TRYBE

Modulo IV – Ciência da Computação

Bloco 36: Programação Orientada a Objetos e Padrões de Projeto

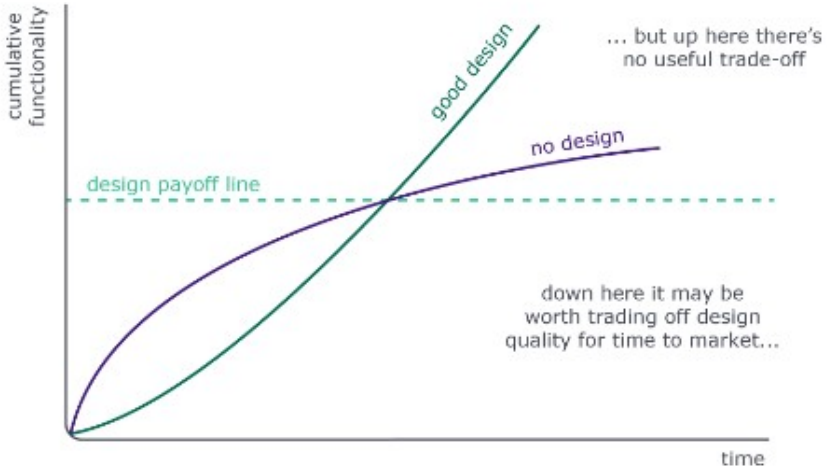


Gráfico que demonstra o esforço do uso X não uso de um design de software

POO payoff por Martin Fowler

1) Introdução à POO

Orientação a Objetos

Objetos são estruturas que armazenam informações e comportamentos, que manifestam através de **mensagens.** Mensagem é uma responsabilidade de um objeto, um método, cuidando da qualidade e da estrutura da **comunicação**.

Paradigmas de programação

Padrões independentes de linguagens especificas.

Principais paradigmas:

1/ Estruturado
Composto apenas por estruturas de programação básicas, como variáveis, funções, módulos, condicionais e laços de repetição.

2/ Orientado a objetos
Utiliza objetos para realizar a troca de mensagens na aplicação

3/ Funcional
Com base na avaliação de funções matemáticas. Efatiza a utilização de funções e expressões, que definem ou evoluem o estado das informações. Dados são imutáveis, mas os estados não.

4/ Lógico
Permite a invocação de funções, que são orientadas a padrões de verificação e de objetivos. Utilizado para árvores de decisões, algoritmos associados à análise de dados e inteligência artificial.

Vantagens do POO e problemas que resolve

POO permite:
- **reaproveitar** code, organizar melhor projetos
- facilitar criar **sistemas complexos**
- **documentar**
- **mapear o mundo real** no code (code soluciona problema do cliente,…)

Solução cara para resolver problemas simples, mais barata para **problemas complexos**. Pois temos:
- objetos pequenos e intercambiáveis, que colaboram enviando mensagens entre si. As mensagens oferecem uma forma de conectar diferentes objetos para compor soluções;
- envio de mensagens que ajuda mudança de comportamento, já que podemos substituir objetos existentes por novos, que desempenham o mesmo papel;
- portanto, **facilitação da evolução** do software.

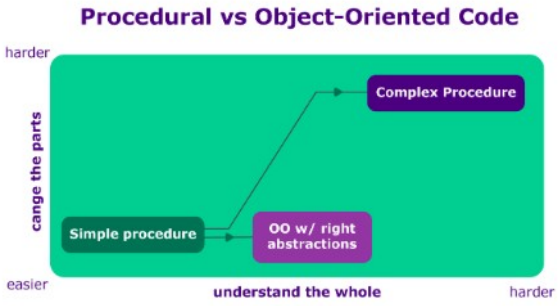


Gráfico que compara código estruturado e OO

Principais termos e conceitos da OO

Classe
Molde para criação de novos objetos, **definindo atributos** e **métodos**.
Pode definir diferentes tipos de dados.
Define o **comportamento**.

Objeto/Instância
Instanciar é o ato de criar um novo objeto/instância a partir da classe definida.
Armazena o estado.

