



PYTHON
ACADEMY

PADRÕES DE PROJETO EM PYTHON (DESIGN PATTERNS): BUILDER

Conheça o padrão de projeto Builder em Python, um dos Design Patterns mais utilizados no desenvolvimento de software.

[PYTHONACADEMY.COM.BR](https://pythonacademy.com.br)

Este ebook foi gerado por



Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs

Deixe que nossa IA faça o trabalho pesado



Syntax Highlight



Adicione Banners Promocionais



Edite em Markdown em Tempo Real



Infográficos feitos por IA

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 comprovadas para problemas comuns no desenvolvimento de software.

Entre eles, o **Builder** se destaca pela sua capacidade de simplificar a criação de objetos complexos em Python.

Compreender e aplicar o padrão **Builder** pode melhorar a qualidade e a manutenção do seu código, tornando-o mais flexível e escalável.

Neste artigo, exploraremos o intuito, os problemas que ele resolve, sua estrutura, aplicabilidade e como implementá-lo na prática com exemplos em Python.

O que é o Padrão Builder e por que utilizá-lo?

O **Builder** é um **padrão de criação** que facilita a construção de objetos complexos passo a passo.

Ele separa a construção do objeto da sua representação, permitindo criar diferentes representações com o mesmo processo de construção.

Essa separação aumenta a flexibilidade e a clareza do código, especialmente quando os objetos possuem múltiplos atributos ou passos de criação.

Problemas que o Builder resolve

Em projetos onde objetos possuem muitos atributos ou etapas de construção, o código pode se tornar difícil de ler e manter.

Problemas comuns incluem:

- **Construtores longos** com muitos parâmetros.
- **Dificuldade na criação de objetos** com diferentes combinações de atributos.
- **Código duplicado** ao criar objetos similares de maneiras diferentes.

O **Builder** oferece uma solução elegante para esses desafios, promovendo a reutilização e a clareza no processo de criação de objetos.

Racional por trás do Builder como solução

O racional do **Builder** é encapsular o processo de criação de um objeto complexo, promovendo uma interface clara e fluente para a construção.

Ao separar a **lógica de construção** da **representação final** do objeto, o Builder permite criar diferentes variações do objeto sem alterar o código de construção.

Isso resulta em um código mais organizado, fácil de entender e de estender conforme as necessidades do projeto.

Estrutura do Padrão Builder

A estrutura típica do **Builder** inclui os seguintes componentes:

- **Builder**: Interface abstrata que define os métodos para criar as partes do objeto.
- **ConcreteBuilder**: Implementação da interface Builder que constrói e monta as partes específicas do objeto.
- **Product**: O objeto complexo que está sendo construído.
- **Director**: Classe responsável por gerenciar o processo de construção utilizando o Builder.

Essa separação de responsabilidades facilita a criação de diferentes representações do objeto sem alterar a lógica de construção.

Estrutura do Builder

```
+-----+           +-----+
|   Director   |           |   Builder   |
+-----+           +-----+
| - builder    |<-----| + build_part_1() |
| + construct() |           | + build_part_2() |
+-----+           | + get_result()  |
                    +-----+
                      ^
                      |
                    +-----+
                    | ConcreteBuilder |
                    +-----+
                    | - product       |
                    | + build_part_1() |
                    | + build_part_2() |
                    | + get_result()   |
                    +-----+
```


Aplicabilidade do Builder

O padrão **Builder** é aplicável em situações como:

- **Criação de objetos complexos** com múltiplos atributos ou componentes.
- **Construção de diferentes representações** do mesmo objeto.
- **Evitar construtores com muitos parâmetros**, melhorando a legibilidade do código.
- **Facilitar a criação de objetos imutáveis**, onde todos os atributos devem ser definidos no momento da criação.

Implementando o Builder na prática

Vamos implementar o padrão **Builder** em Python utilizando um exemplo clássico de construção de uma **casa**.

Primeiro, definimos o **Produto** que será construído:

```
class Casa:
    def __init__(self):
        self.quartos = 0
        self.banheiros = 0
        self.garagem = False

    def __str__(self):
        return f'Casa com {self.quartos} quartos, {self.banheiros} banheiros e garagem: {self.garagem}'
```

Explicação do código:

1. **Classe Casa:** Representa o objeto complexo a ser construído.

2. **Atributos:** `quartos`, `banheiros` e `garagem` definem as características da casa.
3. **Método `str`:** Retorna uma representação legível da casa.

