PYTHON ACADEMY

PADRÕES DE PROJETO EM PYTHON (DESIGN PATTERNS): FACTORY METHOD

Conheça o padrão de projeto Factory Method em Python, um padrão creacional que resolve o problema de instanciação de objetos.

PYTHONACADEMY.COM.BR

Crie ebooks profissionais incríveis em minutos com IA

Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Python 3.13*** *(Fevereiro/Março 2025) Design Patterns modernos: tipo hints, dataclasses, quando usar e não usar.*

Salve salve Pythonista 👋

Os **Padrões de Projeto** são soluções recorrentes para desafios comuns no desenvolvimento de software.

Em Python, o uso de padrões de projeto como o **Factory Method** pode incrementar a flexibilidade e a escalabilidade das aplicações.

Neste artigo, abordaremos o que é o *Factory Method*, o problema que ele resolve, sua estrutura, aplicabilidade e como implementá-lo na prática com exemplos em Python.

Compreender e aplicar o *Factory Method* é essencial para desenvolver códigos mais limpos e manuteníveis.

O que é o Factory Method e por que utilizá-lo?

O **Factory Method** é um padrão de criação que define uma interface para criar objetos, mas deixa as subclasses decidirem qual classe instanciar.

Ele promove o **princípio de responsabilidade única**, separando o processo de criação do uso dos objetos.

Problema que o Factory Method resolve

Em aplicações complexas, a criação de objetos diretamente pode levar a um acoplamento rígido entre classes.

Isso dificulta a manutenção e a escalabilidade do sistema.

O **Factory Method** resolve esse problema ao centralizar a criação de objetos, permitindo maior flexibilidade e facilitando a extensão do código.

Racional por trás do Factory Method como solução

A ideia central é delegar a responsabilidade de criação de objetos para subclasses específicas.

Dessa forma, o código cliente não precisa conhecer as classes concretas, reduzindo o acoplamento e aumentando a reutilização de código.

Estrutura do padrão de projeto Factory Method

A estrutura do **Factory Method** inclui:

1. **Produto (Product)**: Interface ou classe abstrata para os objetos criados.
2. **Produto Concreto (Concrete Product)**: Implementações específicas do Produto.

3. **Criador (Creator)**: Classe abstrata que declara o método de fábrica.
4. **Criador Concreto (Concrete Creator)**: Subclasses que implementam o método de fábrica para criar instâncias de Produtos Concretos.

Aplicabilidade do padrão de projeto Factory Method

O **Factory Method** é aplicável quando:

- Uma classe não pode antecipar os tipos de objetos que precisa criar.
- A classe deseja delegar a responsabilidade de criação para subclasses.
- É necessário promover o acoplamento fraco entre criador e produtos.

> ***Uma pausa rápida:*** *Se você curte conteúdo bem estruturado como este, vai gostar do* ***Ebookr.ai*** *— uma plataforma que criei para gerar ebooks profissionais com IA sobre qualquer tema. Capa gerada por IA, infográficos automáticos e exportação em PDF. Dá uma olhada!*

Implementando o Factory Method na prática

Vamos implementar o **Factory Method** em Python com um exemplo simples.

Estrutura Básica

```python
from abc import ABC, abstractmethod

# Produto
class Produto(ABC):
    @abstractmethod
    def operacao(self) -> str:
        pass

# Produtos Concretos
class ProdutoConcretoA(Produto):
    def operacao(self) -> str:
        return "Resultado do Produto A"

class ProdutoConcretoB(Produto):
    def operacao(self) -> str:
        return "Resultado do Produto B"

# Criador
class Criador(ABC):
    @abstractmethod
    def factory_method(self) -> Produto:
        pass

    def alguma_operacao(self) -> str:
        produto = self.factory_method()
        resultado = f"Criador: Trabalhando com {produto.operacao()}"
        return resultado

# Criadores Concretos
class CriadorConcretoA(Criador):
    def factory_method(self) -> Produto:
        return ProdutoConcretoA()

class CriadorConcretoB(Criador):
    def factory_method(self) -> Produto:
        return ProdutoConcretoB()
```

Explicação do Código

1. **Importações**: Utilizamos `ABC` e `abstractmethod` da biblioteca `abc` para definir classes abstratas.
2. **Produto**: Classe abstrata que declara o método `operacao`.
3. **Produtos Concretos**: `ProdutoConcretoA` e `ProdutoConcretoB` implementam o método `operacao`.
4. **Criador**: Classe abstrata que declara o método de fábrica `factory_method` e um método concreto `alguma_operacao` que utiliza o produto.
5. **Criadores Concretos**: `CriadorConcretoA` e `CriadorConcretoB` implementam o método de fábrica para retornar instâncias dos produtos concretos.

Exemplos Práticos de Factory Method em Python

Vamos ver como utilizar o **Factory Method** na prática.