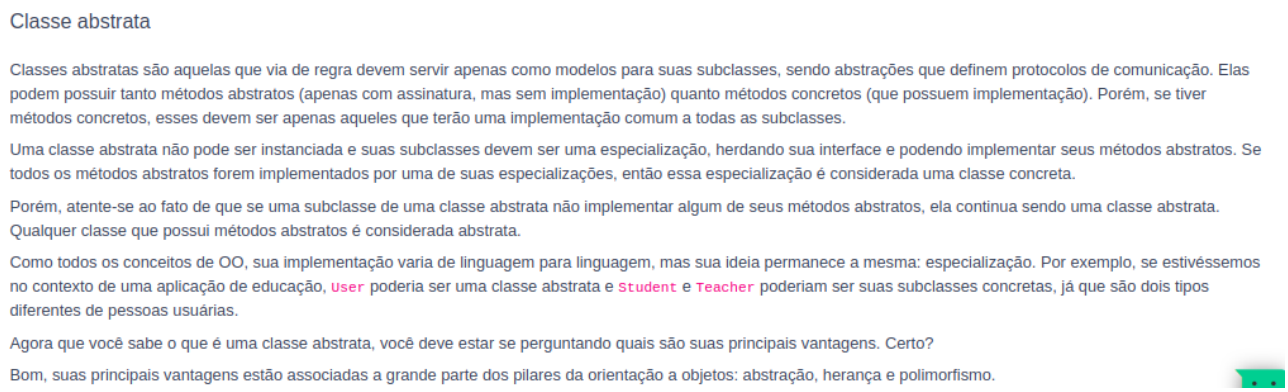
Atributo
Armazena **informações** de uma instância. **Define o estado**.

Método
Funções que possuem acesso aos dados armazenados em atributos, podendo implementar comportamentos e **alterar seus estados**.

Construtor
Método especial para *inicializar instâncias* de uma classe. Pode receber parâmetros usados para *definir as informações* armazenadas em seus atributos.

Interface/Protocolo
Representação abstrata dos protocolos de comunicação ou assinaturas.

Classe abstrata
Devem servir apenas como **modelos para subclasses**, sendo abstrações que definem protocolos de comunicação.
Podem ter **métodos** abstratos (apenas com assinatura, mas sem implementação) quanto concretos.
Não pode ser instanciada.
Vantagens: abstração, herança e polimorfismo.



Classe abstrata

Classes abstratas são aquelas que via de regra devem servir apenas como modelos para suas subclasses, sendo abstrações que definem protocolos de comunicação. Elas podem possuir tanto métodos abstratos (apenas com assinatura, mas sem implementação) quanto métodos concretos (que possuem implementação). Porém, se tiver métodos concretos, esses devem ser apenas aqueles que terão uma implementação comum a todas as subclasses.

Uma classe abstrata não pode ser instanciada e suas subclasses devem ser uma especialização, herdando sua interface e podendo implementar seus métodos abstratos. Se todos os métodos abstratos forem implementados por uma de suas especializações, então essa especialização é considerada uma classe concreta.

Porém, atente-se ao fato de que se uma subclasse de uma classe abstrata não implementar algum de seus métodos abstratos, ela continua sendo uma classe abstrata. Qualquer classe que possui métodos abstratos é considerada abstrata.

Como todos os conceitos de OO, sua implementação varia de linguagem para linguagem, mas sua ideia permanece a mesma: especialização. Por exemplo, se estivéssemos no contexto de uma aplicação de educação, `User` poderia ser uma classe abstrata e `Student` e `Teacher` poderiam ser suas subclasses concretas, já que são dois tipos diferentes de pessoas usuárias.

Agora que você sabe o que é uma classe abstrata, você deve estar se perguntando quais são suas principais vantagens. Certo?

Bom, suas principais vantagens estão associadas a grande parte dos pilares da orientação a objetos: abstração, herança e polimorfismo.

// reler depois porque foi bem.. abstrata essa parte

Pilares da OO

Abstração
Definição de **características de uma classe de forma abstrata, focando em sua interface**.
Tipo especializado até ser implementado.

Herança
Classe herda atributos e métodos de outra.
Boa prática: usar apenas para especialização, caso contrário, usar composição para compartilhar código.

Encapsulamento
Esconder parte da implementação de uma classe, exibindo de forma pública somente aquilo que é necessário para que o cliente consuma sua classe e deixando detalhes rotegidos ou privados.
Alterar apenas na classe base para não sobrescrever métodos encapsulados.

