12

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

12

Introdução a Padrões de Projeto e aos princípios do SOLID

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo veremos uma introdução aos padrões de projeto, fazendo uma breve descrição das situações nas quais eles podem ser aplicados no desenvolvimento de projetos de software em POO. Vamos nos familiarizar com os tipos de padrões de projeto clássicos: criacionais, estruturais e comportamentais. Veremos também uma introdução aos 5 princípios conhecidos por seu acrônimo SOLID, que também nos ajudam a escrever programas mais legíveis e mais fáceis de serem estendidos e mantidos.

11.1. Introdução a Padrões de Projeto

Um padrão de projeto é uma forma reconhecida e consolidada para resolvermos um determinado problema. Vamos pensar em um jogo de xadrez: não existe uma única jogada correta para uma dada situação, e diversas jogadas podem ser vantajosas. Ao longo dos anos, enxadristas de todo o mundo se deparam com diversas jogadas no xadrez, e muitas delas acabam se repetindo, então surgem algumas jogadas "pré-definidas", que ganham nome e ficam famosas por se mostrarem efetivas nas situações para as quais foram pensadas.

Esses conjuntos de jogadas podem ser vistos como um padrão ou uma estratégia para sair de uma dada situação e virar o jogo a seu favor. Assim como no xadrez, os padrões de projeto são estratégias e técnicas usadas para resolver problemas comuns na

programação de softwares e aplicações, por terem se mostrado vantajosas para a solução de tais problemas.SOLID

É importante observar, no entanto, que padrões de projeto não são uma estrutura rígida que nos dita exatamente quais os passos para resolvermos todos os problemas de um mesmo tipo, pois nestas situações, não existe uma única resposta correta, e diferentes abordagens podem ser igualmente boas. Então é preciso, da mesma forma que um enxadrista, adaptar a sua jogada aos movimentos do adversário, isto é, às características e demandas específicas do seu projeto.

É muito comum a confusão entre padrões de projeto e algoritmos, pois ambos descrevem soluções típicas para problemas conhecidos. Um algoritmo irá sempre definir exatamente quais passos seguir para resolver o problema de maneira clara e direta, por outro lado, um padrão de projeto é como uma descrição de alto nível (mais abstrata) da solução, e os detalhes de implementação vão variar de acordo com as características peculiares do problema ao qual é aplicado (SHVETS, A., 2021a).

Conforme avançamos na programação, como comunidade, novos problemas surgem e novas soluções são padronizadas e adotadas de maneira geral, então é natural que diversas dessas soluções acabam sendo incorporadas às linguagens de programação. Portanto, algo que é um padrão de projeto em uma dada linguagem passa a ser apenas uma funcionalidade em outra, o que facilita seu uso.

Isso não significa dizer que aquele padrão de projeto não é mais usado, muito pelo contrário, significa que de tão útil e tão usado, ele agora é parte integrante da linguagem, de modo que estamos utilizando-o o tempo todo para resolver problemas, sem nem pensar a respeito.

Um exemplo clássico é a programação em linguagens de baixo nível, próximas à linguagem de máquina, que em geral não possuem abstrações como classes e funções, e às vezes nem laços ou estruturas de seleção. Então, em tais linguagens, se quisermos escrever uma função vamos precisar usar um “padrão de projeto” para isso. No entanto, funções é algo tão útil, que virtualmente todas as linguagens de programação de nível médio e alto a incorporaram e podemos utilizá-las com comandos extremamente simples, sem nos preocuparmos com o que acontece por baixo dos panos até aquilo virar um código binário que será executado. A função deixou de ser um padrão de projeto para ser uma funcionalidade da linguagem.

Existe um padrão chamado **iterador**, que visa padronizar a forma de percorrer estruturas de dados sem a necessidade de conhecermos seu arranjo interno. Em C#, Java, PHP e outras linguagens, é preciso aplicar um padrão de projeto para construir tais objetos, mas em Python poderíamos argumentar que esse padrão de projeto não é aplicável, pois deixou de ser um padrão e passou a ser parte integrante da sintaxe da linguagem.

