Builder

Objective

Separate the construction of a complex object from its representation, so that different representations of that object can be created using the same construction process.

Function

To facilitate the abstraction of the process of creation of a complex object, centralizing it in a single point.

Structure

As shown in figure 1

The reader encapsulates the analysis of the common input. The hierarchy of constructors makes possible the polymorphic creation of many peculiar representations or objectives.

The structure that meets this pattern is shown in Figure 1

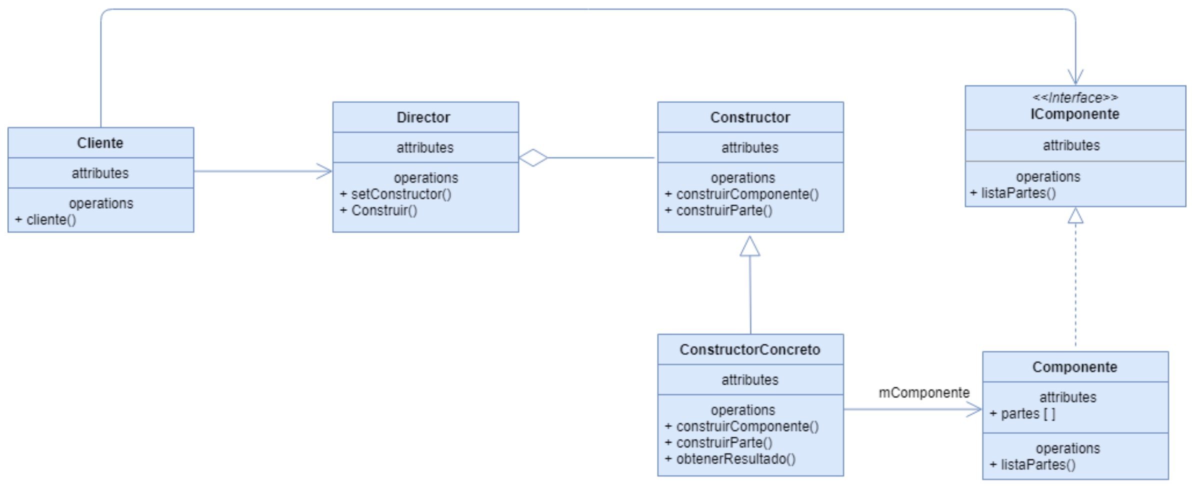

Figure 1: UML Diagram Builder Pattern

Applications

- You need to have careful control over the process of creating an object.
- The algorithm for creating complex objects must be independent of the parts that make up the object and how they are assembled.
- The construction process must allow for different representations of the object being built.

Design Patterns Collaborators

- The object built using a Builder pattern is usually a Composite object.

Scope of action

Applied at the object level.

Problem

To use the same construction process for different objects, the algorithm or construction strategy must be modified; this increases the complexity and reduces the modularity of the system.

Solution

The Builder design pattern allows you to place the construction logic of an object outside the object and program it into an independent class (Builder), which will return to the client a given representation of the object without the client or the client class needing to know the sequence in which the object was created.

