

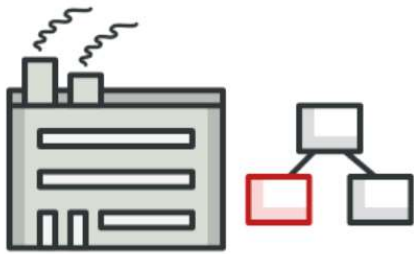
# **Creational Design Pattern in Python**

Creational design patterns deal with object creation mechanisms, which increase flexibility and reuse of existing code.

## **Types of creational design patterns**

1. Factory Method
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton

### **1- Factory Method**



# Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

## Example

### Problem:

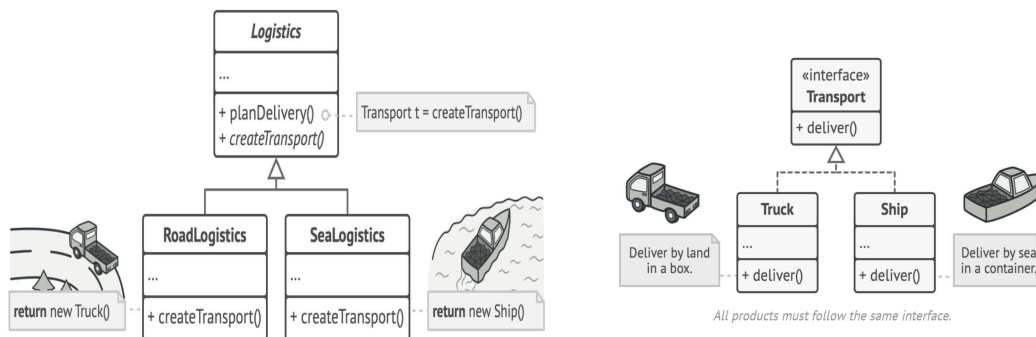
The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class. Then we need to add transportation by ship.



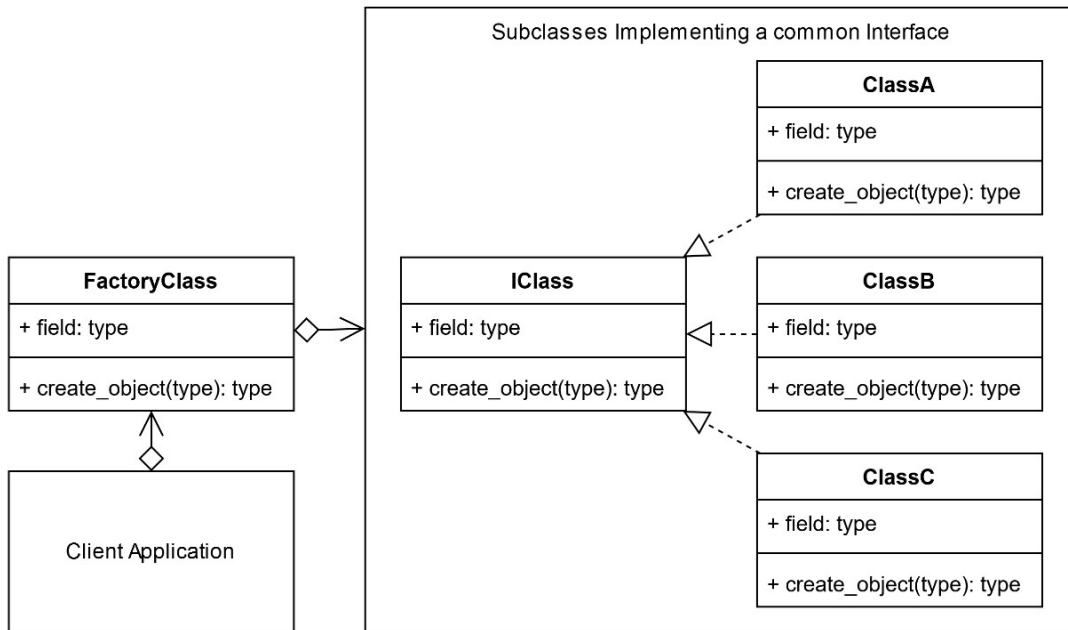
*Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.*

