Creational Design Patterns

Instructor: Mehroze Khan

Creational Design Patterns

- Creational design patterns provide various **object creation** mechanisms, which increase **flexibility** and **reuse** of existing code.
- Creational design patterns **abstract** the instantiation process.
- They help make a system independent of how its objects are created, composed, and represented.
- A **class creational pattern** uses inheritance to vary the class that's instantiated.
- An **object creational pattern** will delegate instantiation to another object.

Creational Design Patterns

- First, they all encapsulate knowledge about which concrete classes the system uses.
- Second, they hide how instances of these classes are created and put together.
- Creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when.

Catalog of Design Patterns

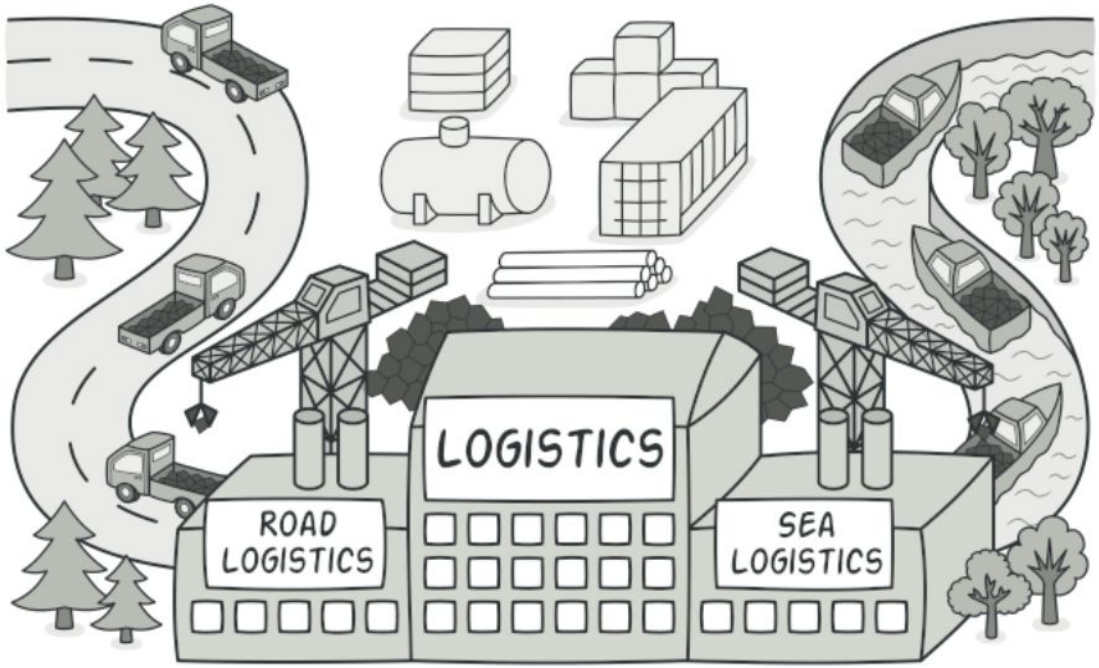
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Factory Method Pattern

Factory Method

Intent:

- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
- It is also known as **Virtual Constructor**



Problem

- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- After a while, your app becomes popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