Diagram or Implementation

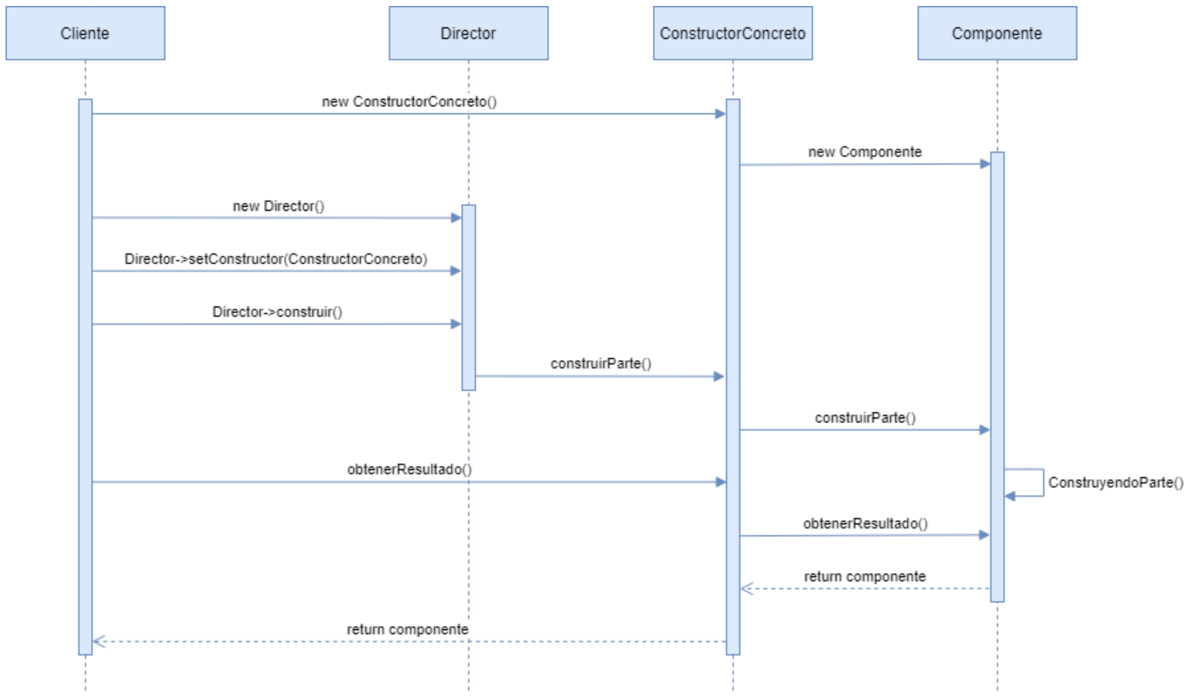

Figure 2: UML Diagram Builder Pattern

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

- Client class creates an instance of the ObjectBuilder .

- The client class executes step 1 of the creation in the ObjectBuilder .

- Internally the ObjectBuilder creates the TargetObject.

- Client class executes step 2 of the creation in the TargetObject.

- Internally the ObjectBuilder creates an instance of OtherObjectA.

- The client executes step 3 of the creation in the TargetObject.

- Internally the ObjectBuilder creates the OtherObjectB instance.

- Client class asks the ObjectBuilder class to create the TargetObject, it takes all the previously created objects, associates them to the TargetObject and returns it.