## Solution:

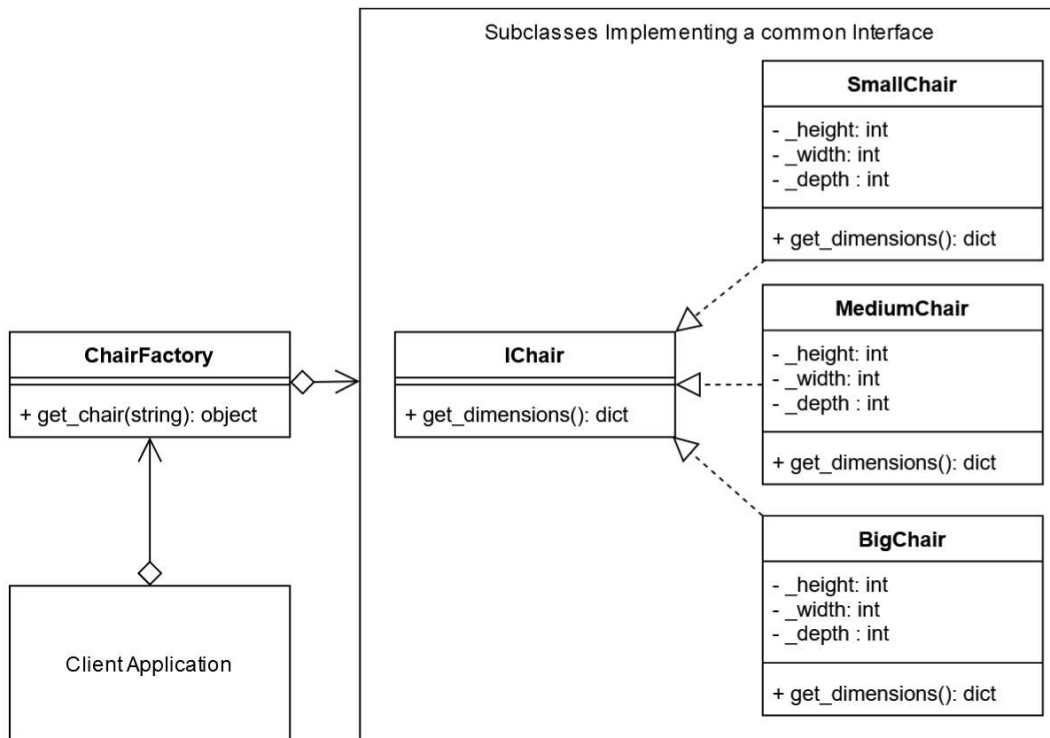
Both Truck and Ship classes should implement the Transport interface, which declares a method called deliver. Each class implements this method differently: trucks deliver cargo by land; ships deliver cargo by sea. The factory method in the RoadLogistics class returns truck objects, whereas the factory method in the SeaLogistics class returns ships.



## Factory Method UML Diagram



## Factory Example UML Diagram



## CODE

```
"The Chair Interface"
from abc import ABCMeta, abstractmethod

class IChair(metaclass=ABCMeta):
    "The Chair Interface (Product)"

    @staticmethod
    @abstractmethod
    def get_dimensions():
        "A static interface method"
```

```
class SmallChair(IChair):
    "The Small Chair Concrete Class implements the IChair interface"

    def __init__(self):
        self._height = 40
        self._width = 40
        self._depth = 40

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

```
class MediumChair(IChair):
    "The Medium Chair Concrete Class implements the ICchair interface"

    def __init__(self):
        self._height = 60
        self._width = 60
        self._depth = 60

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

```
class BigChair(ICchair):
    "The Big Chair Concrete Class implements the ICchair interface"

    def __init__(self):
        self._height = 80
        self._width = 80
        self._depth = 80

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

```
class ChairFactory:
    "The Factory Class"

    @staticmethod
    def get_chair(chair):
        "A static method to get a chair"
        if chair == 'BigChair':
            return BigChair()
        if chair == 'MediumChair':
            return MediumChair()
        if chair == 'SmallChair':
            return SmallChair()
        return None
```

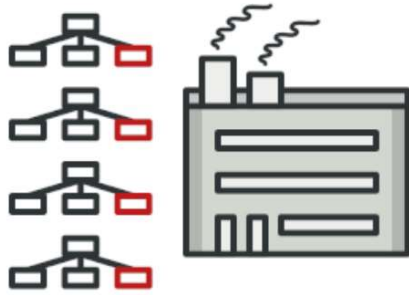
```
# The Client
CHAIR = ChairFactory.get_chair("SmallChair")
print(CHAIR.get_dimensions())
```

## The Output:

---

