Factory Method

Objective

Define an interface for the creation of an object, allowing subclasses to decide which class to instantiate; that is, the Factory Method design pattern allows a class to assume the instantiation of a subclass.

Function

Centralize in a construction class the creation of objects of a subtype of a given type.

Structure

The structure that meets this pattern is shown in Figure 1

- Framework: It represents in an abstract way the object that we want to create, through this interface we define the structure that the created object will have.

- Application (one, two): It represents a concrete implementation of the Framework interface, which is created through the Product (one, two).

- Product: This component can be optional, however, it is recommended to create a product (AbstractFactory) that defines the default behavior of the Products (one, two).

- Product(one,two): It represents a concrete factory which is used to create the ConcreteProduct, this class inherits the basic behavior of the product(AbstractFactory).

The structure that meets this pattern is shown in Figure 1

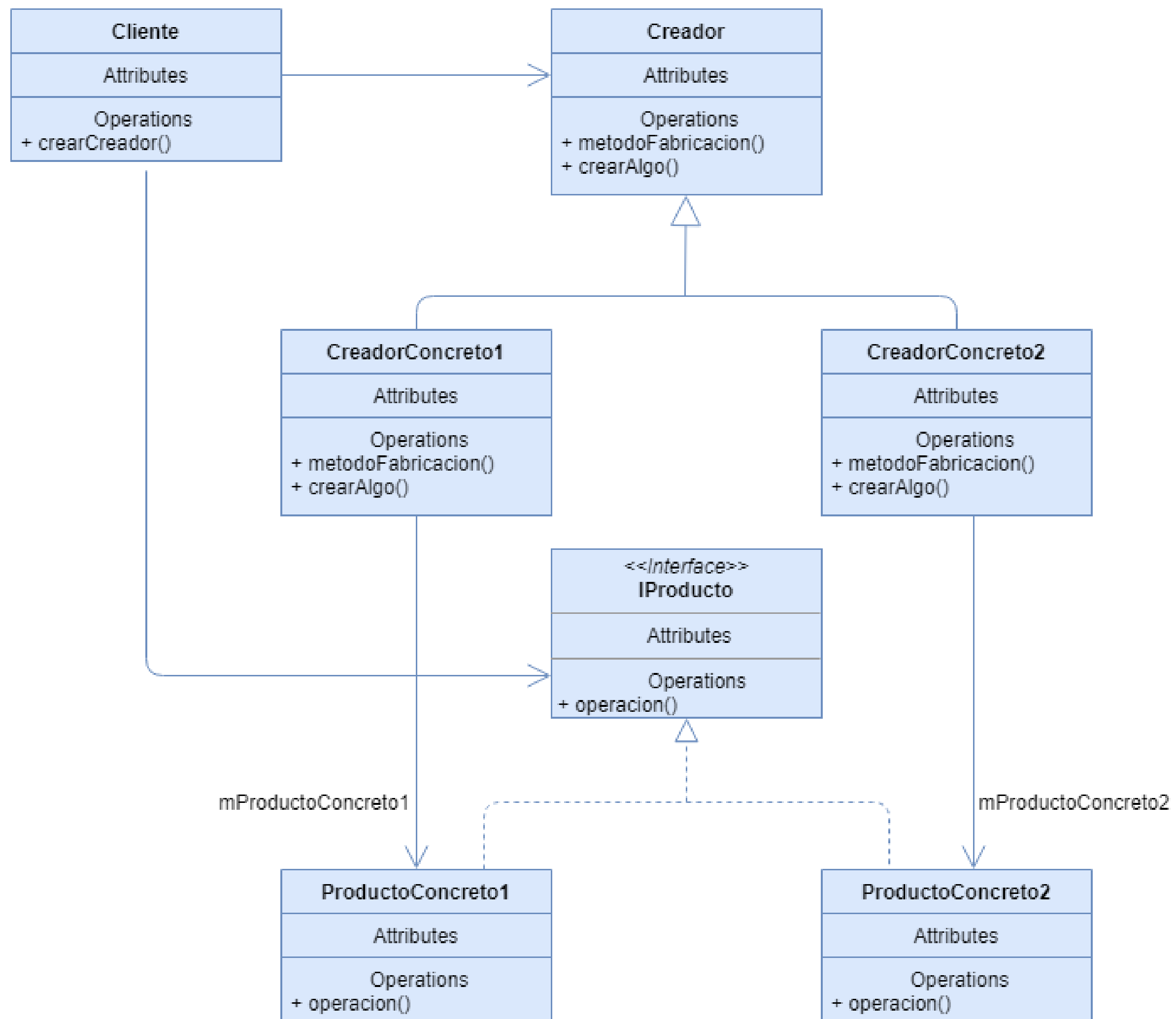

Figure 1: UML Diagram Factory Method Pattern