Em Python, podemos percorrer com um laço for qualquer objeto iterável, como por exemplo listas, tuplas, *strings*, intervalos (*range*), arquivos, conjuntos, dicionários e muitas outras estruturas, como os objetos gerados pelas funções `map`, `filter`, `zip`, entre outras. Além disso, podemos construir nossos próprios iteradores, bastando para isso criarmos uma classe que defina os métodos `__iter__()` e `__next__()` de acordo com o protocolo definido na documentação (PSF, 2021).

É importante lembrar também que a programação, assim qualquer outro campo da ciência, está em constante evolução e novos problemas surgirão, de modo que serão criados novos padrões de projeto, e padrões antigos poderão deixar de existir, seja por serem incorporados às linguagens ou por que a evolução da tecnologia tornou o problema para o qual existiam obsoletos.

Fazendo uma analogia simplista, se existisse por exemplo um padrão de projeto aplicável a máquinas de fax, esse padrão teria deixado de ser útil, pois máquinas de fax são atualmente peças de museu.

11.1.1. O que é um padrão de projeto

Podemos definir resumidamente um padrão de projeto como uma solução típica para um problema comum em projeto de software (SHVETS, A., 2021a). E ele normalmente é composto por quatro parte:

- **Propósito**: descreve brevemente o problema que ele resolve e a solução proposta;
- **Motivação**: contextualiza o problema e descreve o que é possível alcançar com a solução proposta;
- **Estrutura**: evidencia as classes e como se relacionam, normalmente com diagramas UML; e
- **Exemplo de código**: traz um exemplo de aplicação do padrão, feito comumente em alguma linguagem de programação popular.

Os padrões de projeto clássicos foram desenvolvidos e compilados em um livro (GAMMA, E. et al., 1994) por quatro autores que ficaram conhecidos como “A Gangue dos Quatro”. Neste livro os autores descrevem 22 padrões de projetos separados em três grandes grupos:

- **Padrões criacionais**: fornecem mecanismos para criar (instanciar) objetos de modo a aumentar a flexibilidade e a reutilização de código;
- **Padrões estruturais**: explicam como compor objetos e classes em estruturas maiores, mantendo tais estruturas flexíveis e eficientes;
- **Padrões comportamentais**: cuidam da comunicação eficiente e da distribuição de responsabilidades entre objetos.

Para aplicar um padrão de projeto ao desenvolvimento de um software, precisamos lembrar que não o fazemos a partir da escolha de um padrão que gostamos ou que conhecemos bem, e sim a partir do problema que precisamos resolver. Devemos, a partir de um problema existente, avaliar se há um padrão de projeto que se aplica a ele.

11.1.2. Padrões criacionais

"Os padrões criacionais fornecem vários mecanismos de criação de objetos, que aumentam a flexibilidade e a reutilização de código existente" (SHVETS, A., 2021b). Há neste grupo cinco padrões clássicos:

- **Factory**: providencia uma interface para a criação de objetos de uma classe mãe (superclasse) e permite que classes filhas (subclasses) alterem o tipo de objeto que será criado.
- **Abstract Factory**: permite a criação de famílias de objetos relacionados entre si, sem a necessidade de se especificar as classes concretas.
- **Builder**: permite a construção de objetos passo a passo, levando a construção de diferentes variações e representações de um objeto a partir do mesmo código de construção.
- **Prototype**: permite a cópia de objetos existentes sem deixar o código dependente da implementação de suas classes.
- **Singleton**: garante que uma classe só terá um único objeto instanciado a partir dela, providenciando acesso global a esta instância a todos que precisarem.