Study Cases

Restaurant Command System

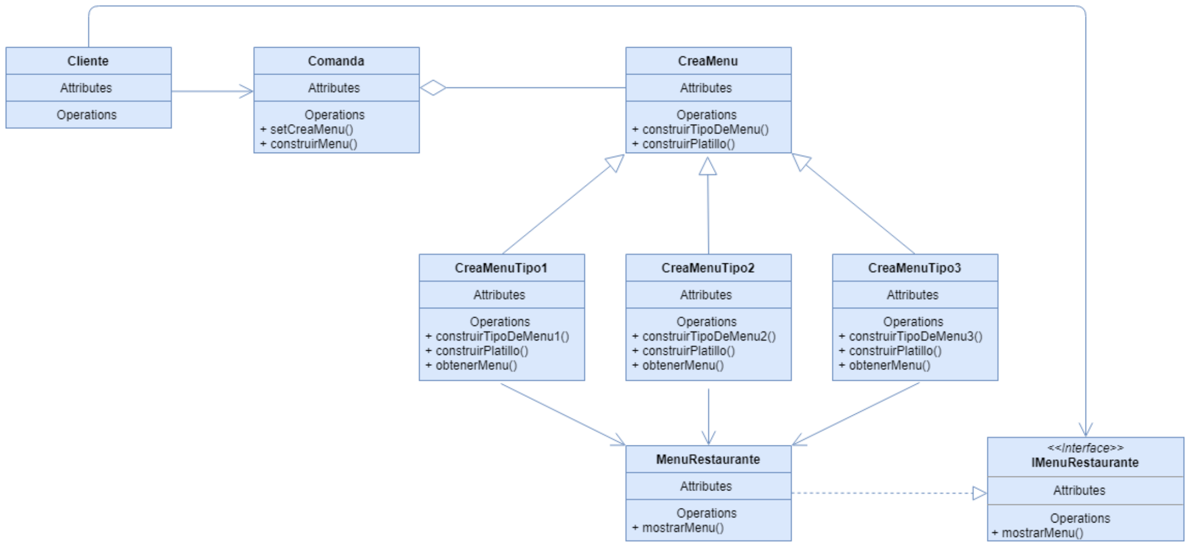

Figure 3: UML Diagram Restaurant Command System

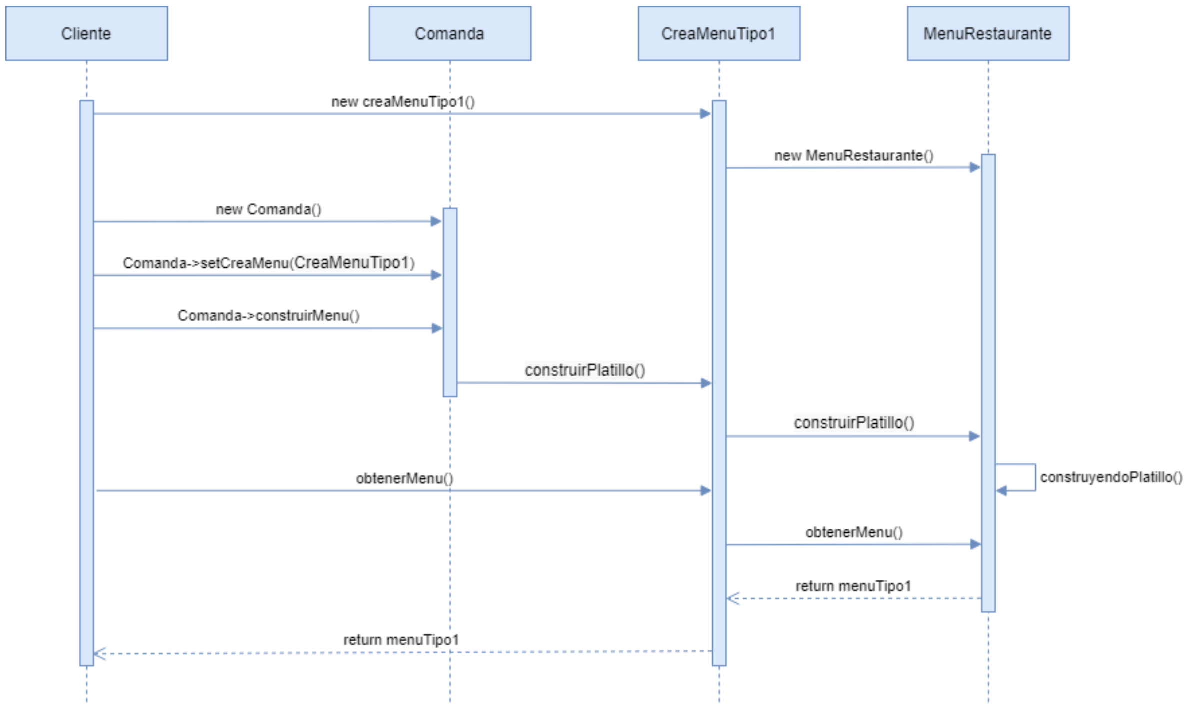

Figure 4: UML Diagram Restaurant Command System

Flight Reservation System

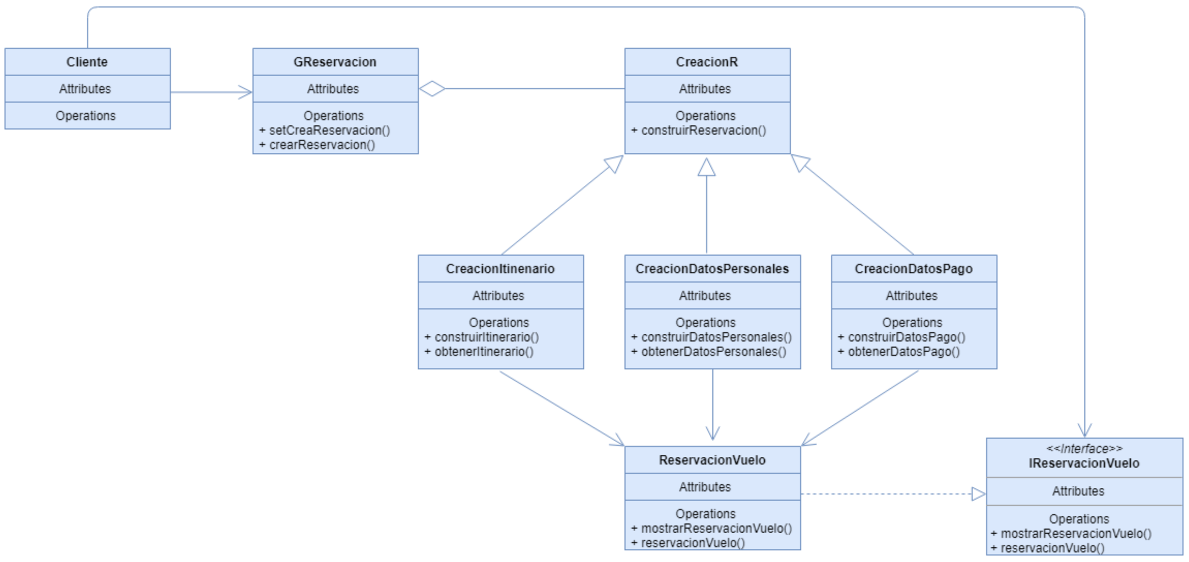

Figure 5: UML Diagram Flight Reservation System

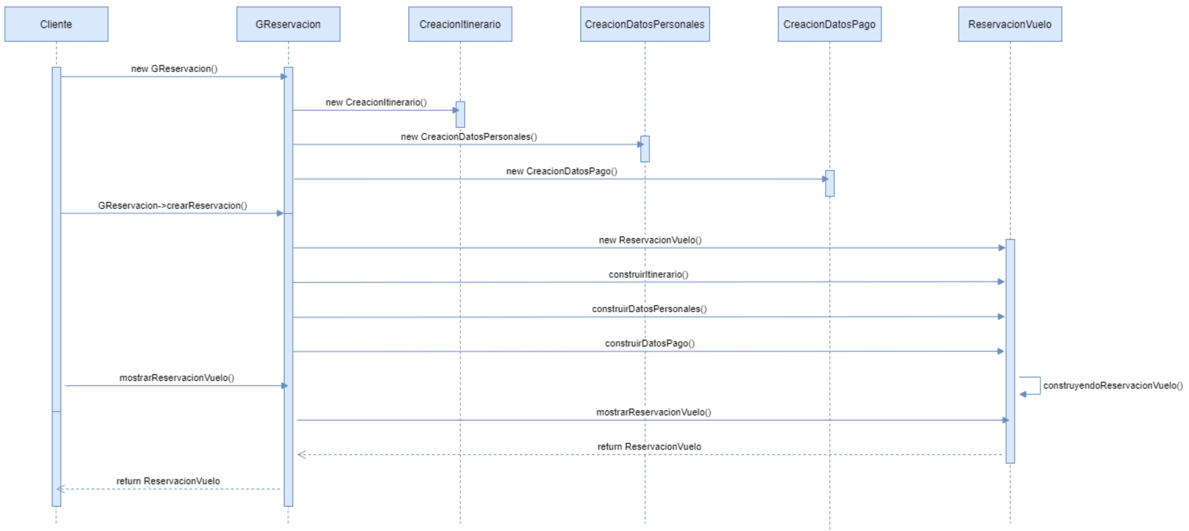

Figure 6: UML Diagram Flight Reservation System

Implementation in Python

```python
from __future__ import annotations
from abc import ABC, abstractmethod, abstractproperty
from typing import Any


class Builder(ABC):
    """
    The Builder interface specifies methods for creating the different parts of
    the Product objects.
    """

    @abstractproperty
    def product(self) -> None:
        pass

    @abstractmethod
    def produce_part_a(self) -> None:
        pass

    @abstractmethod
    def produce_part_b(self) -> None:
        pass

    @abstractmethod
    def produce_part_c(self) -> None:
        pass


class ConcreteBuilder1(Builder):
    """
    The Concrete Builder classes follow the Builder interface and provide
    specific implementations of the building steps. Your program may have
    several variations of Builders, implemented differently.
    """

    def __init__(self) -> None:
        """
        A fresh builder instance should contain a blank product object, which is
        used in further assembly.
        """
        self.reset()

    def reset(self) -> None:
        self._product = Product1()

    @property
    def product(self) -> Product1:
        """
        Concrete Builders are supposed to provide their own methods for
        retrieving results. That's because various types of builders may create