```
{'width': 40, 'depth': 40, 'height': 40}
```

## 2- Abstract Factory



### Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

**(think if it as a Factory that can return Factories.)**










### Example

#### Problem:

A family of related products: Chair + Sofa + CoffeeTable.

Several variants of this family: Modern, Victorian, ArtDeco.



	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

*Product families and their variants.*

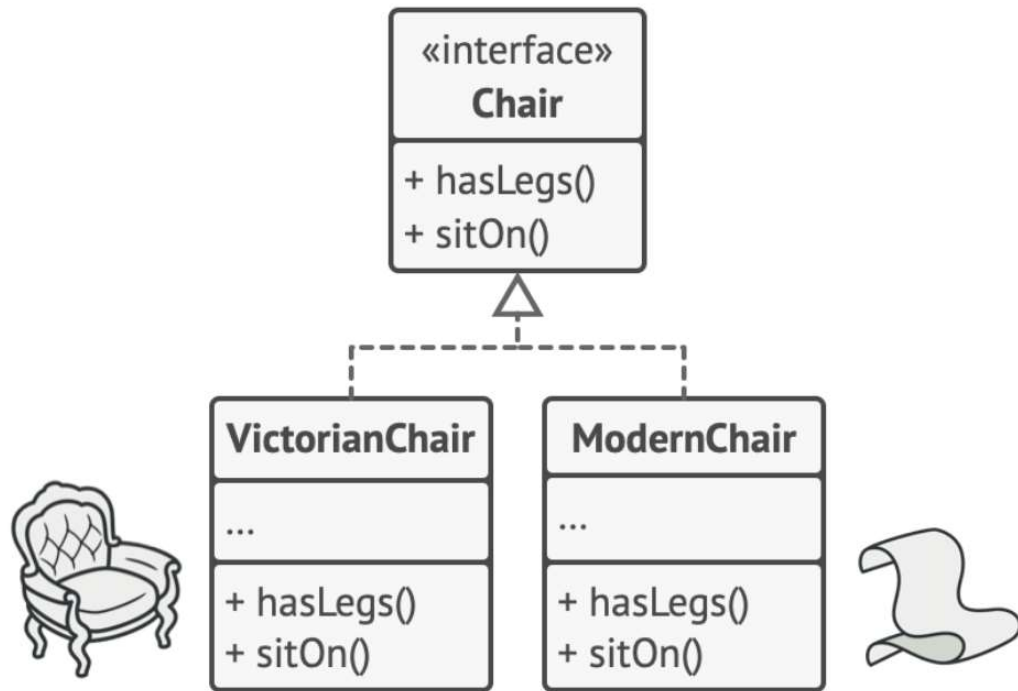
We need a way to create individual furniture objects so that they match other objects of the same family.



*A Modern-style sofa doesn't match Victorian-style chairs.*

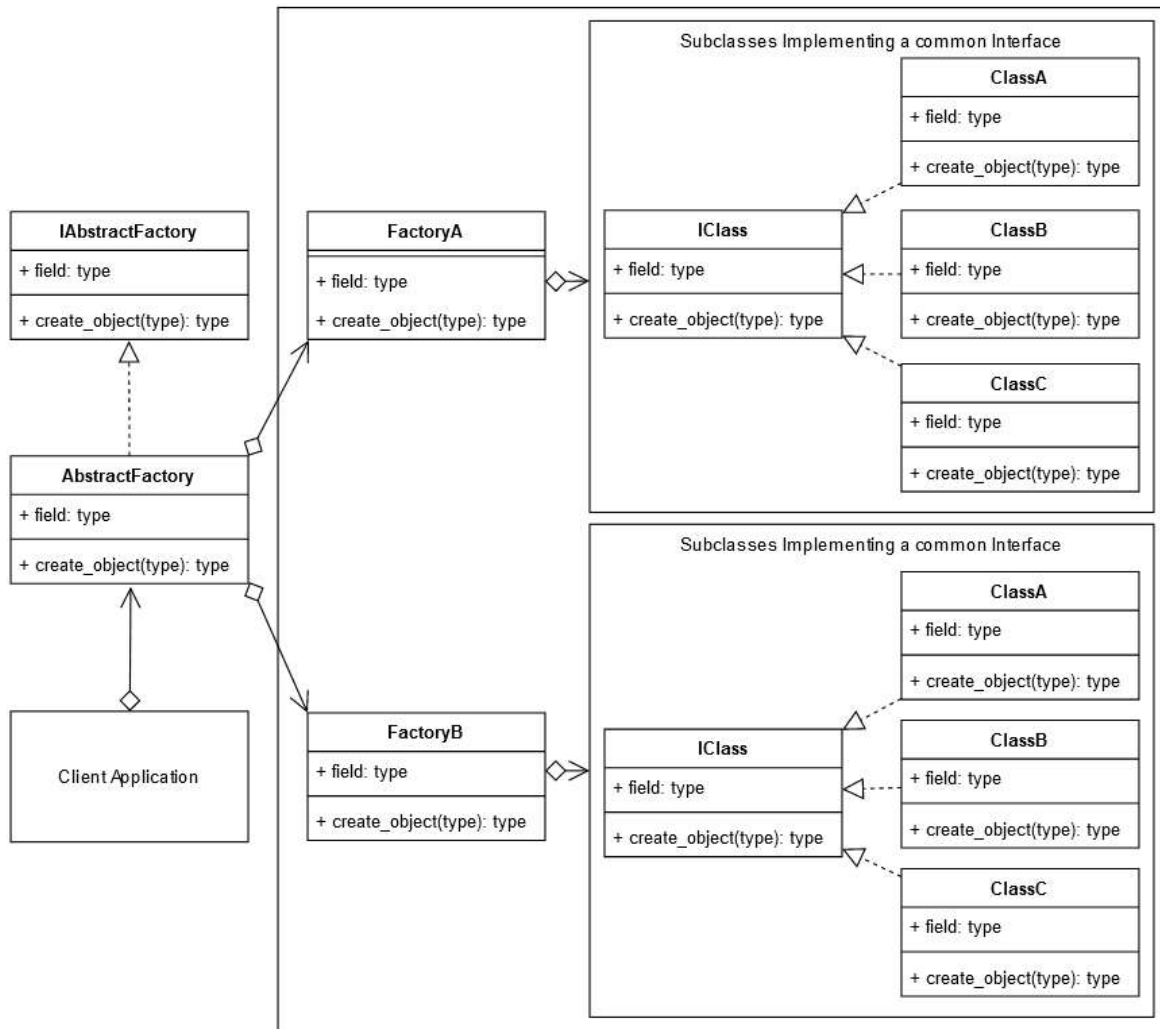
## **Solution:**

Declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products following those interfaces.

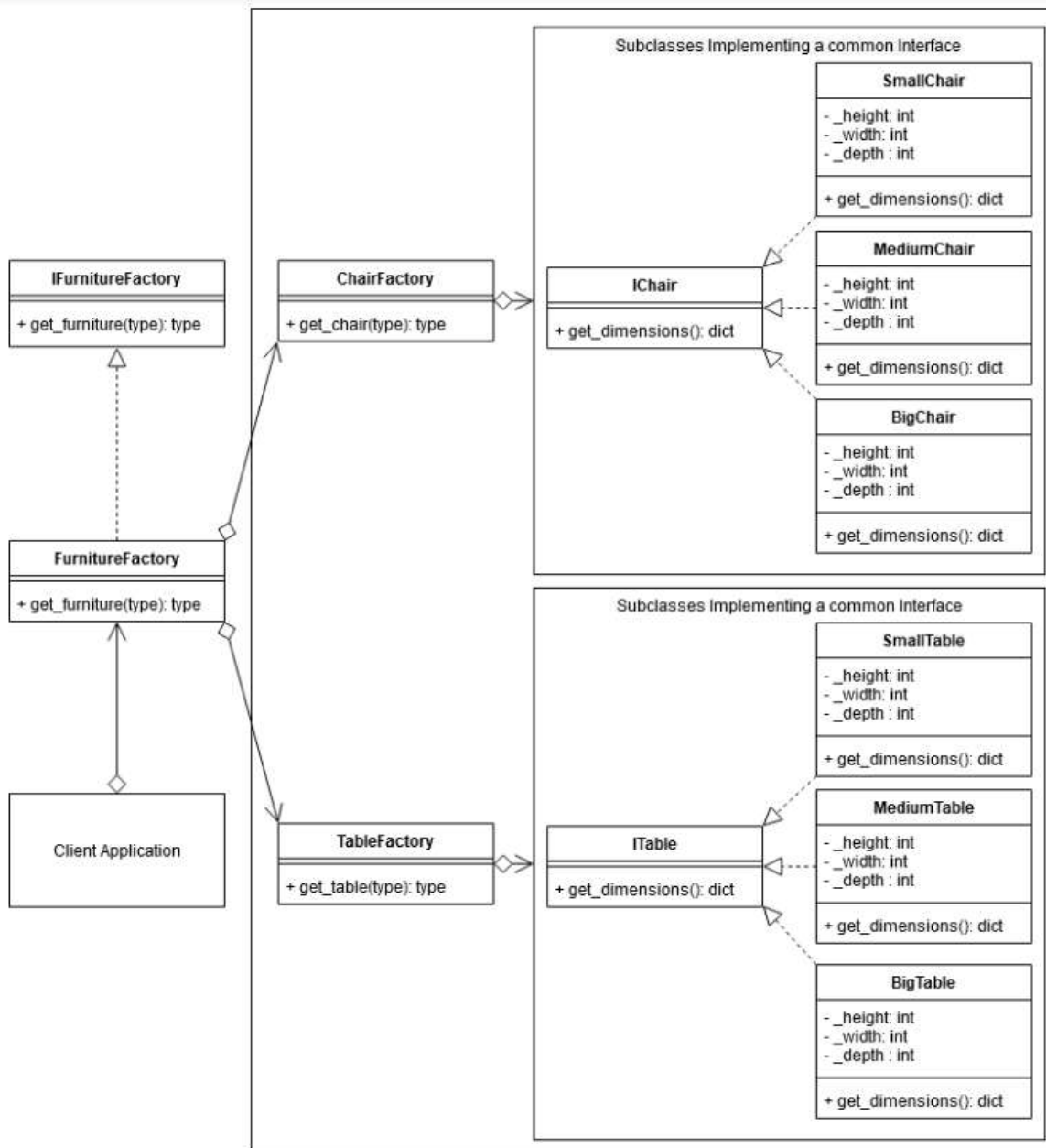


*All variants of the same object must be moved to a single class hierarchy.*

## Abstract Factory UML Diagram



**Abstract Factory Example UML Diagram**



## CODE