11.1.3. Padrões estruturais

"Os padrões estruturais explicam como montar objetos e classes em estruturas maiores, mas ainda mantendo essas estruturas flexíveis e eficientes" (SHVETS, A., 2021c). Os padrões pertencentes a este grupo são:

- **Adapter**: Permite a colaboração de objetos cujas interfaces sejam incompatíveis a priori.
- **Bridge**: Permite a separação de classes complexas e grandes, ou de um conjunto relacionado de classes, em duas hierarquias distintas: abstração e implementação, que podem ser desenvolvidas de maneira independente uma da outra.
- **Composite**: Permite a composição de diversos objetos em estruturas de árvore, que podem então ser vistas e tratadas como um único objeto.
- **Decorator**: Permite que sejam adicionados novos comportamentos a objetos colocando esses objetos dentro de envelopes especiais que contém tais comportamentos.
- **Facade**: Providencia uma interface simplificada para uma biblioteca, módulo, framework ou qualquer outro conjunto complexo de classes.

- **Flyweight**: Permite que sejam salvos mais objetos num mesmo espaço de memória RAM, fazendo um compartilhamento dos estados em comum entre múltiplos objetos, ao invés de fazer com que cada objeto tenha a sua cópia isolada dos dados.
- **Proxy**: Permite que seja fornecido um substituto, ou um objeto temporário, para um outro objeto. Um proxy controla o acesso ao objeto original, de modo que é possível executar instruções, como validações, antes e/ou depois de permitir a interação com o objeto original.

11.1.4. Padrões comportamentais

"Padrões comportamentais são voltados aos algoritmos e a designação de responsabilidades entre objetos" (SHVETS, A., 2021c). Os padrões incluídos neste grupo são:

- **Chain of Responsibility**: Permite que sejam passadas requisições ao longo de uma cadeia de prestadores (objetos responsáveis pela manipulação e execução da requisição). Cada prestador, ao receber um pedido, decide se processa o pedido ou se o encaminha para o próximo prestador na fila.
- **Command**: Transforma uma requisição em um objeto independente, que contém toda a informação necessária a respeito da requisição. Dessa forma, é possível passar tal requisição como argumento para um método ou função, colocá-la em uma fila para ser executada posteriormente, além de permitir a criação de operações que não podem ser desfeitas.
- **Iterator**: Permite que os elementos de um conjunto de dados sejam percorridos sem que tenhamos a necessidade de conhecer a estrutura interna deste conjunto, que pode ser uma lista, pilha, árvore, etc.
- **Mediator**: Permite reduzir um arranjo caótico de dependências entres os objetos ao restringir a comunicação direta entre quaisquer dois objetos e obrigar que toda colaboração/comunicação seja feita através de objeto especial, o mediador, que dá o nome ao padrão.
- **Memento**: Permite que os estados de um objeto sejam salvos, e posteriormente restaurados, sem precisar revelar os detalhes de sua implementação.
- **Observer**: Permite que seja definido um mecanismo de assinatura para notificar múltiplos objetos (todos os assinantes do serviço) quando houver qualquer alteração no objeto que está sendo observado.
- **State**: Permite a um objeto alterar seu comportamento de acordo com mudanças no seu estado interno, fazendo parecer como se o objeto tivesse trocado de classe.
- **Strategy**: Permite a criação de uma família de algoritmos, que podem ser colocados em classes separadas, e cujos objetos são intercambiáveis.

- **Template Method**: Define um esqueleto de um algoritmo em uma superclasse, permitindo que as subclasses sobrescrevam passos específicos desse algoritmo sem modificar sua estrutura.
- **Visitor**: Permite a separação entre os algoritmos e os objetos nos quais eles operam.

11.1.5. Exemplos de aplicação

O site Refactoring Guru, listado como referência para cada grupo de padrões, mantido por Alexander Shvets, autor do livro "Mergulho nos Padrões de Projeto", disponibiliza exemplos práticos da aplicação de cada um dos padrões em diversas linguagens de programação, inclusive Python.

Uma lista completa para todos os exemplos disponíveis em Python pode ser acessada no link: Padrões de Projeto em Python (refactoring.guru).