Polimorfismo
Quando duas ou mais classes contêm **implementações diferentes para métodos com a mesma interface.**

Mão na massa

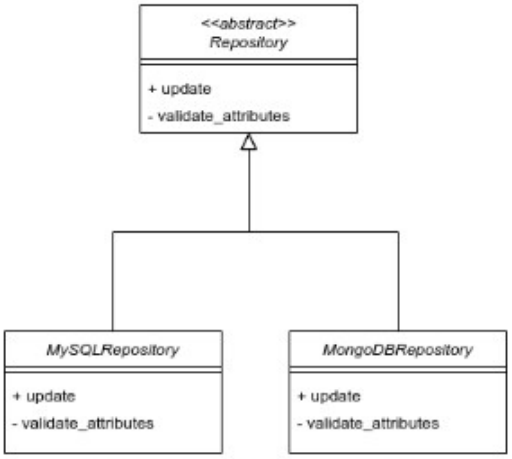


Diagrama de classes dos repositórios

// ver code exemplo, usando um Repository abstrato e dois herdeiros concretos.

1/ abc módulo python para criar classes abstratas;

2/ update(self,…), primeiro param para atualizar interface e implementar update, ao criar classes concretas

```
class MySQLRepository(Repository):
    def update(self, entity, **kwargs):
        "Sobrescreve o método update da interface, implementando a atualização"
        self._validate_attributes(entity, **kwargs)
        entity.update().where(id=entity.id).values(**kwargs)
```

3/ User pode chamar update mas não validate_attribute, conforme encapsulamento.

Dicas diversas

→ **Métodos VS. Attribution**
A(A) método
A=A attribution

→ Forma como conceitos de OO são implementados podem **variar de acordo com a linguagem** de programação.

---

2) Programação orientada a objetos na prática

Cuidado com design das abstrações: dadas desde o começo, não devem gerar mais complexidade do que flexibilidade.

Implementando o exemplo da redefinição de senha

Classe nem sempre precisa armazenar o estado para realizar o seu propósito.

→ **Método de instância versus método de classe:**
instância: métodos atrelados e executados por uma instância de uma classe
classe: métodos atrelados diretamente à classe, não dependendo de uma instância para serem executados. Ou seja, pode chamar o método através da classe.

Herança vs composição

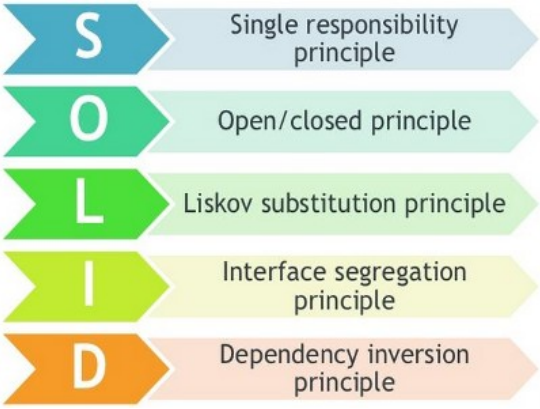
Ambas reaproveitem code.

Herança para **especialização** (para apenas compartilhamento, vai precisar replicar code);
Extende o comportamento para outro objeto.
Exemplo: Uma pessa física é uma pessoa, uma pessoa jurídica é uma pessoa, cada uma especializa a classe Pessoa.

Composição para **compartilhamento de código** (gera um desacoplamento, concedendo uma flexibilidade que nos permite manipular com facilidade).
Delega responsabilidade para outro objeto.
Exemplo: uma classe Pessoa não precisa de todo o que tem no banco Sql, ela usa um pedaço.

Dica geral: *se não souber decidir, vai para composição*, escolha mais provavél.

SOLID



L iskov substitution principle
Em linguagens de tipagem dinâmica, **não precisamos necessariamente utilizar um tipo genérico** para definir seus subtipos. Há uma inferência de tipo, associada com o que chamamos de duck-typing.
// Exemplo: Conta e conta poupança, conta salário, conta corrente. Se der errado, é ou porque a superclass faz alguma coisa a mais, como um método, ou por alguma especialização errada.

I nterface segregation principle
Cada interface deve ter um propósito específico (pois o código cliente pode não necessitar de fazer as duas operações). Ligado com Single responsibility principle.

Dicas diversas

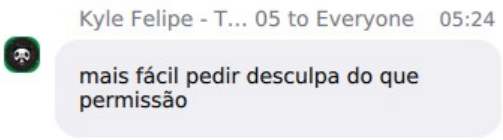
→ **Na dúvida com a herança. *vai para composição***, escolha mais provavél

→ **Métodos mágicos** "__" ou também chamado "dunder" (para double underline):
https://pt.stackoverflow.com/questions/176465/m%C3%A9todos-ou-dunder-em-python-quais-s%C3%A3o-os-mais-utilizados

Há também vários métodos mágicos utilizados para personalizar operações de comparação.