Applications

- The flexibility of the system is important.
- A class is unable to anticipate which objects it should implement.
- Several objects need to be handled in a similar way, but the results of using them are different.

Design Patterns Collaborators

- A Factory Method pattern generally implements the Abstract Factory pattern.

- They are usually called within a Template Method pattern.

Scope of action

Applied at the object level.

Problem

To allow a class to instantiate a subclass, the names of the objects to be created must be clearly specified depending on the state of the application; this increases the complexity and makes it difficult to maintain the application.

Solution

The Factory Method design pattern allows separating the logic of object creation from the client class; in such a way that the parameters sent by such class are the ones that define the object; that is, it is possible to create objects with different functionality through the same interface.

Diagram or Implementation

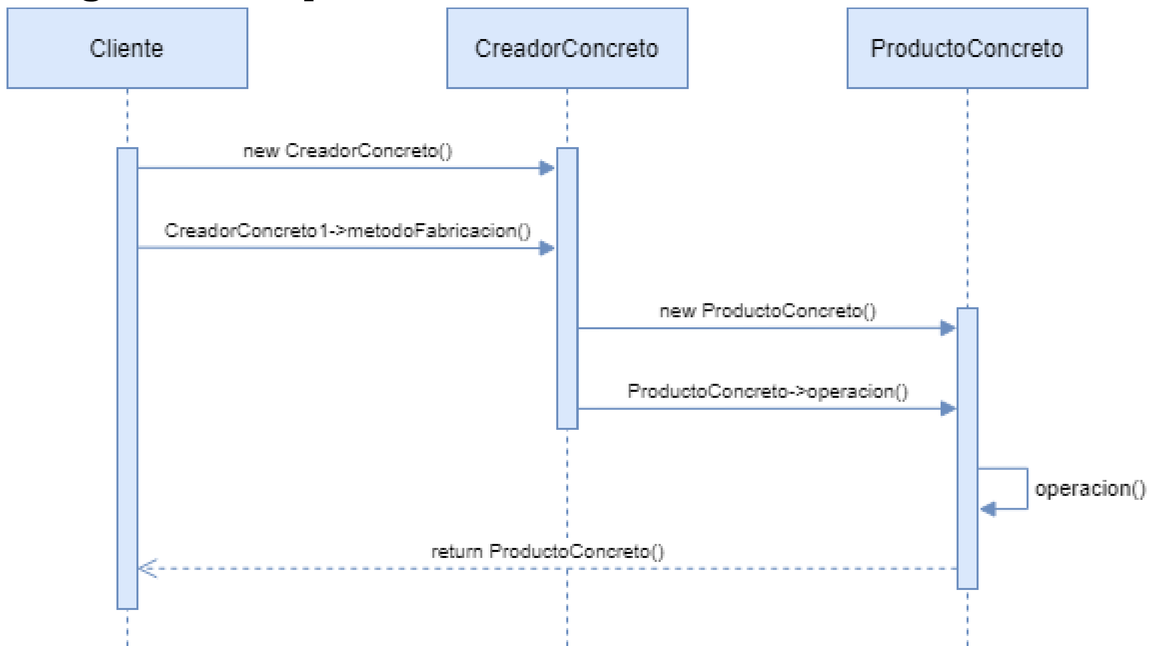

Figure 2: UML Diagram Factory Method Pattern

Figure 2 explains the behaviour of the pattern by means of a sequence diagram.