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

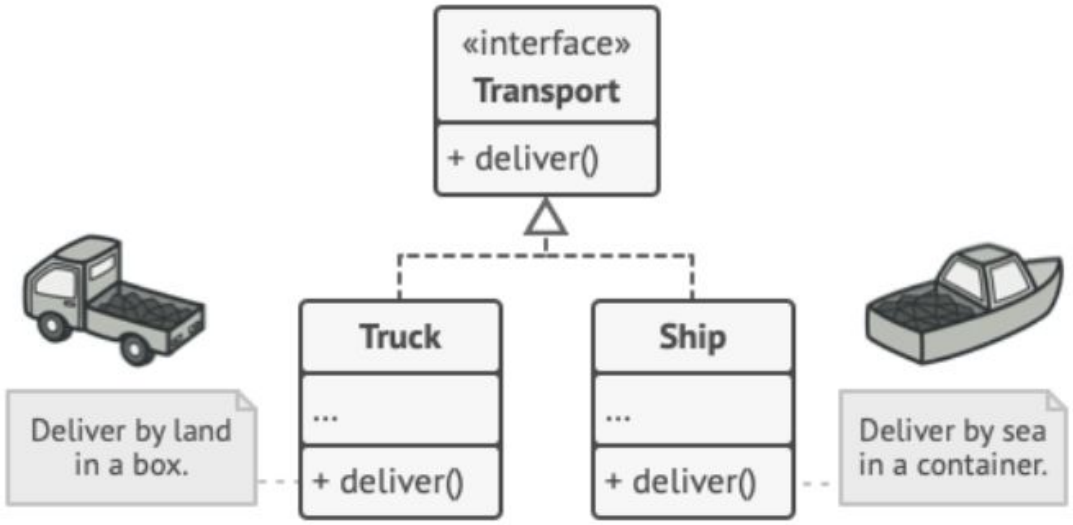
Problem

- At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase.
- Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all these changes again.
- As a result, you will end up with nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

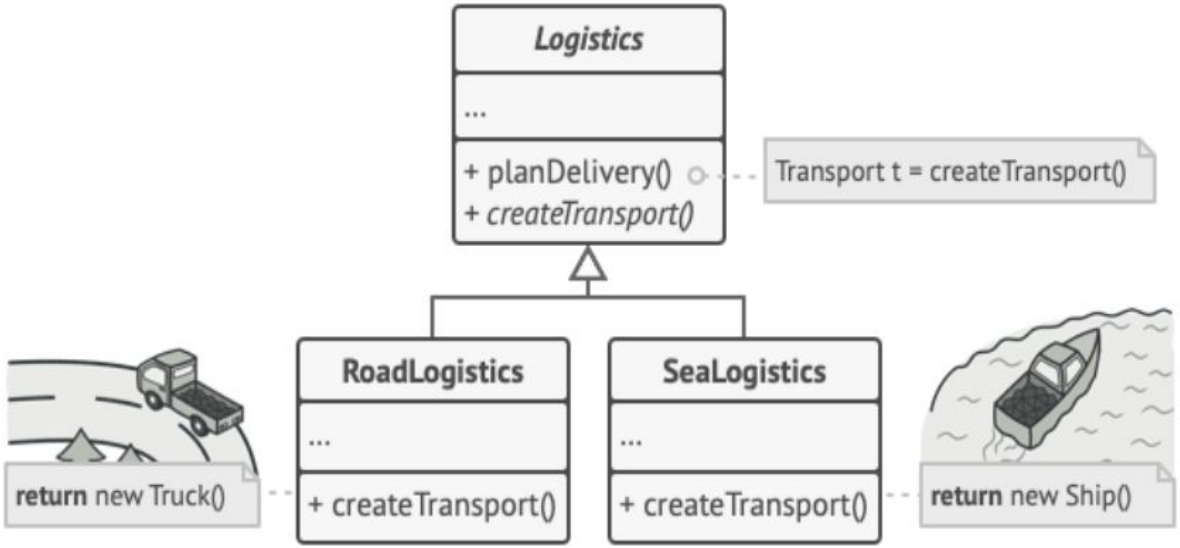
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.
- Objects are still created, but it's being called from within the factory method.
- Objects returned by a factory method are often referred to as **products**.



