FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- **E’ vietata** la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- **E’ inoltre vietata** la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente
dall'autore o da Unica

CREATIONAL PATTERNS

Creational design patterns are design patterns that deal with object creation mechanisms, creating objects in a way that depends on the situation.

The basic form of object creation could result in design problems or in **added complexity to the design**. Creational design patterns solve this problem by controlling the object creation.

Creational design patterns are based on two ideas:
- encapsulate knowledge about which classes are used to create the object
- hide how instances of these classes are created and combined.

Example

Let us consider a CAR class that allows instantiating objects composed of the following parts:
(see code `creational/car`)

- 4 wheels - attribute: size
- engine - attribute: horsepower
- body - attribute: a string,
  i.e. 'SUV', 'Hatchback' (berlina), etc

Create two type of cars

Type	**Body**	**HP**	**Tire size**
Jeep	SUV	400	22
Nissan	Hatchback	100	16

We can build directly the objects that compose the car, instantiate the car and add elements

```
# `creational/car_01`
jeep = Car()
body = Body('SUV')
jeep.set_body(body)
engine = Engine(400)
jeep.set_engine(engine)

for _ in range(4):
    wheel = Wheel(22)
    jeep.attach_wheel(wheel)
```

We can build directly the objects that compose the car, instantiate the car and add elements

```
# `creational/car_01`
jeep = Car()
body = Body('SUV')
jeep.set_body(body)
engine = Engine(400)
jeep.set_engine(engine)

for _ in range(4):
    wheel = Wheel(22)
    jeep.attach_wheel(wheel)
```

Alternatively, we can delegate the responsibility to create objects to the Car `__init__()` method

```
jeep = Car(body='SUV', horsepower=400, tire=22 )
```

The two approaches are equivalent, and their code is confusing and hard to use. It does not allow you to easily identify standard templates for various car models.

Other options:

- define a **class hierarchy that represents the various car models**.

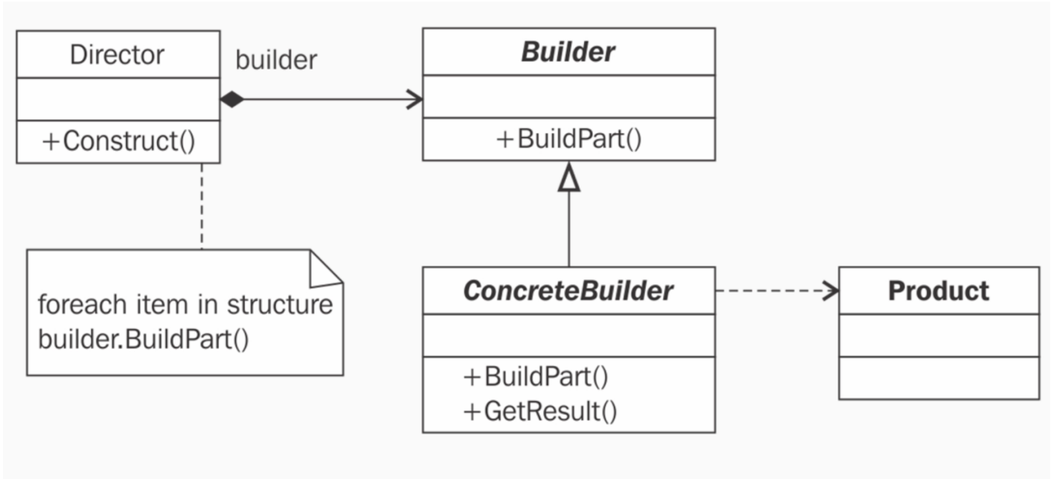
```
class Car:
    #...

class Jeep(Car):
    # Default parameter for the class:
    # body='SUV'
    # horsepower=400
    # tire=22
    #...
```

However, this hierarchy **does not represent different objects**, but a different set of default parameters used to instantiate objects of the same class.

Builder Design Pattern

Builder pattern separates the construction of a complex object from its representation so that **the same construction process can create different representations**.

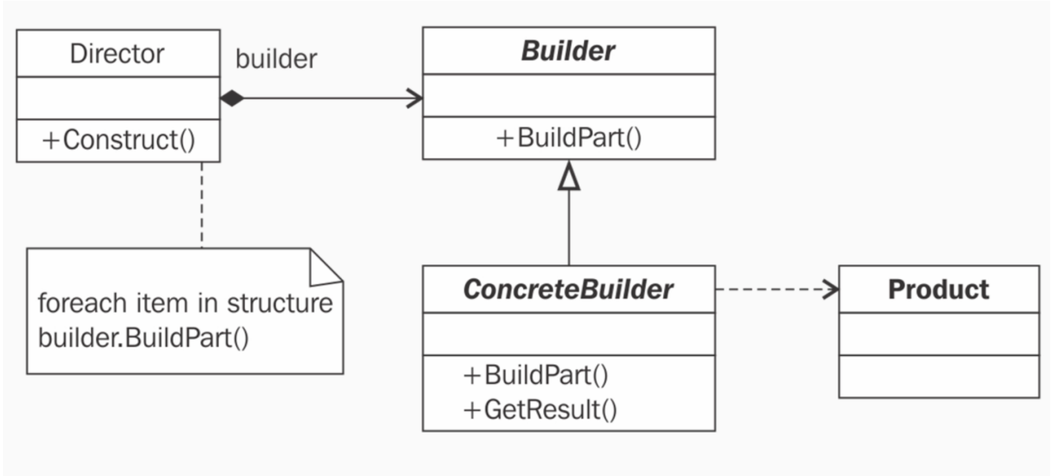


Builder Design Pattern

Director: builds the object using the Builder Interface

```
class Director:

  def get_car(self):
    car = Car()
    # ...
    car.set_body(body)
    car.set_engine(engine)
    car.attach_wheel(wheel)
    # ...
    return car
```