11.2. Introdução aos Princípios do SOLID

Foram organizados por Robert C. Martin e publicados no ano 2000 (MARTIN, R. C., 2000) em um artigo que lista um conjunto de princípios cujo objetivo é deixar o código e a aplicação mais reutilizáveis, robustos e flexíveis. Os primeiros 5 princípios deste artigo se aplicam especificamente ao projeto de classes em POO e ficaram conhecidos pelo acrônimo SOLID, formado a partir das iniciais em inglês de cada um:

- Princípio da **Responsabilidade Única (Single Responsibility)**:

  "Nunca deve haver mais de uma única razão para que uma classe precise ser alterada";
- Princípio do **Aberto/Fechado (Open/Closed)**:

  "Um módulo deve ser aberto para extensão e fechado para modificação";
- Princípio da **Substituição de Liskov (Liskov Substitution)**:

  "Subclasses devem poder substituir suas classes base";
- Princípio da **Segregação de Interfaces (Interface Segregation)**:

  "Várias interfaces específicas são melhores do que uma única interface de propósito geral";
- Princípio da **Inversão de Dependências (Dependency Inversion)**:

  "Dependa de abstrações. Não dependa de concreções";

Estes cinco princípios respondem a diferentes aspectos de uma mesma questão em comum: como gerenciar as dependências do nosso código ou aplicação? Como escrever e organizar nosso código para que nossa aplicação seja fácil de entender, manter e estender?

No entanto, as definições acima nos dão o objetivo final que queremos alcançar com o nosso código, mas dizem pouco a respeito de como chegar lá ou o que fazer e o que não fazer na prática, ao escrever um trecho de código.

Para isso é preciso estudo e prática contínuos, como disse Robert C. Martin (MARTIN, R. C., 2009), é preciso se aprofundar no estudo de tais princípios na literatura e praticar a análise de código, tanto nosso próprio código quanto o de colegas, em busca de situações em que os princípios foram aplicados ou violados, e entender o motivo em cada caso. Uma boa sugestão é também participar de grupos de estudo e discussão a respeito e praticar a aplicação de um ou mais princípios, sempre avaliando se o resultado é de fato melhor após aplicá-los.

11.2.1. Princípio da responsabilidade única

Este princípio foi introduzido por Robert C. Martin (MARTIN, R. C., 2005), e diz que uma classe ou módulo deve ter apenas uma razão para ser alterada, isto é, todas as funções e classes de um módulo, ou todos os métodos de uma classe devem estar alinhados em torno de um objetivo único de modo que só precisem ser alterados caso haja uma demanda diretamente relacionada a esse objetivo.

Vamos pensar no seguinte exemplo: estamos desenvolvendo um programa e precisamos adicionar a funcionalidade de realizar atualizações automáticas. Para isso precisamos monitorar o lançamento de novas versões da aplicação em um determinado repositório, baixar o pacote da atualização quando este estiver disponível e aplicá-lo.

Cada nível de abstração exigirá um nível de separação de responsabilidades diferente, por exemplo, podemos colocar todas essas funcionalidades em um mesmo módulo, que será responsável por atualizar o programa, ou seja, conseguimos definir uma única responsabilidade para nosso módulo e ele só precisará ser alterado quando houver uma alteração no processo de atualização.

No entanto, ao descermos um nível na abstração e pensarmos nas classes, se tivermos uma única classe responsável por todas as tarefas, estaremos criando uma classe que terá diversas razões para mudar, e ao alterar qualquer uma das partes, precisaremos testar todas as outras para garantir que não quebramos nada no código.

Por exemplo, caso o serviço de verificação de novas atualizações mude, precisaremos editar nossa classe, e como a mesma classe é responsável por também baixar e instalar as atualizações, é perfeitamente possível que nossas alterações acabem afetando os demais processos, então precisaremos testar todas as funcionalidades.

Por outro lado, se separamos tais tarefas em três classes diferentes: uma para verificar novas atualizações, uma para baixar o pacote de atualizações e outra para aplicá-las, podemos alterar completamente cada uma das classes sem nos preocuparmos

com o que acontece nas demais classes, pois isolamos a responsabilidade, ou seja, a razão para ser alterada, de cada uma.