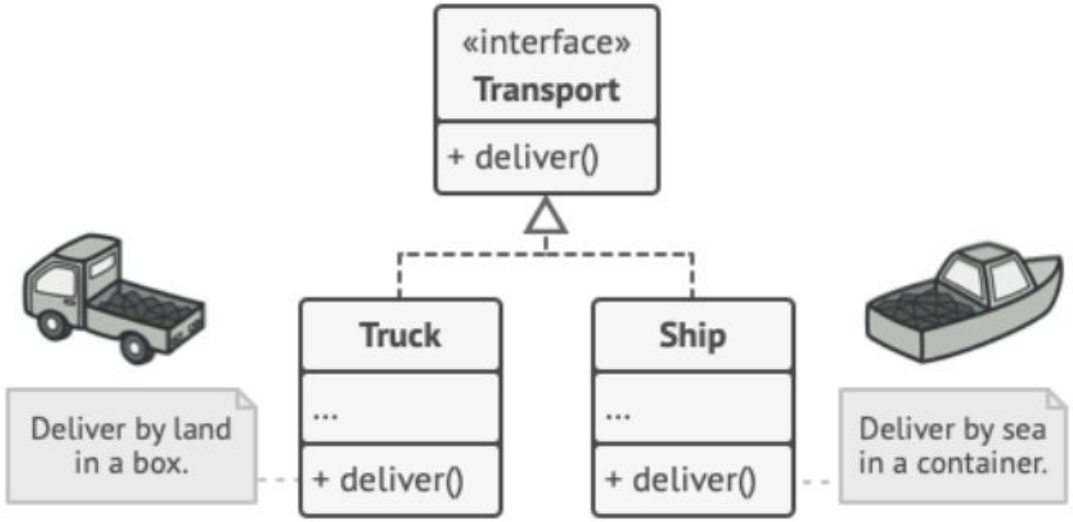
All products must follow the same interface.



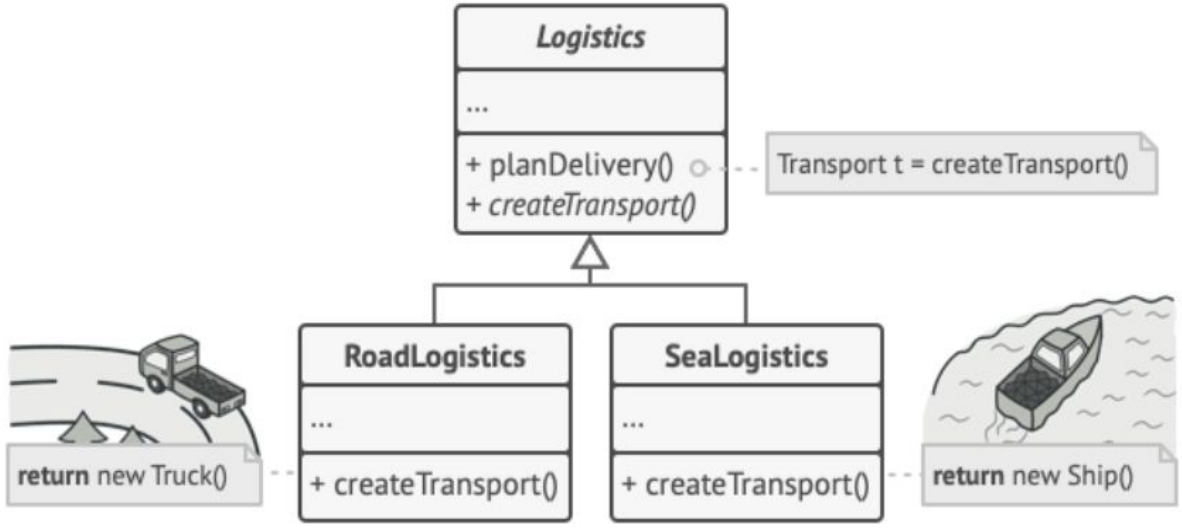
Subclasses can alter the class of objects being returned by the factory method.

Solution

- At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another.
- However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.
- There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface.
- Also, the factory method in the base class should have its return type declared as this interface.



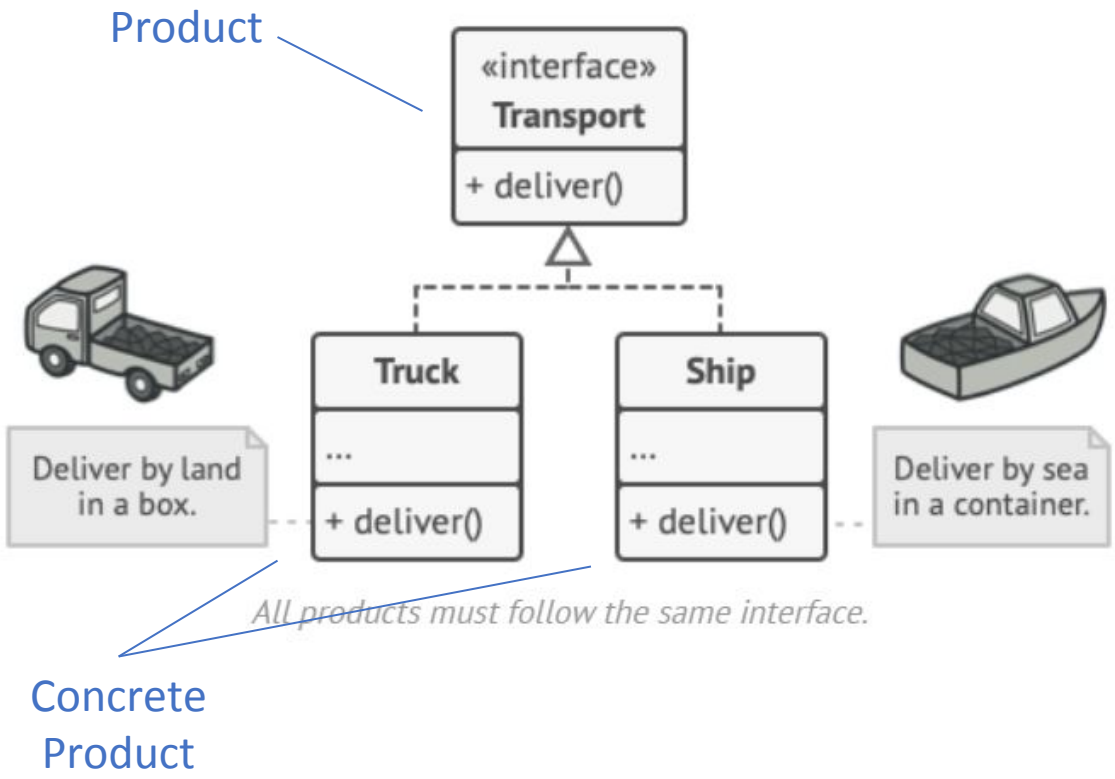
All products must follow the same interface.