        entirely different products that don't follow the same interface.
        Therefore, such methods cannot be declared in the base Builder interface
        (at least in a statically typed programming language).

        Usually, after returning the end result to the client, a builder
        instance is expected to be ready to start producing another product.
        That's why it's a usual practice to call the reset method at the end of
        the `getProduct` method body. However, this behavior is not mandatory,
        and you can make your builders wait for an explicit reset call from the
        client code before disposing of the previous result.
        """
        product = self._product
        self.reset()
        return product

    def produce_part_a(self) -> None:
        self._product.add("PartA1")

    def produce_part_b(self) -> None:
        self._product.add("PartB1")

    def produce_part_c(self) -> None:
        self._product.add("PartC1")


class Product1():
    """
    It makes sense to use the Builder pattern only when your products are quite
    complex and require extensive configuration.

    Unlike in other creational patterns, different concrete builders can produce
    unrelated products. In other words, results of various builders may not
    always follow the same interface.
    """

    def __init__(self) -> None:
        self.parts = []

    def add(self, part: Any) -> None:
        self.parts.append(part)

    def list_parts(self) -> None:
        print(f"Product parts: {', '.join(self.parts)}", end="")


class Director:
    """
    The Director is only responsible for executing the building steps in a
    particular sequence. It is helpful when producing products according to a