```

"The Abstract Factory Interface"
from abc import ABCMeta, abstractmethod
class IFurnitureFactory(metaclass=ABCMeta):
    "Abstract Furniture Factory Interface"

    @staticmethod
    @abstractmethod
    def get_furniture(furniture):
        "The static Abstract factory interface method"

class FurnitureFactory(IFurnitureFactory):
    "The Abstract Factory Concrete Class"

    @staticmethod
    def get_furniture(furniture):
        "Static get_factory method"
        try:
            if furniture in ['SmallChair', 'MediumChair', 'BigChair']:
                return ChairFactory.get_chair(furniture)
            if furniture in ['SmallTable', 'MediumTable', 'BigTable']:
                return TableFactory.get_table(furniture)
            raise Exception('No Factory Found')
        except Exception as _e:
            print(_e)
        return None

```

```

class ChairFactory:
    "The Factory Class"

    @staticmethod
    def get_chair(chair):
        "A static method to get a chair"
        try:
            if chair == 'BigChair':
                return BigChair()
            if chair == 'MediumChair':
                return MediumChair()
            if chair == 'SmallChair':
                return SmallChair()
            raise Exception('Chair Not Found')
        except Exception as _e:
            print(_e)
        return None

"The Chair Interface"
class IChair(metaclass=ABCMeta):
    "The Chair Interface (Product)"

    @staticmethod
    @abstractmethod
    def get_dimensions():
        "A static interface method"

```