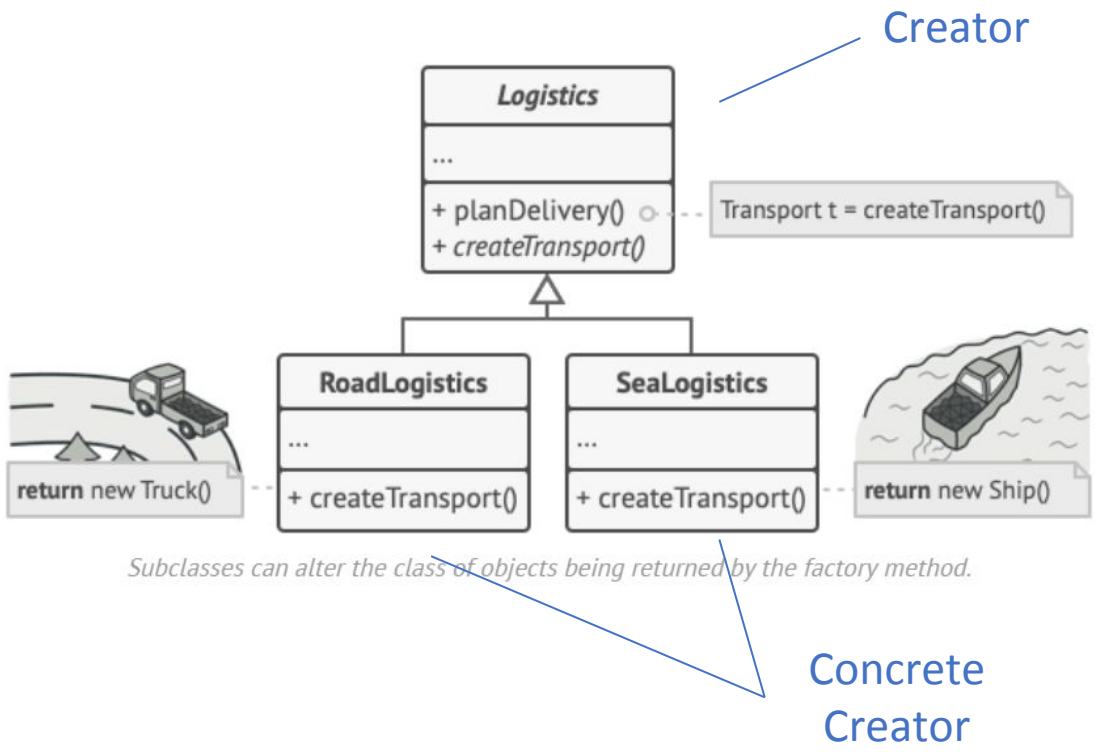


Subclasses can alter the class of objects being returned by the factory method.

Solution

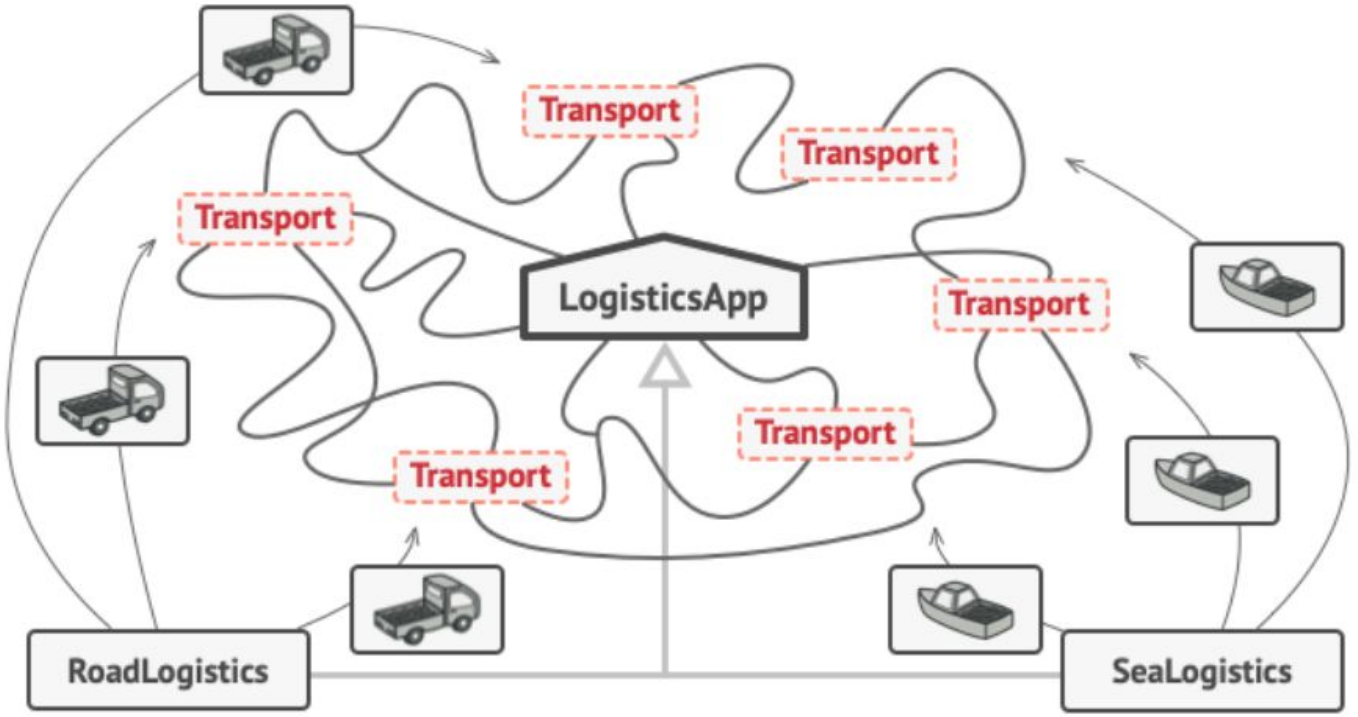
- For example, both Truck and Ship classes should implement the Transport interface, which declares a method called deliver.
- Each class implements this method differently: trucks deliver cargo by land and ships deliver cargo by sea.
- The factory method in the RoadLogistics class returns truck objects,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