- Client class asks the ConcreteFactory class to create the ProductA .

- ConcreteFactory locates the concrete implementation of ProductoA and creates a new instance.

- The ConcreteFactory returns ProductA (ConcreteProduct) created.

- The client asks the ConcreteFactory to create ProductB.

- The concreteFactory locates the concrete implementation of ProductB and creates a new instance.

- concreteFactory returns to createProductB (ConcreteProduct) created.

Case Studies

Flight and Hotel Reservation System

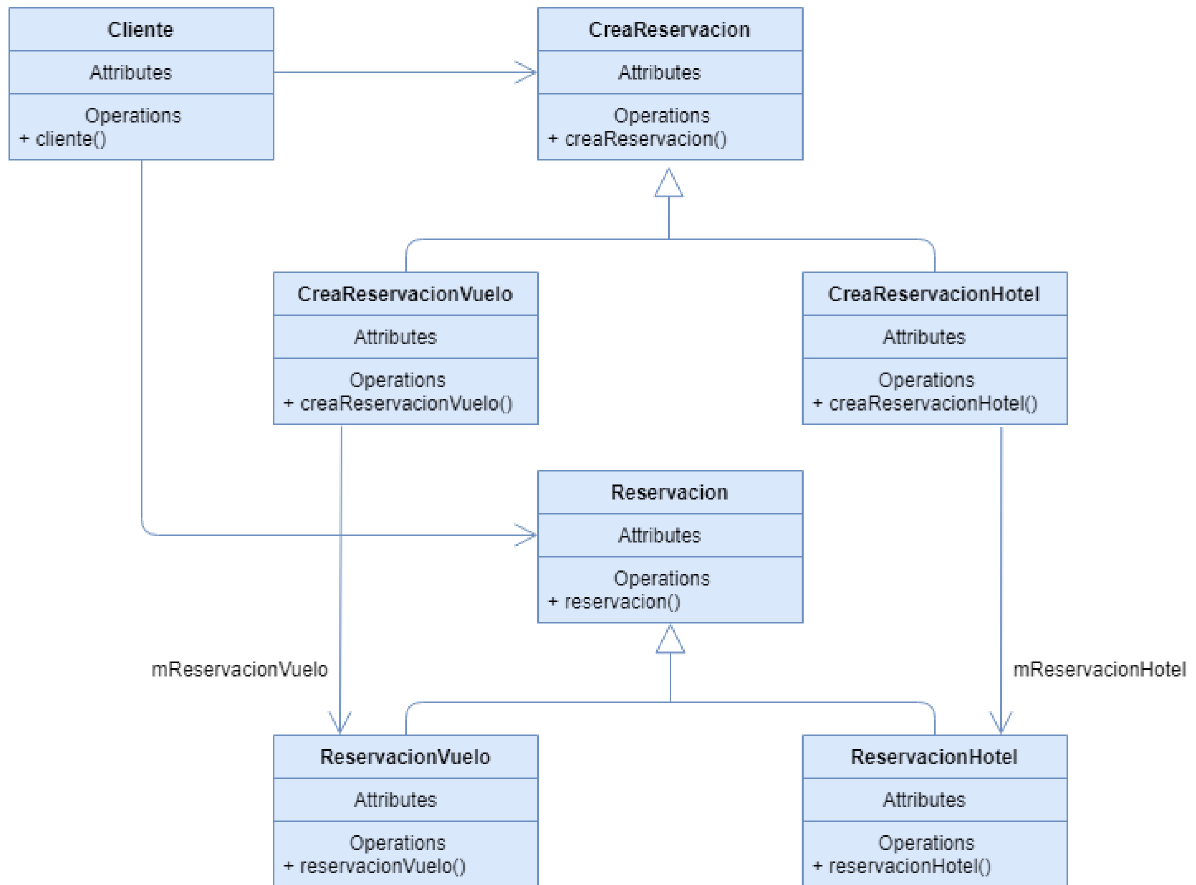

Figure 3: UML Diagram Flight and Hotel Reservation System

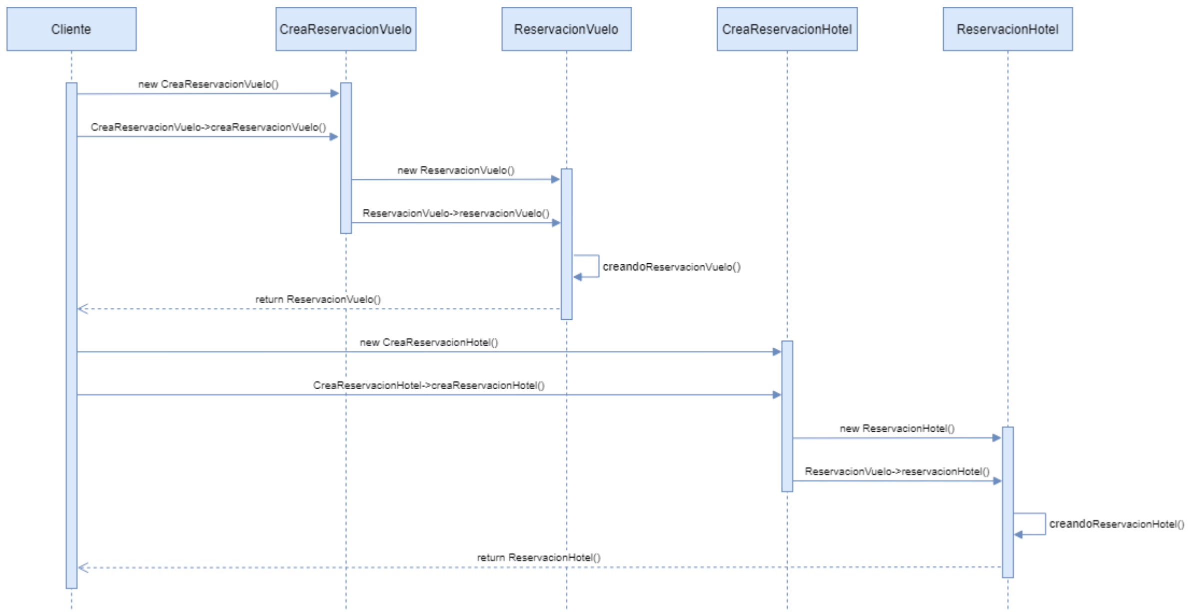

Figure 4: UML Diagram Flight and Hotel Reservation System

Reservation System

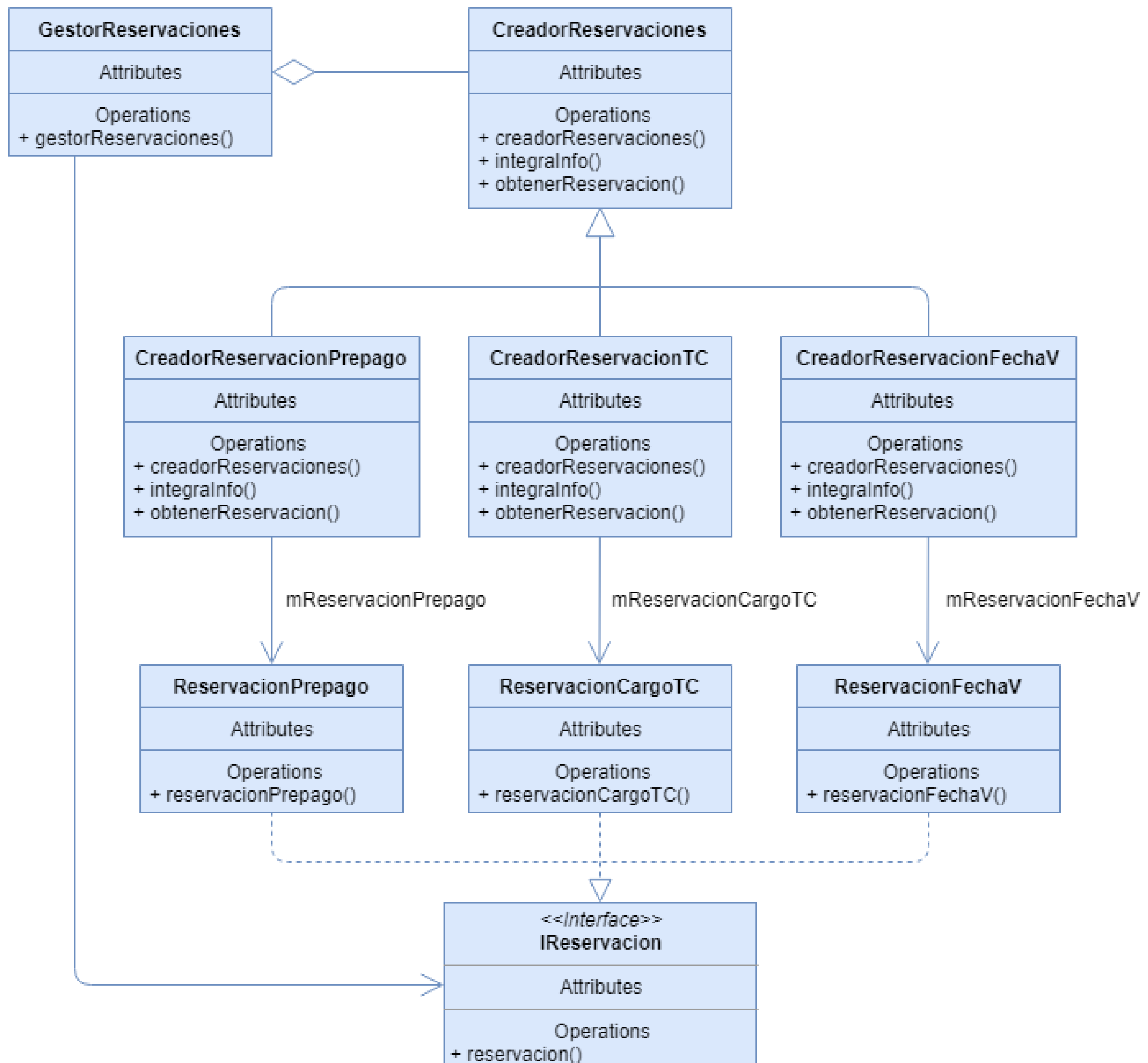

Figure 5: UML Diagram Reservation System

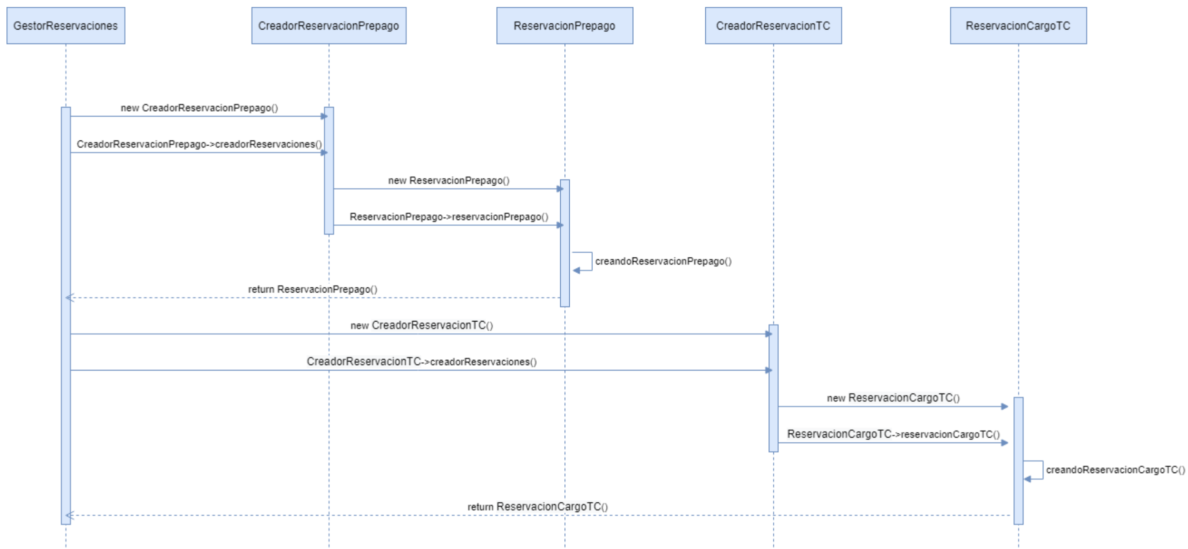

Figure 6: UML Diagram Reservation System

Implementation in Python

```
from __future__ import annotations
