



ChatGPT



Explore GPTs

Today

Builder Pattern Library System ...

Abstract Factory Pattern Example

Check Python Init Method

Enhanced Abstract Factory Pattern

Movie Ticket Factory Pattern

Design Patterns Refactoring

Using He for Dog

Isnull vs Coalesce SQL

Yesterday

builder design pattern with python code, you can write code from below domain,

Library Management System,
Parking Lot,
Online Shopping System - E-Commerce,
Stack Overflow,
Movie Ticket Booking System,
Design an ATM,
Airline Management System,
Hotel Management System,
Restaurant Management system,
Design Chess,
Design an Online Stock Brokerage System,
Design a Car Rental System,
Design LinkedIn,
Design Cricinfo,
Design Facebook Social Network,
Design payment gateway



Get citation



Let's use the Builder Design Pattern to create a simple Library Management System in Python. The Builder Design Pattern helps in constructing complex objects step by step. It separates the construction of a complex object from its representation so that the same construction process can create different representations.

Here's an example of how we can apply the Builder Design Pattern for a Library Management System:

python

Copy code

```
class Book:
    def __init__(self):
        self.title = None
        self.author = None
        self.isbn = None
        self.publisher = None
        self.year = None

    def __str__(self):
        return f"Title: {self.title}, Author: {self.author}, ISBN: {self.isbn}, Publisher: {self.publisher}, Year: {self.year}"

class BookBuilder:
    def __init__(self):
        self.book = Book()

    def set_title(self, title):
        self.book.title = title
        return self

    def set_author(self, author):
        self.book.author = author
        return self

    def set_isbn(self, isbn):
        self.book.isbn = isbn
        return self

    def set_publisher(self, publisher):
        self.book.publisher = publisher
        return self

    def set_year(self, year):
        self.book.year = year
        return self

    def build(self):
        return self.book

class Library:
    def __init__(self):
        self.books = []
```

```

def add_book(self, book):
    self.books.append(book)

def show_books(self):
    for book in self.books:
        print(book)

# Usage
library = Library()

book1 = (BookBuilder()
        .set_title("The Catcher in the Rye")
        .set_author("J.D. Salinger")
        .set_isbn("978-0-316-76948-0")
        .set_publisher("Little, Brown and Company")
        .set_year(1951)
        .build())

book2 = (BookBuilder()
        .set_title("To Kill a Mockingbird")
        .set_author("Harper Lee")
        .set_isbn("978-0-06-112008-4")
        .set_publisher("J.B. Lippincott & Co.")
        .set_year(1960)
        .build())

library.add_book(book1)
library.add_book(book2)

library.show_books()

```

In this example:

- `Book` is the product class.
- `BookBuilder` is the builder class that helps in constructing the `Book` object.
- `Library` is a class that manages a collection of books.

The builder pattern makes the creation of complex objects more manageable and readable, especially when dealing with many attributes. The `BookBuilder` class provides methods to set the attributes of the `Book` class and a `build` method to return the final `Book` object.

give other more practical example for builder pattern



Certainly! Let's consider a more practical example using the Builder Design Pattern to create a `Computer` class in a Computer Assembly System. This example will demonstrate how to build a complex object with various optional configurations.

```

python
class Computer:
    def __init__(self):
        self.cpu = None
        self.gpu = None
        self.ram = None
        self.storage = None
        self.os = None

    def __str__(self):
        return (f"CPU: {self.cpu}, GPU: {self.gpu}, RAM: {self.ram} GB, "
                f"Storage: {self.storage} GB, OS: {self.os}")

class ComputerBuilder:
    def __init__(self):
        self.computer = Computer()

    def set_cpu(self, cpu):
        self.computer.cpu = cpu
        return self

    def set_gpu(self, gpu):

```

```

        self.computer.gpu = gpu
        return self

    def set_ram(self, ram):
        self.computer.ram = ram
        return self

    def set_storage(self, storage):
        self.computer.storage = storage
        return self

    def set_os(self, os):
        self.computer.os = os
        return self

    def build(self):
        return self.computer

class ComputerStore:
    def __init__(self):
        self.computers = []

    def add_computer(self, computer):
        self.computers.append(computer)

    def show_computers(self):
        for computer in self.computers:
            print(computer)

# Usage
store = ComputerStore()

gaming_pc = (ComputerBuilder()
             .set_cpu("Intel i9")
             .set_gpu("NVIDIA RTX 3080")
             .set_ram(32)
             .set_storage(1000)
             .set_os("Windows 11")
             .build())

workstation = (ComputerBuilder()
              .set_cpu("AMD Ryzen 9")
              .set_gpu("NVIDIA Quadro RTX 4000")
              .set_ram(64)
              .set_storage(2000)
              .set_os("Windows 10 Pro")
              .build())

budget_pc = (ComputerBuilder()
            .set_cpu("Intel i5")
            .set_gpu("Integrated")
            .set_ram(8)
            .set_storage(500)
            .set_os("Linux")
            .build())

store.add_computer(gaming_pc)
store.add_computer(workstation)
store.add_computer(budget_pc)


store.show_computers()

```

In this example:

- `Computer` is the product class.
- `ComputerBuilder` is the builder class that provides methods to configure different components of a `Computer`.
- `ComputerStore` is a class that manages a collection of computers.

This example demonstrates how you can use the Builder Design Pattern to create `Computer` objects with various configurations, making the construction process more readable and manageable. Each configuration method in `ComputerBuilder` returns the builder itself, allowing for method chaining, which makes the code more concise and fluent.

 **Upgrade plan**
Get GPT-4, DALL-E, and more

 Message ChatGPT



ChatGPT can make mistakes. Check important info.