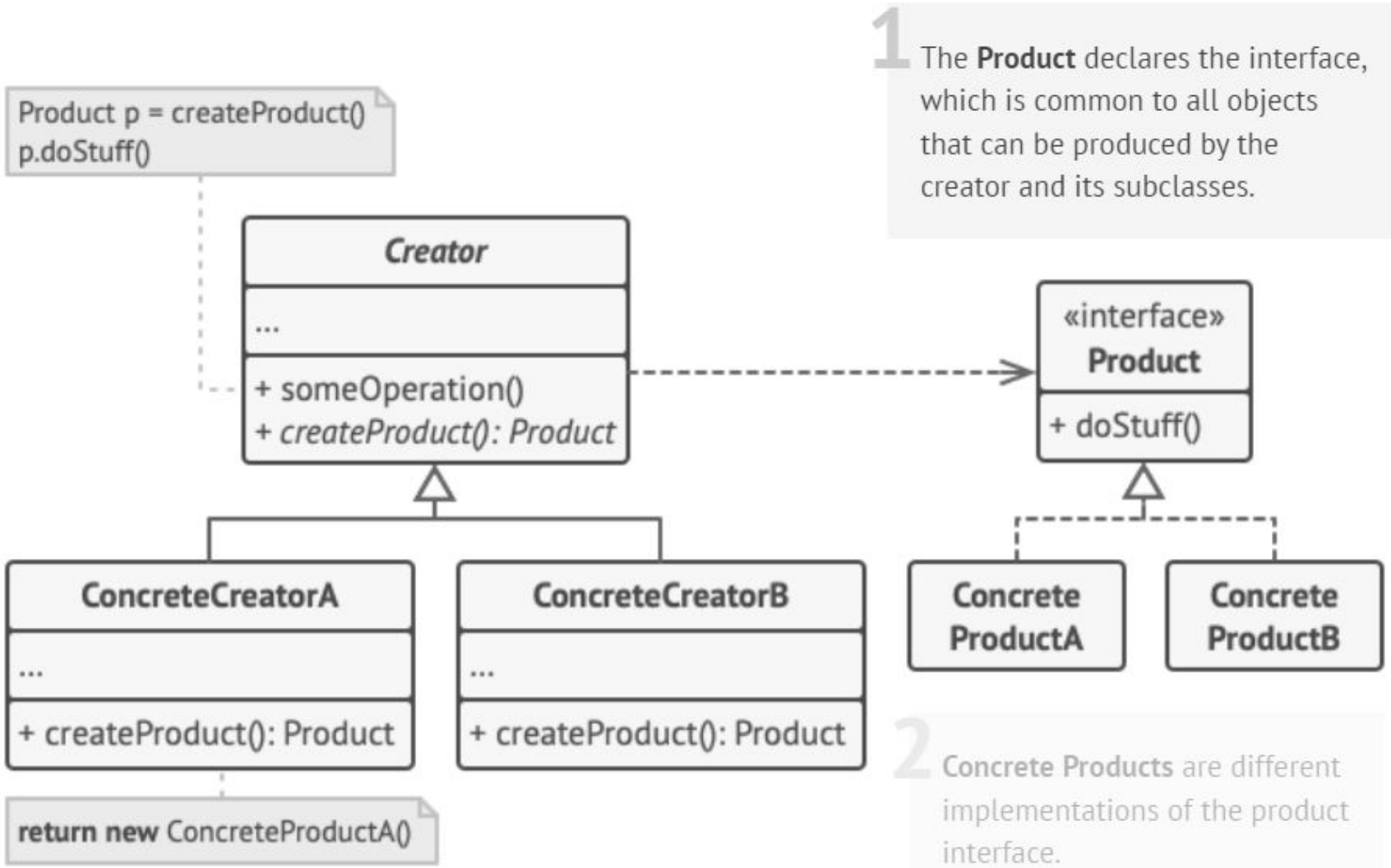
Solution

- The code that uses the factory method (often called the **client code**) doesn't see a difference between the actual products returned by various subclasses.
- The client treats all the products as abstract Transport.
- The client knows that all transport objects are supposed to have the deliver method, but exactly how it works isn't important to the client.