Exemplo 1: Logger Configurável

Imagine um sistema que precisa de diferentes tipos de loggers.

```python
from abc import ABC, abstractmethod

# Produto
class Logger(ABC):
    @abstractmethod
    def log(self, mensagem: str) -> None:
        pass

# Produtos Concretos
class ConsoleLogger(Logger):
    def log(self, mensagem: str) -> None:
        print(f"Console: {mensagem}")

class FileLogger(Logger):
    def log(self, mensagem: str) -> None:
        with open("log.txt", "a") as arquivo:
            arquivo.write(f"File: {mensagem}\n")

# Criador
class Aplicacao(ABC):
    @abstractmethod
    def criar_logger(self) -> Logger:
        pass

    def executar(self, mensagem: str) -> None:
        logger = self.criar_logger()
        logger.log(mensagem)

# Criadores Concretos
class AplicacaoConsole(Aplicacao):
    def criar_logger(self) -> Logger:
        return ConsoleLogger()

class AplicacaoFile(Aplicacao):
    def criar_logger(self) -> Logger:
        return FileLogger()
```

Utilização do Exemplo

```python
# Cliente
def main():
    app_console = AplicacaoConsole()
    app_console.executar("Esta é uma mensagem para o console.")

    app_file = AplicacaoFile()
    app_file.executar("Esta é uma mensagem para o arquivo.")

if __name__ == "__main__":
    main()
```

E a saída será:

```
Console: Esta é uma mensagem para o console.
```

```
File: Esta é uma mensagem para o arquivo.
```

Explicação do Exemplo

1. **Logger**: Interface para diferentes tipos de loggers.
2. **ConsoleLogger e FileLogger**: Implementações concretas que logam no console e em um arquivo, respectivamente.
3. **Aplicacao**: Classe abstrata que declara o método de fábrica `criar_logger` e um método `executar` que utiliza o logger.
4. **AplicacaoConsole e AplicacaoFile**: Criadores concretos que retornam instâncias de `ConsoleLogger` e `FileLogger`.
5. **Cliente**: No método `main`, criamos instâncias de aplicações que logam de formas diferentes sem alterar o código que usa os loggers.

Conclusão

O **Factory Method** é um padrão de projeto poderoso que promove a **flexibilidade** e a **manutenibilidade** em aplicações Python.

Ao centralizar a criação de objetos, ele reduz o acoplamento e facilita a extensão do sistema sem modificar o código existente.

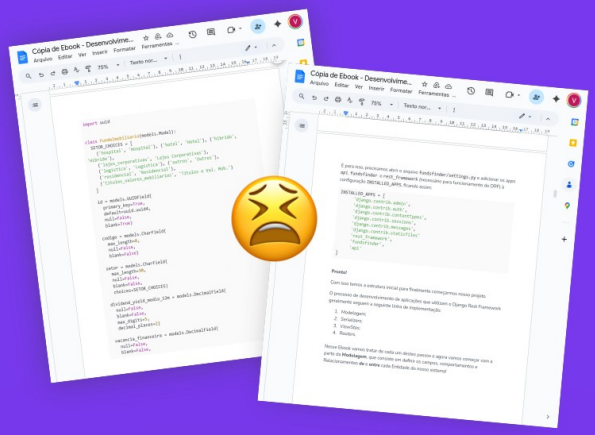
Vimos como implementar e aplicar o *Factory Method* com exemplos práticos, demonstrando sua utilidade em cenários reais.

Incorporar esse padrão em seus projetos Python pode levar a um código mais organizado e escalável.

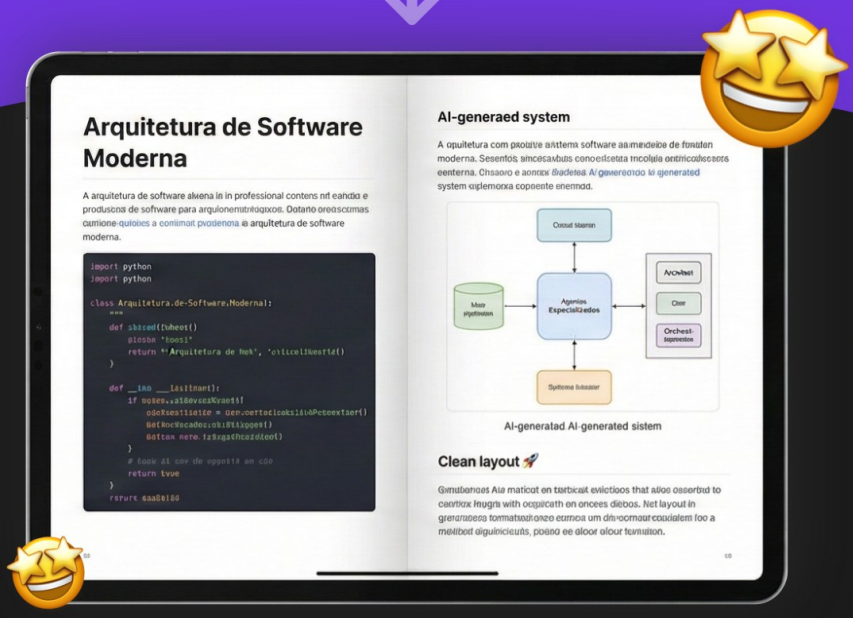
Não se esqueça de conferir!

Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS