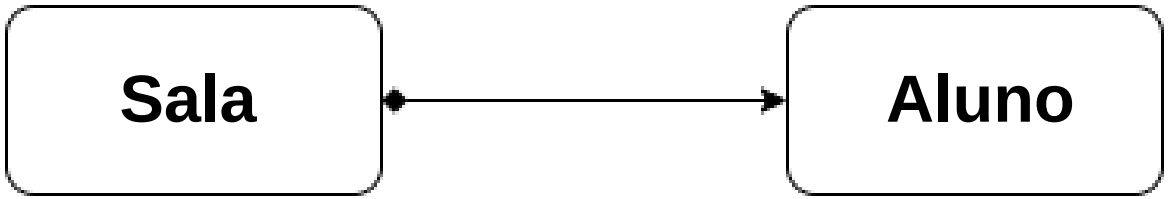
Fonte: Autoria própria

Composição

- A composição é um tipo específico de agregação, em que um objeto é composto de uma ou mais instâncias da outra.

- A distinção se dá pelo fato do componente só existir como parte de um contêiner.

- Esse termo diversas vezes é utilizado para expressar agregação, devido ao fato de soar mais natural para nossa conversa.



Fonte: Autoria própria

Visualizando o cenário

```java
public class Curso {

    List<Professor> professores;
    List<Material> materiais;
    Coordenador coordenador;

}
```

```java
public class Professor {

    Material material;

    public Professor(Material material) {
        this.material = material;
    }

    public Aula preparaAula() {
        Aula aulaUm = material
                .getConteudo(material.PRIMEIRA_AULA);

        return aulaUm;
    }

}
```

```java
public class Sala {

    List<Aluno> alunos;

}
```

Fonte: Autoria própria

Um bom *design*

- Antes de prosseguirmos com os padrões, vamos falar sobre um bom processo de criação de arquiteturas de *software.*

- Reutilização de código.

- Mudanças e extensões.

- Segurança de encapsulamento.

    - Boas práticas com interfaces (programar para interface e não para implementação).

    - Priorize composição em vez de herança.

Interatividade

Dentro do conhecimento de desenvolvimento de *software*, herança veio para resolver qual problema?

a) Custo de memória, que precisava alocar o dobro por dois componentes que possuem os mesmos atributos.
b) Para competir com as interfaces que estavam sendo usadas com mais afinco.
c) O problema de ler o código com dificuldade.
d) Falta de reaproveitamento de código.
e) A dificuldade de alteração de código já criado.

Resposta

Dentro do conhecimento de desenvolvimento de *software*, herança veio para resolver qual problema?

a) Custo de memória, que precisava alocar o dobro por dois componentes que possuem os mesmos atributos.
b) Para competir com as interfaces que estavam sendo usadas com mais afinco.
c) O problema de ler o código com dificuldade.
d) Falta de reaproveitamento de código.
e) A dificuldade de alteração de código já criado.

Padrões de projeto

- Os *design patterns* são soluções típicas para problemas recorrentes em *design* de *software*.
- São receitas pré-montadas que podem ser personalizadas para resolver problemas recorrentes no seu código.
- O padrão não é um código específico, não é possível copiar e colar um padrão de projeto, se isso fosse possível, seria uma dependência de projeto.

Padrões de projeto

- Assim, a partir das melhores práticas, reuniram-se as melhores e mais eficazes soluções, transformando-as em padrões, e algumas até em *Frameworks* (pacotes prontos de códigos que fornecem alguma solução específica).

- Nesse contexto, demos passos claros para se alcançar um objetivo, por outra perspectiva, você é capaz de olhar os resultados e quais são seus passos, mas a implementação depende do desenvolvedor.

Padrões de projeto

- No desenvolvimento de um sistema, espera-se que alguns requisitos sejam garantidos, como por exemplo: desempenho (eficiência e eficácia - que suporte a necessidade do Cliente e que funcione corretamente), compreensão, facilidade na manutenção na utilização e na reutilização.
- Os *Design Patterns* surgiram da necessidade de criação, acerto ou manutenção de sistemas com agilidade, rapidez e preços competitivos.

Fonte: https://br.pinterest.com/pin/536491374352273644/

Factory

- O padrão *Factory*, ou fábrica, é um padrão de *design* que providencia uma interface para criar objetos em uma subclasse.

- Ele permite que as subclasses alterem o tipo do objeto que será criado.

- Com a mudança e melhoria na expansão do código, é possível que haja necessidade da criação de outros objetos que possuam a mesma lógica.

Factory

- Esse padrão define que a solução está em trocar chamadas diretas do método de construção (*new*) por um método da fábrica.

- Em um cenário de banco de dados, podemos criar uma fábrica de objetos de um tipo específico.

Abstract Factory

- O padrão *Factory* é um padrão de *design* de criação que produz uma família de objetos sem especificar suas classes concretas.

- Em um sistema de um jogo, podemos imaginar a *Abstract Factory* como uma fábrica de inimigos, que é uma classe abstrata.

- Temos a implementação de uma classe *Factory* para cada tipo de objeto que ela deve trazer; sendo assim, o retorno em comparação à interface não muda.

Adapter

- É um padrão de *design* estrutural que permite o uso de objetos que, com interfaces incompatíveis, colaboram.

- Podemos pensar como a analogia às tomadas de decisão do mundo real.

- Com esse padrão, convertemos uma interface de um objeto para que outro objeto possa compreendê-la.

Adapter

- Podemos aplicar um exemplo disso em um *e-commerce* do qual uma entidade bem feita em nosso banco de dados utiliza *Enums*, porém, ao passar para outro sistema, é necessário utilizar uma letra do alfabeto para referenciar se já foi vendido, se está em promoção etc.

Interatividade

Dentro do conceito de padrões de *design*, qual dos itens a seguir **não** se aplica?

a) A utilização de componentes estáticos para podermos acessar com mais facilidade.

b) Devemos seguir esses padrões para os *designs* que são exatamente iguais aos projetos previamente criados com eles.

c) A facilidade na resolução de um problema já conhecido.

d) Padrões de *design* são vastamente conhecidos, então facilita-se a passagem de conhecimento.

e) Padrões de *design* também facilitam a visualização da aplicação como um todo, não focando apenas em uma classe.

Resposta

Dentro do conceito de padrões de *design*, qual dos itens a seguir **não** se aplica?

a) A utilização de componentes estáticos para podermos acessar com mais facilidade.

b) Devemos seguir esses padrões para os *designs* que são exatamente iguais aos projetos previamente criados com eles.

c) A facilidade na resolução de um problema já conhecido.

d) Padrões de *design* são vastamente conhecidos, então facilita-se a passagem de conhecimento.

e) Padrões de *design* também facilitam a visualização da aplicação como um todo, não focando apenas em uma classe.

Builder

- É um padrão de projeto para criações que permite a construção de objetos complexos parte por parte.

- Esse padrão permite criar diferentes tipos e representações de um objeto usando o mesmo código de construção. O segredo é extrair esse código para sua própria classe.

- O sistema, às vezes, pode se tornar muito complexo pelos diferentes tipos de variação que um objeto pode ter. Os *builders* diferentes também podem executar de maneira diferente.

Builder

- Podemos aplicar um exemplo disso em uma loja de peças de vestuário, em que algumas devem preencher certos requisitos e especificações de tamanho que não fazem sentido em outras.

Iterator

- É um padrão de projeto comportamental que permite passar por elementos de uma coleção sem expor a sua representação (a própria coleção). Melhor utilizado em estruturas de dados complexas como árvores, pilhas e gráficos.

- A ideia por trás do *Iterator* é extrair o comportamento de uma coleção em um objeto separado, assim encapsulando os detalhes como posição atual ou quantos elementos.

Iterator

- Podemos inclusive, implementar nosso próprio algoritmo. Possibilitando a criação de diversas interfaces para trabalhar de maneira diferente.

- Podemos pensar em um GPS que pega pontos distintos e deve descobrir o melhor caminho de diversas maneiras.

Observer

- É um padrão de *design* comportamental que define um mecanismo de observação denominado inscrição para notificar objetos sobre qualquer evento que pode ocorrer ao objeto que estão observando.

- Em alguns casos, seria complicado colocar um método para buscar os valores de um objeto em todos os momentos para validar uma mudança.

- Em contrapartida, se uma alteração no objeto possuísse uma maneira de chamar todos os métodos de todos os outros objetos criados. Nenhuma dessas abordagens é cabível.

Observer

- Dessa maneira, um objeto que possua um estado de interesse é chamado de editor (*Publisher*). Todos os outros objetos que gostariam de verificar essas mudanças são chamados de inscritos (*Subscribers*).

- Esse padrão sugere que seja adicionado um mecanismo de inscrição para que objetos individuais possam entrar, permanecer ou deixar de serem inscritos.

- Podemos imaginar uma tela feita em AWT ou *Swing* que quando um componente do tipo caixa de texto é alterado, ele deve sumir com o texto padrão ou captar se há caracteres não alfanuméricos.

Singleton

- É um padrão de projeto de criação para permitir que uma classe possua apenas uma instância, enquanto providencia a ela acesso global.

- Esse padrão promove que uma classe tenha apenas uma instância e acesso global, isso permite utilização de atributos únicos em toda aplicação, os usos principais são relacionados ao compartilhamento de recursos como um arquivo salvo.

Singleton

- Nesse caso, devemos fazer o construtor ser privado, para impedir outras classes de instanciar esse objeto com *new*. Após isso, criar um método estático que age como um construtor, dentro dele, há a chamada do construtor privado e salva em um atributo estático. Todas as próximas chamadas a esse método retornam o objeto pego.

- Podemos relacionar o uso desse *design* com uma única instância de conexão ao banco de dados.

Strategy

- É um padrão comportamental que permite a definição de uma família de algoritmos, separando cada uma em uma classe e fazendo o objeto adaptável.

- Esse padrão sugere que criemos uma classe que resolva uma especificidade de maneiras diferentes, extraindo seus algoritmos para classes separadas.

- Temos uma classe contexto que terá um atributo interface das estratégias que ele irá utilizar, e o seu método delegará o trabalho para a estratégia ligada ao objeto em vez de executar por si só.

Strategy

- Essa classe contexto não é responsável por selecionar um algoritmo apropriado para o trabalho, em vez disso, ela recebe uma das estratégias, ela não tem conhecimento nem do que as estratégias devem fazer.

- Podemos verificar isso com o exemplo de meios de pagamento.

Interatividade

Queremos criar uma classe que possa ser instanciada com apenas alguns atributos preenchidos, para isso usamos o *design pattern*:

a) Observer.
b) Factory.
c) Strategy.
d) Singleton.
e) Builder.

Resposta

Queremos criar uma classe que possa ser instanciada com apenas alguns atributos preenchidos, para isso usamos o *design pattern*:

a) Observer.
b) Factory.
c) Strategy.
d) Singleton.
e) Builder.

Solid

- Princípios da programação orientada a objetos voltados ao desenvolvimento de *software* para facilitar compreensão e extensão.

- Deve ser aplicável a qualquer linguagem orientada a objetos.

Solid

- Single Responsibility Principle (Princípio da Responsabilidade Única)
- Open-Closed Principle (Princípio Aberto-Fechado)
- Liskov Substituition Principle (Princípio da Substituição de Liskov)
- Interface Segregation Principle (Princípio da Segregação de Interface)
- Dependency Inversion Principle (Princípio da Inversão de Dependência)

Princípio da Responsabilidade Única

- Uma classe deve ter uma, e somente uma, responsabilidade coesa. Apenas um motivo para mudar.