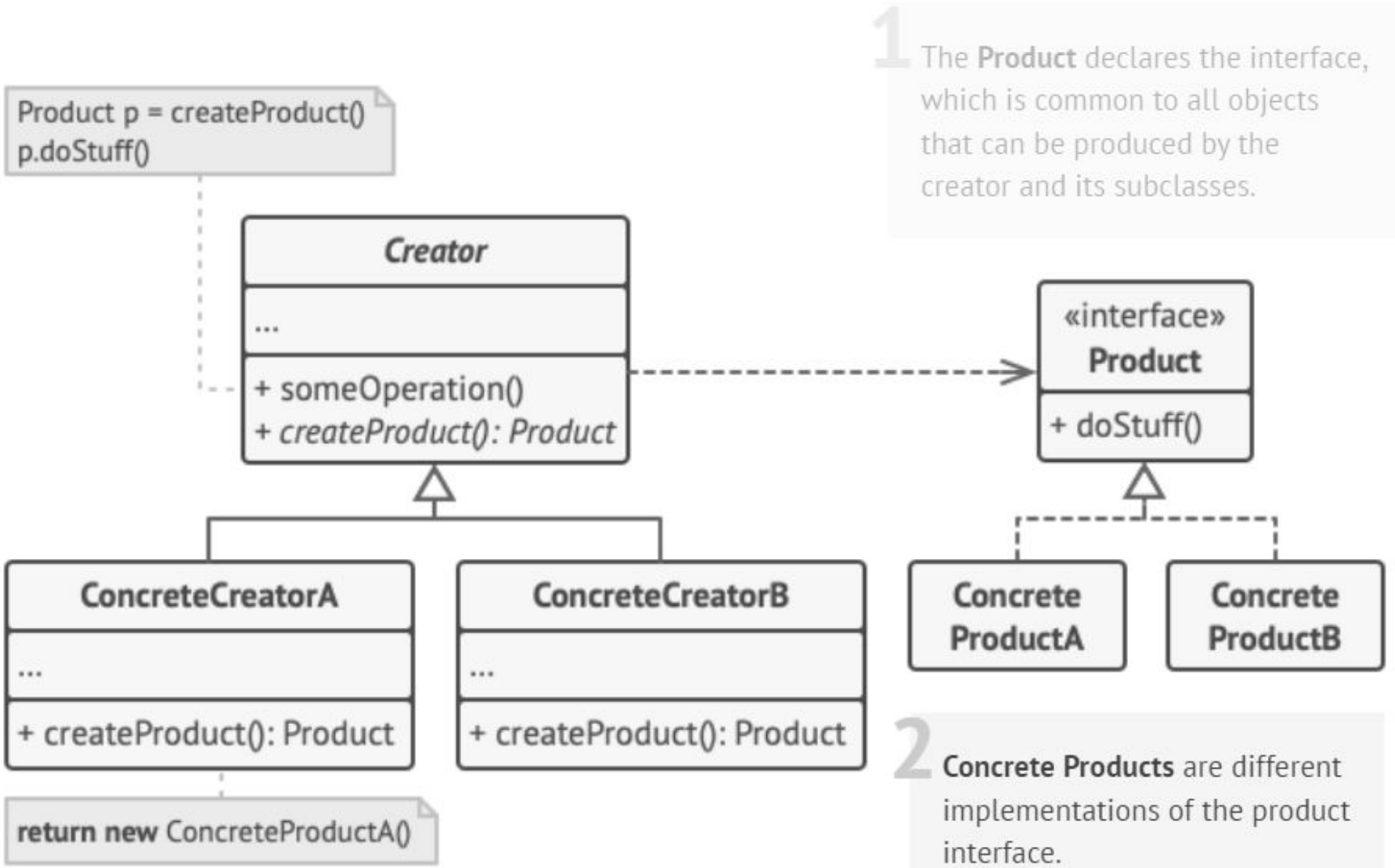


As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.

Structure


1 The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
«interface»
Product
+ doStuff()
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
Concrete ProductA
Concrete ProductB
return new ConcreteProductA()
2 Concrete Products are different implementations of the product interface.

