Ministerul Educației și Cercetării al Republicii MoldovaUniversitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Creational Design Patterns

Laborator work 1

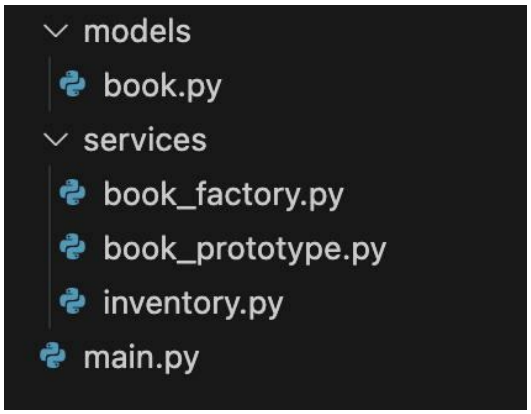
Elaborated:
st. gr. FAF-222 Dimbitchi Sergiu

Chişinău - 2023

Scopul lucrări :

1. Study and understand the Creational Design Patterns.
2. Choose a domain, define its main classes/models/entities and choose the appropriate instantiation mechanisms.
3. Use some creational design patterns for object instantiation in a sample project.

Explicația exercițiului :



book.py

models > book.py > ...

```python
# models/book.py
from abc import ABC, abstractmethod


# Abstract Product
class Book(ABC):
    def __init__(self, title: str, author: str, price: float):
        self.title = title
        self.author = author
        self.price = price

    @abstractmethod
    def get_format(self):
        pass


# Concrete Product 1 - Physical Book
class PhysicalBook(Book):
    def get_format(self):
        return "Physical Copy"


# Concrete Product 2 - EBook
class EBook(Book):
    def get_format(self):
        return "E-Book"

```

The Book class is an abstract class (defined using Python's ABC module). It serves as a blueprint for different types of books. The PhysicalBook and EBook classes are concrete products, which means they provide specific implementations of the abstract Book class.

book_factory.py

~/Desktop/TMPS/lab_1/services/book_factory.py

```python
# services/book_factory.py
from models.book import PhysicalBook, EBook


class BookFactory:
    @staticmethod
    def create_book(book_type: str, title: str, author: str, price: float):
        if book_type == "physical":
            return PhysicalBook(title, author, price)
        elif book_type == "ebook":
            return EBook(title, author, price)
        else:
            raise ValueError("Unknown book type")

```

Design Pattern:

- Factory Method: This pattern provides an interface for creating objects, but allows subclasses or helper classes to determine which class to instantiate. It decouples object creation from the main class, making it easier to extend.

How It Works:

- The BookFactory class has a static method create_book that acts as the factory. Based on the book_type parameter, it either creates a PhysicalBook or an EBook.

- The client doesn't need to know how to instantiate these objects directly, they just provide the parameters and the factory takes care of object creation.

- This helps in simplifying object creation logic and promotes the Open-Closed Principle, as new book types can be added without modifying existing code.

book_prototype.py

services > book_prototype.py > ...

```python
# services/book_prototype.py
import copy
from models.book import Book


class BookPrototype:
    def __init__(self, book: Book):
        self.book = book

    def clone(self):
        return copy.deepcopy(self.book)

```

Design Pattern:

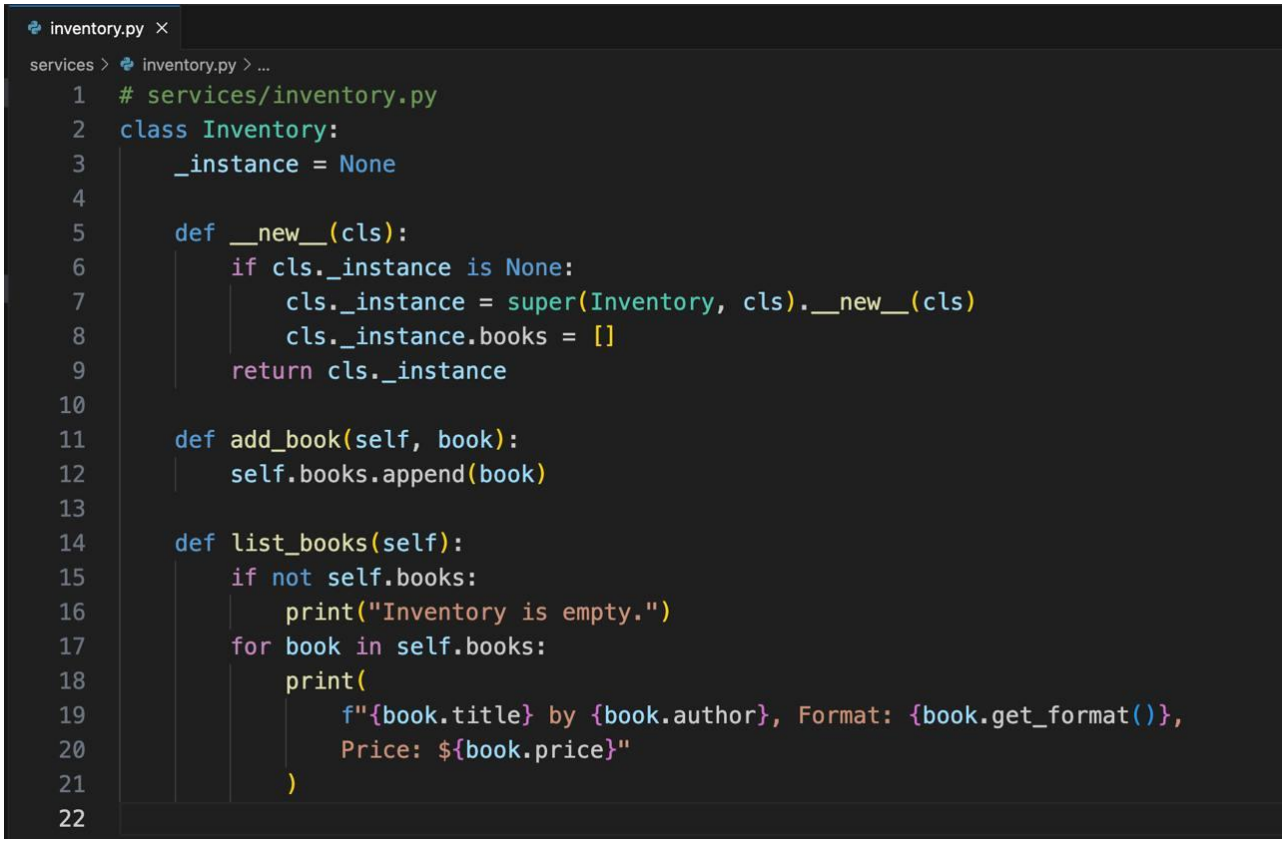
- Prototype: This pattern is used to create objects by copying (cloning) an existing instance (prototype). Instead of instantiating a new object, a clone of the prototype object is made.

How It Works:

- The BookPrototype class holds a reference to an existing Book object (or any of its subclasses).

- The clone() method uses copy.deepcopy() to create a new, independent copy of the original Book instance. This ensures that any modifications to the clone do not affect the original object.

- This pattern is useful when object creation is expensive or complex, and it's easier to clone an existing object than to create a new one.



Design Pattern:

- Singleton: This pattern ensures that a class has only one instance, and provides a global access point to this instance. It's often used for managing shared resources, like databases or configuration settings.

How It Works:

- The Inventory class overrides the __new__ method to control instance creation.

- If the _instance is None, a new instance of Inventory is created and assigned to _instance. On subsequent calls, the existing _instance is returned, ensuring only one instance of Inventory exists.

- The books list is tied to this single instance, so it's shared across all uses of the Inventory class. This means any changes (e.g., adding books) are reflected globally.

- The Singleton pattern ensures there is a single source of truth for the Inventory

main.py

```python
# main.py
from services.book_factory import BookFactory
from services.inventory import Inventory
from services.book_prototype import BookPrototype

if __name__ == "__main__":
    # Create books using Factory Method
    book1 = BookFactory.create_book("physical", "1984", "George Orwell", 12.99)
    book2 = BookFactory.create_book("ebook", "Brave New World", "Aldous Huxley", 7.99)

    # Access the Singleton Inventory
    inventory = Inventory()
    inventory.add_book(book1)
    inventory.add_book(book2)

    # List the books in inventory
    print("\n--- Initial Inventory ---")
    inventory.list_books()

    # Clone an existing book using Prototype pattern
    book_prototype = BookPrototype(book1)
    book_clone = book_prototype.clone()
    inventory.add_book(book_clone)

    # List books again to include the clone
    print("\n--- Inventory After Cloning ---")
    inventory.list_books()

```

In main.py, two books (book1 and book2) are created using BookFactory and added to a shared Inventory instance. The initial inventory is listed, and then book1 is cloned using BookPrototype, with the clone added to the inventory. Finally, the updated inventory, including the clone, is listed again.