Definindo o Builder

```
from abc import ABC, abstractmethod

class CasaBuilder(ABC):
    @abstractmethod
    def adicionar_quartos(self, numero):
        pass

    @abstractmethod
    def adicionar_banheiros(self, numero):
        pass

    @abstractmethod
    def adicionar_garagem(self, tem_garagem):
        pass

    @abstractmethod
    def construir(self):
        pass
```

Explicação do código:

1. **Classe `CasaBuilder`:** Interface abstrata que define os métodos para construir as partes da casa.
2. **Métodos Abstratos:** `adicionar_quartos`, `adicionar_banheiros`, `adicionar_garagem` e `construir` devem ser implementados pelas classes concretas.

Implementando o ConcreteBuilder

```
class ConcreteCasaBuilder(CasaBuilder):
    def __init__(self):
        self.casa = Casa()

    def adicionar_quartos(self, numero):
        self.casa.quartos = numero
        return self

    def adicionar_banheiros(self, numero):
        self.casa.banheiros = numero
        return self

    def adicionar_garagem(self, tem_garagem):
        self.casa.garagem = tem_garagem
        return self

    def construir(self):
        return self.casa
```

Explicação do código:

1. **Classe ConcreteCasaBuilder:** Implementa a interface `CasaBuilder`.
2. **Métodos de Construção:** Cada método adiciona uma parte específica à casa.
3. **Método construir:** Retorna o objeto `Casa` construído.

***Falando em Builder:** Assim como o padrão Builder constrói objetos passo a passo, o **DevBook** constrói ebooks técnicos de forma estruturada — usando IA para gerar conteúdo, formatar código e exportar PDFs profissionais. Dá uma conferida!*


Crie Ebooks técnicos incríveis em minutos com IA


Conheça a 1ª IA Especializada na criação de Ebooks **com código!**




Chega de formatar código no Google Docs

Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

Definindo o Director

```
class Diretor:
    def __init__(self, builder: CasaBuilder):
        self.builder = builder

    def construir_casa_simples(self):
        return (self.builder
                .adicionar_quartos(2)
                .adicionar_banheiros(1)
                .adicionar_garagem(False)
                .construir())

    def construir_casa_completa(self):
        return (self.builder
                .adicionar_quartos(4)
                .adicionar_banheiros(3)
                .adicionar_garagem(True)
                .construir())
```

Explicação do código:

1. **Classe Diretor:** Gerencia o processo de construção usando o `CasaBuilder`.
2. **Métodos de Construção:** `construir_casa_simples` e `construir_casa_completa` definem diferentes representações da casa.

Exemplos práticos do Builder em Python

Vamos ver como utilizar o **Builder** na prática para construir diferentes tipos de casas.

```
# Instanciando o builder e o diretor
builder = ConcreteCasaBuilder()
diretor = Diretor(builder)

# Construindo uma casa simples
casa_simples = diretor.construir_casa_simples()
print(casa_simples)

# Construindo uma casa completa
casa_completa = diretor.construir_casa_completa()
print(casa_completa)
```

E a saída será:

```
Casa com 2 quartos, 1 banheiros e garagem: False
Casa com 4 quartos, 3 banheiros e garagem: True
```

Explicação do código:

1. **Instâncias:** Criamos `builder` e `diretor`.
2. **Construção de Casas:** Utilizamos o `diretor` para construir diferentes tipos de casas.
3. **Impressão:** Exibimos as características das casas construídas.

Aplicação em Projetos Reais

Em projetos reais, o **Builder** pode ser utilizado para:

- **Construção de objetos complexos**, como interfaces gráficas ou documentos.
- **Configuração de objetos com múltiplas opções**, como configurações de sistema.

- **Criação de objetos imutáveis**, garantindo que todas as propriedades sejam definidas no momento da construção.

Conclusão

Neste artigo, exploramos o incrível potencial do **Builder** para a criação de objetos complexos em Python.

Vimos como utilizá-lo e implementá-lo em seu projeto, e aprendemos a criar modelos básicos usando `CasaBuilder` e `ConcreteCasaBuilder`.

Além disso, destacamos as capacidades de separar a lógica de construção da representação final, facilitando a criação de diferentes variações do mesmo objeto sem duplicação de código.

Esperamos que este conteúdo tenha ajudado a entender como o **Builder** pode simplificar e tornar mais robusto o gerenciamento de objetos em seus projetos Python.

Não se esqueça de conferir!



DevBook

Crie Ebooks técnicos em minutos com IA


Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



 Syntax Highlight

 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