- Seus atributos e métodos devem fazer sentido para completar apenas uma execução.

- Há implementações desse conceito em casos de uso.

Princípio da Responsabilidade Única

- Uma classe com uma responsabilidade terá muito menos casos de teste.
- Menos funcionalidade em uma única classe terá menos dependências.
- Classes menores e bem organizadas são mais fáceis de pesquisar do que as monolíticas.
- Podemos usar de exemplo uma classe livro que possui métodos como pesquisar palavras e mudar palavra no texto.

Princípio Aberto-Fechado

- As classes devem ser abertas para extensão e fechadas para modificação.

- Impedimos de modificar código existente e assim causar erros inesperados e inexistentes previamente no código, com exceção de abrir a classe para corrigir seus erros existentes.

O acoplamento existe, como podemos mitigá-lo?

Princípio Aberto-Fechado

- É difícil mudar interfaces bem feitas e estáveis (como *List*), isso as torna mais robustas.

- Assim, podemos estender interfaces que serão mais difíceis de serem alteradas, e quanto mais existir implementações, mais difícil é propagar suas alterações e elas provavelmente são fáceis e bem definidas.

- Podemos pensar em tipos de pagamento para auxiliar nessa abstração.

Princípio da Substituição de Liskov

- Se uma classe é subtipo de outra classe, devemos ser capazes de substituir B por A sem interromper o comportamento do nosso programa.

- Ao usar herança, precisamos pensar o que isso implica em relação aos casos existentes à classe mãe.

- Podemos pensar que um dos maiores causadores de problema são as condições da classe filho, que não devem ser mais fortes que as da classe mãe.

Princípio da Substituição de Liskov

- As classes que herdam são totalmente dependentes das classes das quais elas reaproveitam o código.

- Também podemos facilitar esse princípio por meio de composição em vez de depender de encapsulamento.

- Como exemplo, podemos pensar em uma classe Conta Salário que é filha de uma classe Conta com uma renda previamente estipulada.

Princípio da Segregação de Interface

- Interfaces maiores devem ser divididas em menores.
- As implementações só devem se preocupar com os métodos que são de seu interesse.
- Devemos aproveitar a capacidade de implementar diversas interfaces em uma classe.

Princípio da Segregação de Interface

- Se tratarmos interfaces como classes, também devemos cuidar de suas responsabilidades claras e simples.

- Podemos pensar para esse princípio o caso de um sistema que computa compras, e sua interface sabe processar, confirmar pagamento cartão, confirmar pagamento dinheiro e confirmar pagamento boleto.

Princípio da Inversão de Dependência

- Em vez de módulos de alto nível dependerem de módulos de baixo nível, ambos dependerão de abstrações.

- Abstrações não devem depender de detalhes, detalhes devem depender de abstrações.

- O primeiro passo é transformar atributos concretos em abstratos, assim quem deve se preocupar com o objeto é quem deseja instanciar essa classe; se ela preencher os requisitos, isso se chama injeção de dependência (não inversão de dependência).

Princípio da Inversão de Dependência

- Se uma classe utiliza um objeto que é uma abstração, ela não precisa se preocupar com o que vai receber, ela precisa saber somente que sabe receber e trabalhar com ele.

- Podemos usar de exemplo um sistema para montar *hardware*. Ele recebe abstrações dos dispositivos e não o componente em si.

Interatividade

Uma classe criada seguindo os princípios só deve ser estendida, assim evitamos problemas de compatibilidade. Isso devido a qual princípio?

a) Responsabilidade única.
b) Aberto-fechado.
c) Substituição de Liskov.
d) Segregação de interfaces.
e) Inversão de dependência.

Resposta

Uma classe criada seguindo os princípios só deve ser estendida, assim evitamos problemas de compatibilidade. Isso devido a qual princípio?

a) Responsabilidade única.
b) Aberto-fechado.
c) Substituição de Liskov.
d) Segregação de interfaces.
e) Inversão de dependência.

ATÉ A PRÓXIMA!