```

"A Class of Chair"
class SmallChair(IChair):
    "The Small Chair Concrete Class implements the IChair interface"

    def __init__(self):
        self._height = 40
        self._width = 40
        self._depth = 40

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }

"A Class of Chair"
class MediumChair(IChair):
    """The Medium Chair Concrete Class implements the IChair interface"""

    def __init__(self):
        self._height = 60
        self._width = 60
        self._depth = 60

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }

"A Class of Chair"
class BigChair(IChair):
    "The Big Chair Concrete Class that implements the IChair interface"

    def __init__(self):
        self._height = 80
        self._width = 80
        self._depth = 80

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }

```



---

```

"The Factory Class"
class TableFactory:
    "The Factory Class"

    @staticmethod
    def get_table(table):
        "A static method to get a table"
        try:
            if table == 'BigTable':
                return BigTable()
            if table == 'MediumTable':
                return MediumTable()
            if table == 'SmallTable':
                return SmallTable()
            raise Exception('Table Not Found')
        except Exception as _e:
            print(_e)
        return None

"The Table Interface"
from abc import ABCMeta, abstractmethod

class ITable(metaclass=ABCMeta):
    "The Table Interface (Product)"

    @staticmethod
    @abstractmethod
    def get_dimensions():
        "A static interface method"

```

```
"A Class of Table"
class SmallTable(ITable):
    "The Small Table Concrete Class implements the ITable interface"
```

```
    def __init__(self):
        self._height = 60
        self._width = 100
        self._depth = 60

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

```
"A Class of Table"
class MediumTable(ITable):
    "The Medium Table Concrete Class implements the ITable interface"
```

```
    def __init__(self):
        self._height = 60
        self._width = 110
        self._depth = 70

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

```
"A Class of Table"
class BigTable(ITable):
    "The Big Chair Concrete Class implements the ITable interface"
```

```
    def __init__(self):
        self._height = 60
        self._width = 120
        self._depth = 80

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

```
"Abstract Factory Use Case Example Code"
FURNITURE = FurnitureFactory.get_furniture("SmallChair")
print(f'{FURNITURE.__class__} : {FURNITURE.get_dimensions()}')

FURNITURE = FurnitureFactory.get_furniture("MediumTable")
print(f'{FURNITURE.__class__} : {FURNITURE.get_dimensions()}')
```

## The Output:

```
<class '__main__.SmallChair'> : {'width': 40, 'depth': 40, 'height': 40}
<class '__main__.MediumTable'> : {'width': 110, 'depth': 70, 'height': 60}
```

## 3- Builder



## Builder

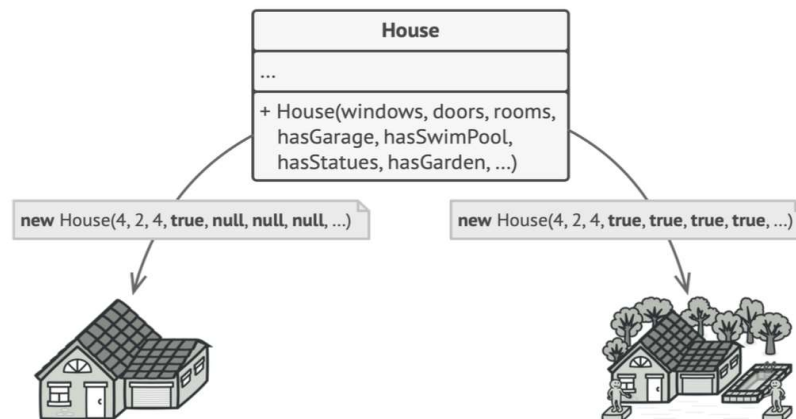
Lets you construct complex objects step by step.  
The pattern allows you to produce different types and representations of an object using the same construction code.

## Example

### Problem:

complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters.

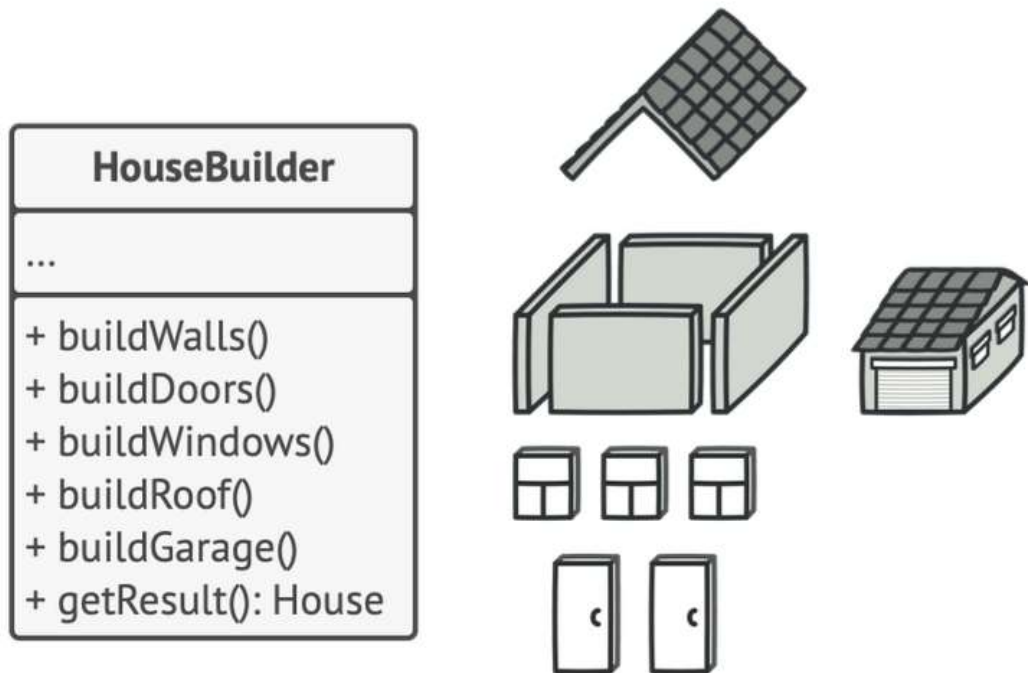
- You can create a giant constructor right in the base House class with all possible parameters that control the house object. (that make problem)