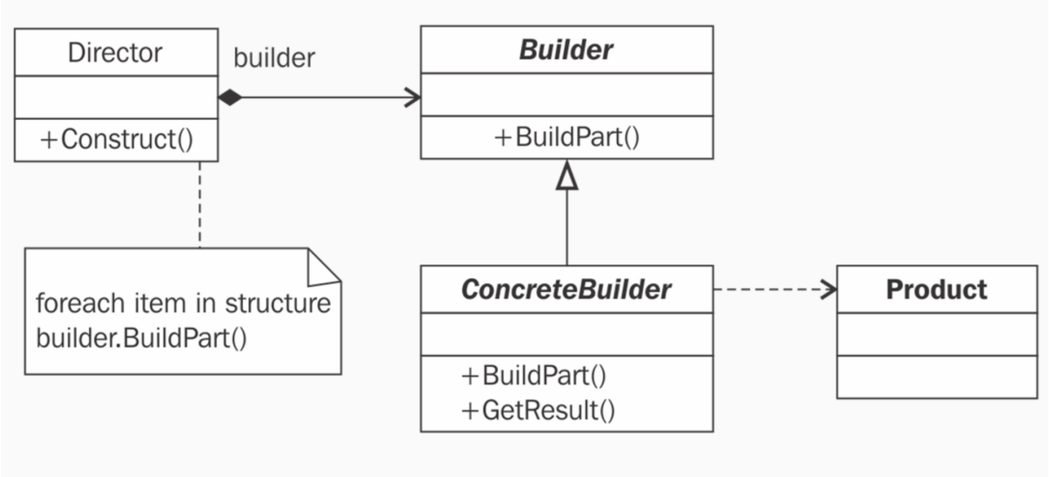
Builder Design Pattern

Builder: defines the abstract interface that creates the parts of the Product object (i.e., the car).

```
class BuilderInterface(ABC):
  @abstractmethod
  def get_body(self):
    pass

  @abstractmethod
  def get_engine(self):
    pass

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



Builder Design Pattern

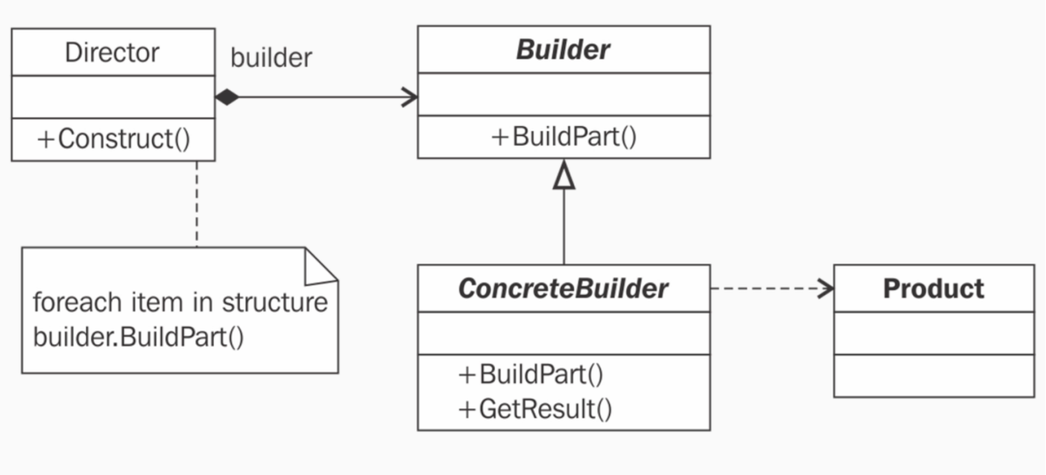
ConcreteBuilder: builds and assembles parts of the product by implementing the Builder interface; defines and tracks the representation it creates.

```
class JeepBuilder(BuilderInterface):

  def get_body(self):
    body = Body('SUV')
    return body

  def get_engine(self):
    engine = Engine(400)
    return engine

  def get_wheel(self):
    wheel = Wheel(22)
    return wheel
```

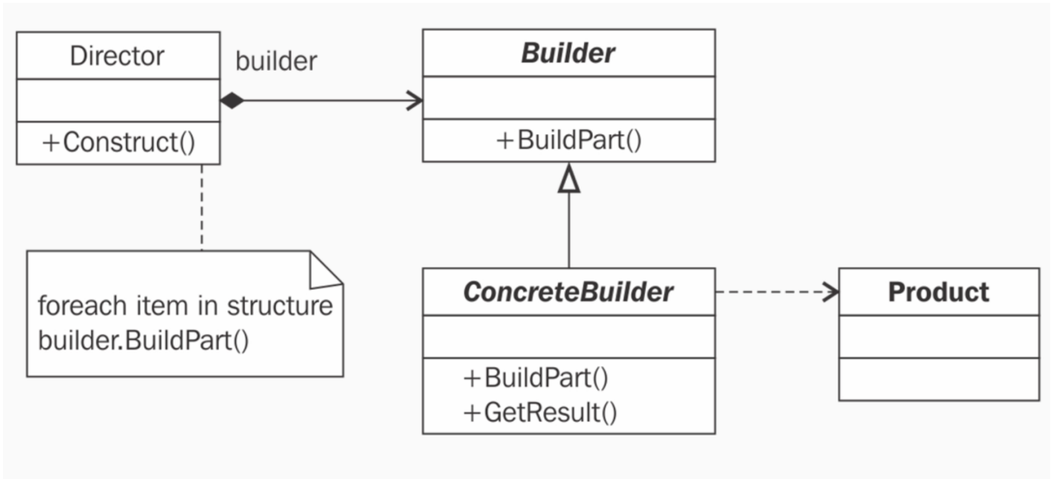


Builder Design Pattern

Product: the complex object and the parts that compose it.

```
class Car:
  def __init__(self):
    self._wheels = list()
    self._engine = None
    self._body = None

  def set_body(self, body):
    # ...
```



Example: `creational/car_builder`

The Builder design pattern solves problems like:
- How can a class (the same construction process) create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

The Builder class focuses on the correct construction of the object. The original class only focuses on how objects work. This simplifies the original class.

The Builder Design Pattern is particularly useful when we want to make sure that an object is valid before instantiating it, and we do not want to add the control logic in the `__init__()` methods.

A builder also allows us to build an object step-by-step, i.e. when parsing text or getting parameters from an interactive interface.

Other Creational patterns:

- Abstract factory
- Builder
- Dependency injection
- Factory method
- Lazy initialization
- Multiton
- Object pool
- Prototype
- RAII
- Singleton