11.2.2. Princípio do Aberto/Fechado

Este princípio foi inicialmente introduzido por Bertrand Meyer, em 1988 (Meyer, B., 1997), e diz que classes, módulos, funções, etc., devem ser abertos à extensão e fechados à modificação.

Podemos aplicar este princípio a classes através da herança, fazendo com que uma classe ganhe novas funcionalidades (métodos e atributos) através da criação de uma subclasse, sem que a classe original ou qualquer um de seus clientes precise ser alterado. Com isso adicionamos novas funcionalidades à nossa aplicação sem precisar alterar o código que dependia da nossa classe inicial.

Outra forma de aplicar este princípio é através da injeção de dependências[1], pois é possível incluir novas funcionalidades à aplicação, sem precisar alterar as classes existentes.

Pense em uma classe *X* que precisa persistir uma determinada informação, contida em um dicionário, na memória. Inicialmente iremos persistir esses dados em um arquivo *.json, mas se criarmos o arquivo internamente, estaremos presos a este comportamento (salvar em um arquivo), e adicionar a possibilidade de salvar em um banco de dados, por exemplo, exigiria modificar a classe *X*.

No entanto, se injetarmos um objeto que tenha um método `salvar`, que recebe um dicionário e salva o seu conteúdo em um arquivo *.json, a nossa classe *X* não precisa mais saber onde a informação será salva, basta a ela chamar o método `salvar` passando o dicionário a ser salvo como parâmetro. No futuro, podemos alterar o objeto injetado por outro, que possua também um método salvar com a mesma assinatura, mas que salve os dados em um banco de dados ou envie-os para uma api, não importa, pois dessa forma conseguimos estender a funcionalidade da nossa classe *X*, sem modificar seu código interno.

Isso é importante pois raramente sabemos de antemão todos os possíveis usos que nossas classes e módulos terão, então quanto mais fácil for estendê-los para acomodar novas funcionalidades, mas sem modificá-los para não quebrar a compatibilidade com os clientes e usos já existentes, mais fácil será a manutenção e expansão da nossa aplicação.

[1] Injeção de dependência é quando recebemos o objeto do qual dependemos por parâmetro ao invés de criar uma instância diretamente dentro da função ou método.

11.2.3. Princípio da Substituição de Liskov

Este princípio foi inicialmente introduzido por Barbara Liskov, em 1988, e detalhado em seu artigo de 1994 (LISKOV, B. H., WING, J. M., 1994), e diz o seguinte:

> Seja $q(x)$ uma propriedade provável sobre um objeto x do tipo T. Então $q(y)$ deve ser também provável para um objeto y do tipo S, sendo S um subtipo de T.

Em outras palavras, dadas duas classes Te S, com S sendo uma subclasse de T, devemos poder usar um objeto de S como substituto para um objeto de T sem que isso cause qualquer efeito perceptível na aplicação.

No entanto, se para essa substituição funcionar, for necessário verificar se o objeto é uma instância da classe mãe ou das classes filhas, antes de se tomar alguma ação (chamar um método ou acessar um atributo), então o projeto de tais classes não segue o princípio de substituição de Liskov.

Por exemplo, pense em uma classe `PassaroAereo`, com duas subclasses `PassaroMecanico` e `PassadoDeCompania`, todo lugar do código que utilize uma instância de `PassaroAereo`, deve poder passar a utilizar uma instância de qualquer uma de suas classes filhas sem que nenhuma alteração precise ser feita no código, pois ambas as classes filhas terão herdado todos os métodos e atributos necessários para se passar pela classe mãe. Se um método, como `voar(partida, chegada)`, é chamado em uma instância da classe mãe, tal chamada deve continuar funcionando com a mesma assinatura em objetos das classes filhas sem que o código da chamada precise ser alterado.