*The constructor with lots of parameters has its downside: not all the parameters are needed at all times.*

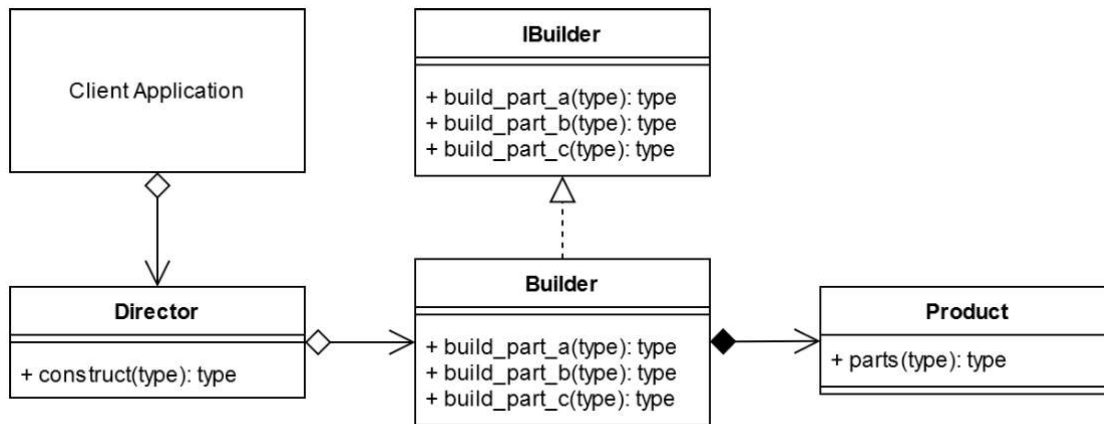
## Solution:

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.



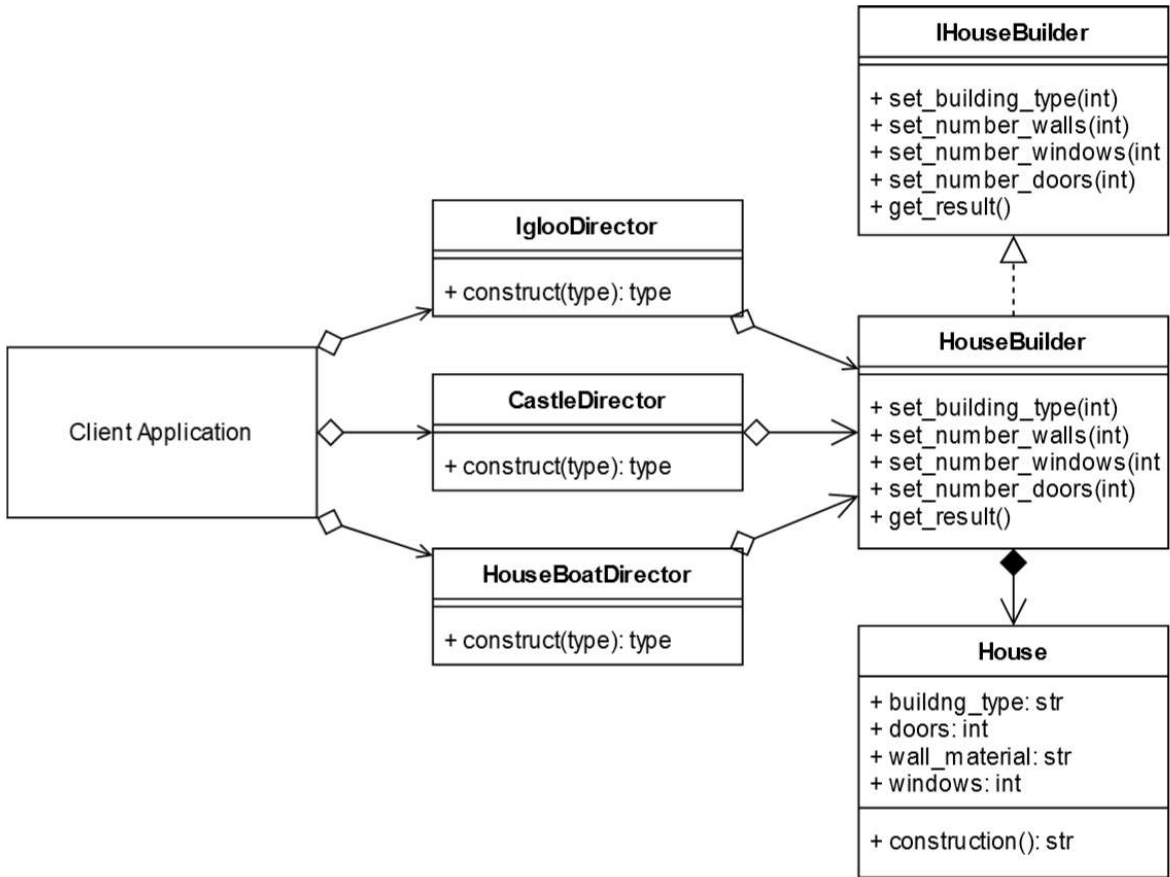
*The director knows which building steps to execute to get a working product.*