- `__lt__`: Menor que (less than)
- `__le__`: Menor ou igual (less or equal)
- `__eq__`: Igual (equals)
- `__ne__`: Não igual (not equal)
- `__ge__`: Maior ou igual (greather or equal)
- `__gt__`: - Maior que (greather than)

→ Resumindo mentalidade do Python ;)



→ Aula ao vivo era geralmente para entender *self que inicializa instância*, como *passar interface com ou sem dunder*, *@property*

---

3) Padrões de projeto

Padrões de projeto: formas de resolver problemas comuns na OO.

Padrões de projeto

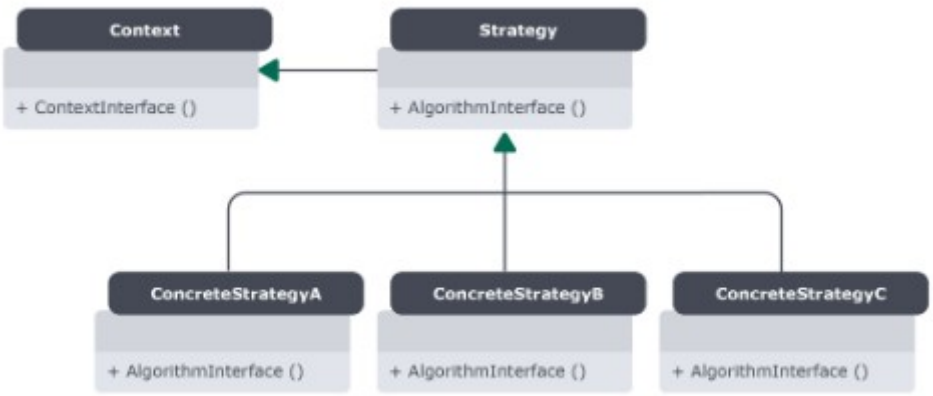
Design patterns: para padronizar resoluções de problemas.
23 no livro inicial, divididos em **3 grupos: criacionais, estruturais, comportamentais**.
No mercado, podem ter mais.

Um padrão é definido e documentado com um *nome, o problema que busca resolver, uma solução dada por ele e as consequências* sobre esta escolha.

→ Iterator
Percorrer uma coleção.
Interface dá jeito de ir para próximo elemento (*next*) e entender quando acabou (*hasNext*).

→ Adapter
Converter a interface de uma classe em outra interface esperada pelo cliente, resolvendo incompatibilidade entre as interfaces.
Introduz um **tradutor**, mantendo o **desacoplamento** entre código do cliente e objeto adaptado.

→ Strategy
Problema: classe de contexto enorme pois com tarefa específica, porém com **implementações diferentes**.
Solucionar com **classe aparte** Strategy: "define uma *família de algoritmos*, *encapsula* cada uma delas e torná-las *intercambiáveis*. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam."



Code Smells

Para detetar **falta de Clean Code**, entender quando **refatorar**.
Linter tem suas regras orientadas para evitar code smells.

Alguns dos classics

→ **long methods, large class** - grandes métodos/classes mais de uma responsabilidade
→ **duplicate code**, que deve atualizar sempre em mais de um lugar;
→ **dead code**, devia tirar pois sempre estará nos commits precedentes;
→ **speculative generality**, quando tentou antecipar e tornou mais complexo;
→ **overcommenting**;
etc.

MiddleMan

Cortar o intermediário.

Data Clumps

Grupo de variáveis frequentemente passado junto como **parâmetro.** Indica que esses grupos devem ser transformados em suas **próprias classes**.

Ver mais exemplos de code smells.

Dicas diversas

→ construtor no python chama init, para criar instância

---

4) Projeto - Relatórios de Estoque

Dicas diversas do projeto

→ **Rodar e testar aos poucos seu code: duas alternativas**

1/ *python3 -i your-path.py*
(para abril um terminal Python do arquivo onde quer testar funcionalidades, ver prints…)
>>> method(parameters)

2/ *python your-path.py*
(para rodar um arquivo python diretamente)
Sabendo que no final do referido arquivo, tem:

```
if __name__ == "__main__":
    # checks if file executed directly or called elsewhere
    print(Class.method(parameters))
```

→ **Counter**