O comportamento realizado pode ser diferente (polimorfismo - a forma como cada subtipo de pássaro voa pode ser diferente), mas é possível pedir para um objeto pássaro voar do ponto A ao ponto B sem se preocupar se tal objeto é uma instância direta de `PassaroAereo` ou de uma de suas subclasses, isto é, sem fazer nenhuma verificação de tipo no código.

11.2.4. Princípio da Segregação de Interfaces

Este princípio não se aplica a linguagens dinamicamente tipadas, como Python ou Ruby, por exemplo, devido à natureza da própria linguagem (METZ, S., 2009). Quando trabalhamos com linguagens estaticamente tipadas, como C++ ou Java, ao interagir com uma classe, estamos interagindo com sua interface, e portanto devemos criar interfaces específicas para cada cliente de modo a minimizar a dependência em tais interfaces e reduzir assim as alterações necessárias no código caso uma interface precise ser alterada.

Ao quebrarmos uma interface genérica em diversas interfaces específicas, cada cliente só precisa saber a respeito dos métodos que lhe são de interesse, e não precisam lidar com métodos que não são de sua responsabilidade. Isso é vantajoso também para que, ao alterar um determinado trecho de código, não seja necessário recompilar toda a aplicação.

Em Python, ao interagir com um objeto, dependemos apenas da assinatura dos métodos que iremos utilizar e o restante do objeto não importa, portanto este princípio é automaticamente seguido devido à própria natureza dinâmica da linguagem.

11.2.5. Princípio da Inversão de Dependências

Este princípio foi introduzido por Robert C. Martin (MARTIN, R. C., 2005), e consiste de duas partes:

I. Módulos[2] de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
II. Abstrações não devem depender dos detalhes de implementação. Os detalhes devem depender da abstração.

É importante ressaltar que Inversão de Dependências e Injeção de Dependências são duas coisas diferentes, mas que trabalham juntas. Podemos pensar na segunda como sendo uma das técnicas empregadas para se atingir a primeira. Ao injetarmos uma dependência em um módulo, estamos invertendo a ordem tradicional de tais dependências. Vamos ilustrar isso com um exemplo em Python.

Pense em um sistema automatizado de uma padaria, poderíamos escrever as seguintes classes, como mostrado na Codificação 12.1.

Codificação 12.1: Classes que violam o princípio da Inversão de Dependências

```
class Pao:
    def assar(self):
        return 'assando pão'

class Forno:
    def assar(self):
        pao = Pao()
        pao.assar()
```

Fonte: do autor, 2021

Neste exemplo, a classe `Forno` depende diretamente da classe `Pao`, e para assar alguma outra coisa neste forno, precisaríamos modificar o código desta classe, violando também o princípio Aberto/Fechado.

[2] O termo "módulo" aqui faz referência a qualquer bloco de código que forme uma unidade lógica, como funções, classes, métodos, arquivos, etc.

Para resolver este problema podemos usar uma interface, mas como esse conceito não existe no Python, podemos usar uma classe simples para fazer o papel de uma interface, como mostra a Codificação 12.2.

Codificação 12.2: Correção das classes de modo a obedecer o princípio da Inversão de Dependências

```
class Massa:
    def assar(self):
        pass

class Pao(Massa):
    def assar(self):
        return 'assando pão'

class Forno:
    def assar(self, massa):  # massa precisa ser do tipo Massa[3]
        massa.assar()
```

Fonte: do autor, 2021

Com isso, invertemos a ordem das dependências, pois agora a classe `Pao` depende de `Massa` e a classe `Forno` também depende de `Massa`, por meio do método `assar` cujo parâmetro `massa` deve ser[4] do tipo `Massa` (injeção de dependência). E agora, o cliente da nossa classe será responsável por saber o que será assado, como mostra a Codificação 12.3.

Codificação 12.3: Utilização da classe Forno com injeção de dependência

```
>>> pao = Pao()
>>> forno = Forno()
>>> forno.assar(pao)
'assando pão'
```

Fonte: do autor, 2021

E para assar outro tipo de massa, basta criar a nova classe, sem que nenhuma alteração precise ser feita na classe `Forno`, como mostra as Codificações 12.4 e 12.5. Ela automaticamente é capaz de assar qualquer[5] coisa que defina um método `assar`.