## Builder UML Diagram



**Director:** Has a `construct()` method that when called creates a customized product using the methods of the Builder.

## Builder Example UML Diagram



**CODE**

```

: "The Builder Interface"
from abc import ABCMeta, abstractmethod

class IHouseBuilder(metaclass=ABCMeta):
    "The House Builder Interface"

    @staticmethod
    @abstractmethod
    def set_building_type(building_type):
        "Build type"

    @staticmethod
    @abstractmethod
    def set_wall_material(wall_material):
        "Build material"

    @staticmethod
    @abstractmethod
    def set_number_doors(number):
        "Number of doors"

    @staticmethod
    @abstractmethod
    def set_number_windows(number):
        "Number of windows"

    @staticmethod
    @abstractmethod
    def get_result():
        "Return the final product"

```



```
"The Builder Class"
class HouseBuilder(IHouseBuilder):
    "The House Builder."

    def __init__(self):
        self.house = House()

    def set_building_type(self, building_type):
        self.house.building_type = building_type
        return self

    def set_wall_material(self, wall_material):
        self.house.wall_material = wall_material
        return self

    def set_number_doors(self, number):
        self.house.doors = number
        return self

    def set_number_windows(self, number):
        self.house.windows = number
        return self

    def get_result(self):
        return self.house
```

```

"The Product"
class House():
    "The Product"

    def __init__(self, building_type="Apartment", doors=0,
                  windows=0, wall_material="Brick"):
        # brick, wood, straw, ice
        self.wall_material = wall_material
        # Apartment, Bungalow, Caravan, Hut, Castle, Duplex,
        # HouseBoat, Igloo
        self.building_type = building_type
        self.doors = doors
        self.windows = windows

    def construction(self):
        "Returns a string describing the construction"
        return f"This is a {self.wall_material} "\
            f"{self.building_type} with {self.doors} "\
            f"door(s) and {self.windows} window(s)."
```

```

"A Director Class"
class IglooDirector:
    "One of the Directors, that can build a complex representation."

    @staticmethod
    def construct():
        """Constructs and returns the final product
        Note that in this IglooDirector, it has omitted the set_number_of
        windows call since this Igloo will have no windows.
        """
        return HouseBuilder()\
            .set_building_type("Igloo")\
            .set_wall_material("Ice")\
            .set_number_doors(1)\
            .get_result()

"A Director Class"
class CastleDirector:
    "One of the Directors, that can build a complex representation."

    @staticmethod
    def construct():
        "Constructs and returns the final product"
        return HouseBuilder()\
            .set_building_type("Castle")\
            .set_wall_material("Sandstone")\
            .set_number_doors(100)\
            .set_number_windows(200)\
            .get_result()

"A Director Class"
class HouseBoatDirector:
    "One of the Directors, that can build a complex representation."

    @staticmethod
    def construct():
        "Constructs and returns the final product"
        return HouseBuilder()\
            .set_building_type("House Boat")\
            .set_wall_material("Wood")\
            .set_number_doors(6)\
            .set_number_windows(8)\
            .get_result()

```

```
IGLOO = IglooDirector.construct()
CASTLE = CastleDirector.construct()
HOUSEBOAT = HouseBoatDirector.construct()

print(IGLOO.construction())
print(CASTLE.construction())
print(HOUSEBOAT.construction())
```

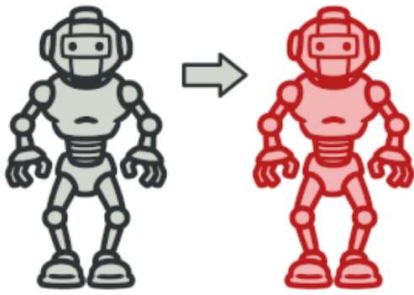
## The Output:

This is a Ice Igloo with 1 door(s) and 0 window(s).

This is a Sandstone Castle with 100 door(s) and 200 window(s).

This is a Wood House Boat with 6 door(s) and 8 window(s).

## 4-Prototype



## Prototype

Lets you copy existing objects without making your code dependent on their classes.

### Example

#### Problem:

you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.