from abc import ABC, abstractmethod


class Creator(ABC):
    """
    The Creator class declares the factory method that is supposed to return an
    object of a Product class. The Creator's subclasses usually provide the
    implementation of this method.
    """

    @abstractmethod
    def factory_method(self):
        """
        Note that the Creator may also provide some default implementation of
        the factory method.
        """
        pass

    def some_operation(self) -> str:
        """
        Also note that, despite its name, the Creator's primary responsibility
        is not creating products. Usually, it contains some core business logic
        that relies on Product objects, returned by the factory method.
        Subclasses can indirectly change that business logic by overriding the
        factory method and returning a different type of product from it.
        """
```

```
        # Call the factory method to create a Product object.
        product = self.factory_method()

        # Now, use the product.
        result = f"Creator: The same creator's code has just worked with
{product.operation()}"

        return result


"""
Concrete Creators override the factory method in order to change the resulting
product's type.
"""


class ConcreteCreator1(Creator):
    """
    Note that the signature of the method still uses the abstract product type,
    even though the concrete product is actually returned from the method. This
    way the Creator can stay independent of concrete product classes.
    """

    def factory_method(self) -> ConcreteProduct1:
        return ConcreteProduct1()


class ConcreteCreator2(Creator):
    def factory_method(self) -> ConcreteProduct2:
        return ConcreteProduct2()


class Product(ABC):
    """
    The Product interface declares the operations that all concrete products
    must implement.
    """

    @abstractmethod
    def operation(self) -> str:
        pass


"""
Concrete Products provide various implementations of the Product interface.
"""


class ConcreteProduct1(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct1}"


class ConcreteProduct2(Product):
```

```python
    def operation(self) -> str:
        return "{Result of the ConcreteProduct2}"


def client_code(creator: Creator) -> None:
    """
    The client code works with an instance of a concrete creator, albeit through
    its base interface. As long as the client keeps working with the creator via
    the base interface, you can pass it any creator's subclass.
    """

    print(f"Client: I'm not aware of the creator's class, but it still works.\n"
          f"{creator.some_operation()}", end="")


if __name__ == "__main__":
    print("App: Launched with the ConcreteCreator1.")
    client_code(ConcreteCreator1())
    print("\n")

    print("App: Launched with the ConcreteCreator2.")
    client_code(ConcreteCreator2())
```