[3] Aqui seria possível utilizarmos as dicas de tipo do Python, para indicar ao programador qual o tipo esperado de um objeto: `def assar(self, massa: Massa): massa.assar()`
[4] Observe que o Python não irá forçar essa verificação de tipo, nem mesmo se usarmos as dicas de tipo, portanto cabe a nós programadores documentarmos nossas classes e as utilizarmos da maneira correta.
[5] Em Python não é preciso que a nova classe herde de `Massa`, embora seja recomendado para manter a consistência.

Codificação 12.4: Criação de novas classes

```
class Lasanha(Massa):
    def assar(self):
        return 'assando lasanha'

class Bolo(Massa):
    def assar(self):
        return 'assando bolo'
```

Fonte: do autor, 2021

Codificação 12.5: Utilização das novas classes

```
>>> bolo = Bolo()
>>> lasanha = Lasanha()
>>> forno.assar(bolo)
'assando bolo'
>>> forno.assar(lasanha)
'assando lasanha'
```

Fonte: do autor, 2021

Poderíamos também ter criado a classe Massa como abstrata, pois isso deixa explícito seu propósito, tornando o código mais legível e garantindo que os métodos de interesse sejam implementados nas subclasses, como mostra a Codificação 12.6.

Codificação 12.6: Implementação alternativa com criação de uma classe abstrata

```
from abc import ABC, abstractmethod
class Massa(ABC):
    @abstractmethod
    def assar(self):
        pass
```

Fonte: do autor, 2021

Bibliografia

GAMMA, E. et al. **Design patterns**: elements of reusable object-oriented software. Indianapolis: Pearson Education. 1994.

LISKOV, B. H., WING, J. M. **A behavioral notion of subtyping**, 1994. Disponível em: <https://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf>. Acesso em: 21 ago. 2021.

MARTIN, R. C. **Design principles and design patterns**. 2000. Disponível em: <https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>[6]. Acesso em: 15 fev. 2021.

MARTIN, R. C. **Getting a SOLID start.** - Clean Coder, 2009. Disponível em: <https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start>. Acesso em: 21 ago. 2021.

MARTIN, R. C. **The Principles of OOD**. 2005. Disponível em: <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>. Acesso em: 21 ago. 2021.

METZ, S., **GORUCO 2009** - SOLID object-oriented design by sandi metz. 2009. Disponível em: <GORUCO 2009 - SOLID Object-Oriented Design by Sandi Metz - YouTube>. Acesso em: 21 ago. 2021.

Meyer, B. **Object-oriented software construction**, 1997. Prentice Hall: second edition. Disponível em: <https://archive.org/details/objectorientedso00meye_0>. Acesso em: 21 ago. 2021.

PSF. **Glossário:** Iterador. 2021. Disponível em: <https://docs.python.org/3/glossary.html#term-iterator>. Acesso em: 29 abr. 2021.

SHVETS, A. **Refactoring guru:** o que é um padrão de projeto? 2021a. Disponível em: <https://refactoring.guru/pt-br/design-patterns/what-is-pattern>. Acesso em: 29 abr. 2021.

SHVETS, A. **Refactoring guru:** padrões de projeto criacionais. 2021b. Disponível em: <https://refactoring.guru/pt-br/design-patterns/creational-patterns>. Acesso em: 29 abr. 2021.

SHVETS, A. **Refactoring guru:** padrões de projeto estruturais. 2021c. Disponível em: <https://refactoring.guru/pt-br/design-patterns/structural-patterns>. Acesso em: 29 abr. 2021.

SHVETS, A. **Refactoring guru:** padrões de projeto comportamentais. 2021d. Disponível em: <https://refactoring.guru/pt-br/design-patterns/behavioral-patterns>. Acesso em: 29 abr. 2021.

[6] O site original encontra-se fora do ar, portanto foi usado aqui o link da Wayback Machine (archive.org).