*Copying an object "from the outside" isn't always possible.*

## Solution:

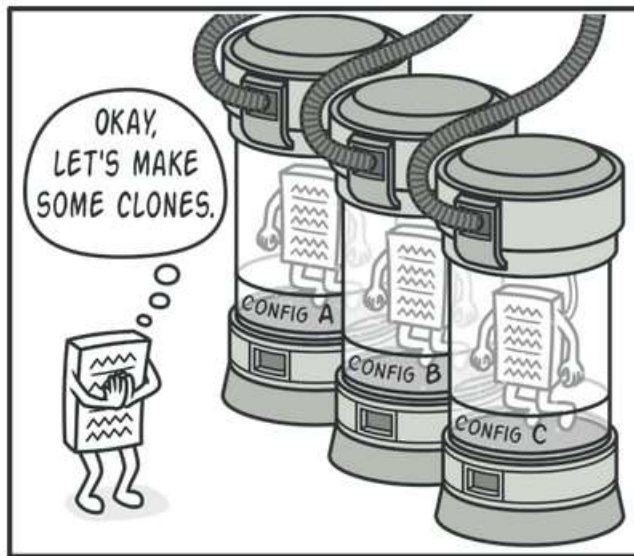
The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single clone method.

The implementation of the clone method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

An object that supports cloning is called a prototype. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

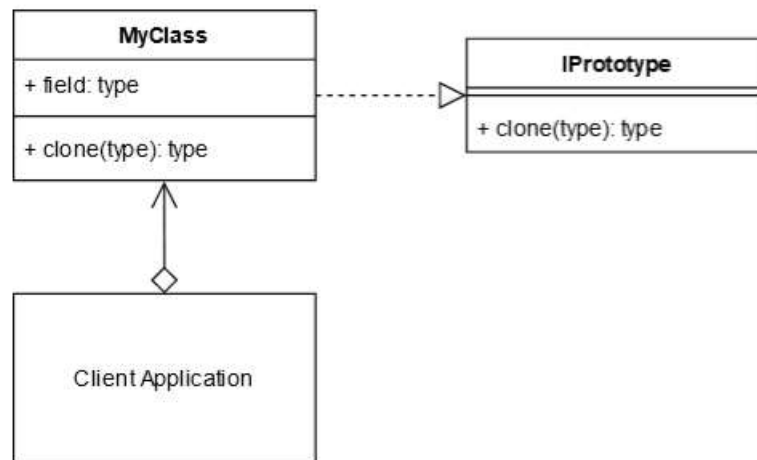
Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.





*Pre-built prototypes can be an alternative to subclassing.*

## Prototype UML Diagram



## CODE

```
: import copy

class Shape:
    "Interface with clone method"
    def clone(self):
        # Using deepcopy to ensure a deep copy of the object
        return copy.deepcopy(self)

class Circle(Shape):
    "A Concrete Class"
    def __init__(self, radius):
        self.radius = radius

    def __str__(self):
        return f"Circle with radius {self.radius}"

# The Client
circle = Circle(5)
clone_circle = circle.clone()

# Comparing the original object and the cloned object
print(circle)
print(clone_circle)
```



## Output

```
Circle with radius 5  
Circle with radius 5
```

## 5-Singleton



### Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

## Example

**Problem:**

Sometimes we only ever need (or want) one instance of a particular class.

**singleton: An object that is the only object of its type.  
Ensuring that a class has at most one instance.**

- suppose you are working on a big project and using a complex object with data. There are many points when you need to get and put data in an object. There may be a chance that if you (Or a new developer) get an object of the class by mistake, then it will return a new object with blank data. But in the case of the singleton class, it retrieves a singleton object, that is common and global from each point.
- As you can see in the image, A non-singleton class can have a different object in different ways. But a singleton class provides only one object in a single way. You can get objects in any other way.



Non singleton class

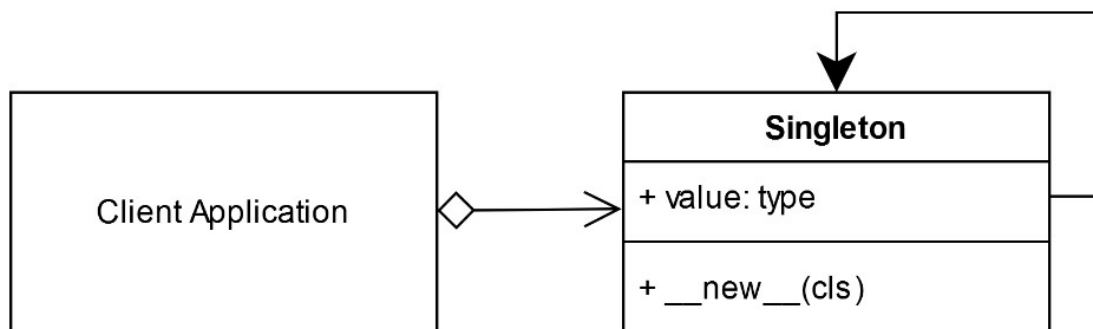


Singleton class

## Solution:

Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

## Singleton UML Diagram



## CODE

```

"Singleton Concept Sample Code"
class Logger:
    "The Singleton Class"
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Initialization code can go here
        return cls._instance

    def log(self, message):
        print(message)

# client
logger1 = Logger()
logger2 = Logger()

# test if logger1=logger2
logger1.log("Hello from logger1")
logger2.log("Hello from logger2")

print(logger1 == logger2)

```

The `_instance` class attribute is used to store the single instance of `Logger`.

The `__new__` method is overridden to check if the instance already exists. If it doesn't, it creates one using `super(Singleton, cls)`; otherwise, it returns the existing instance.

## Output

```
Hello from logger1  
Hello from logger2  
True
```