    specific order or configuration. Strictly speaking, the Director class is
    optional, since the client can control builders directly.
    """

    def __init__(self) -> None:
        self._builder = None
    @property
    def builder(self) -> Builder:
        return self._builder

    @builder.setter
    def builder(self, builder: Builder) -> None:
        """
        The Director works with any builder instance that the client code passes
        to it. This way, the client code may alter the final type of the newly
        assembled product.
        """
        self._builder = builder

    """
    The Director can construct several product variations using the same
    building steps.
    """

    def build_minimal_viable_product(self) -> None:
        self.builder.produce_part_a()

    def build_full_featured_product(self) -> None:
        self.builder.produce_part_a()
        self.builder.produce_part_b()
        self.builder.produce_part_c()


if __name__ == "__main__":
    """
    The client code creates a builder object, passes it to the director and then
    initiates the construction process. The end result is retrieved from the
    builder object.
    """

    director = Director()
    builder = ConcreteBuilder1()
    director.builder = builder

    print("Standard basic product: ")
    director.build_minimal_viable_product()
    builder.product.list_parts()

    print("\n")

    print("Standard full featured product: ")
    director.build_full_featured_product()
    builder.product.list_parts()

    print("\n")

    # Remember, the Builder pattern can be used without a Director class.
    print("Custom product: ")
    builder.produce_part_a()
    builder.produce_part_b()
    builder.product.list_parts()
```