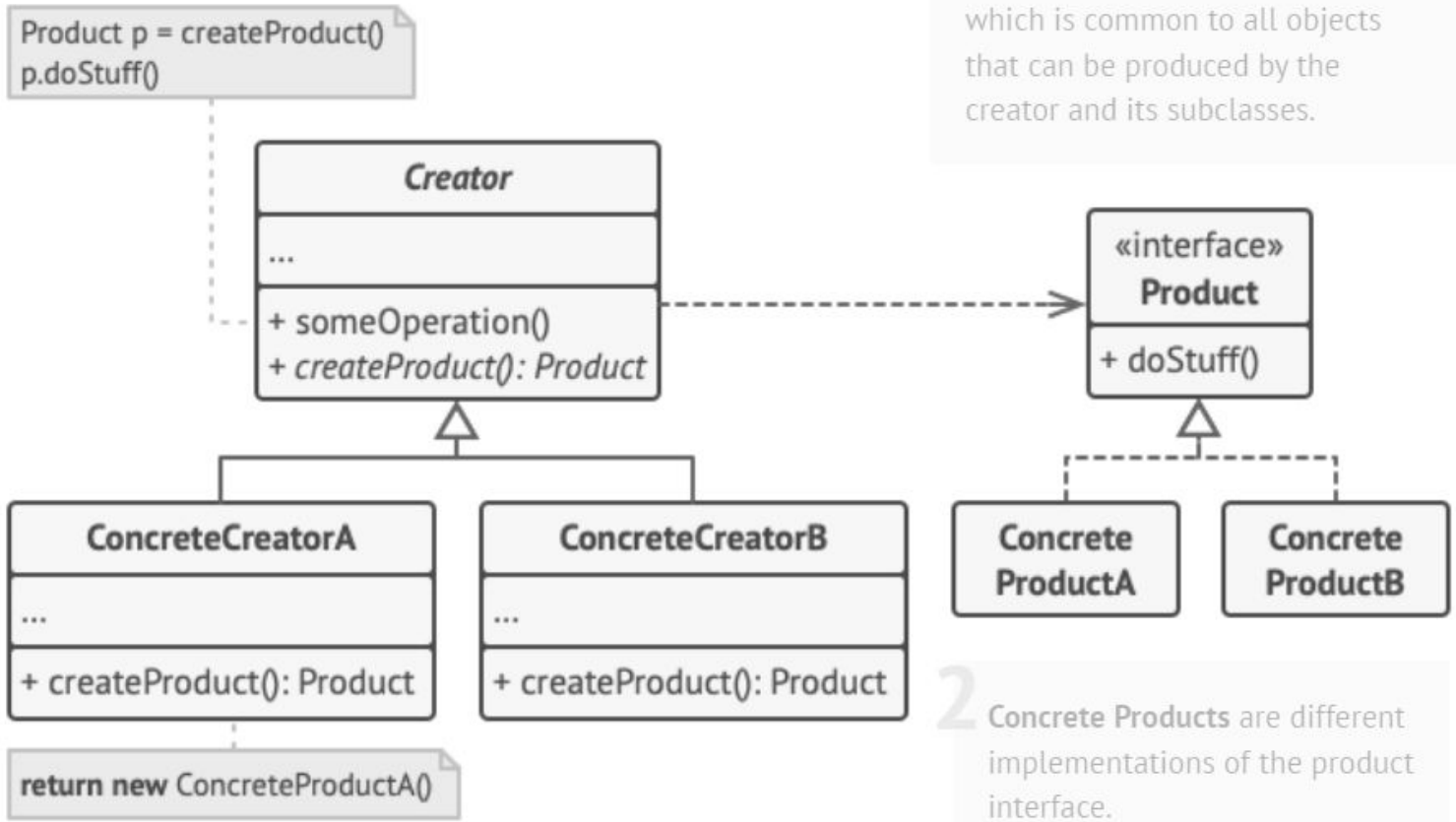
Structure


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
«interface»
Product
+ doStuff()
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
Concrete ProductA
Concrete ProductB
return new ConcreteProductA()
1 The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2 Concrete Products are different implementations of the product interface.

3 The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

You can declare the factory method as `abstract` to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.



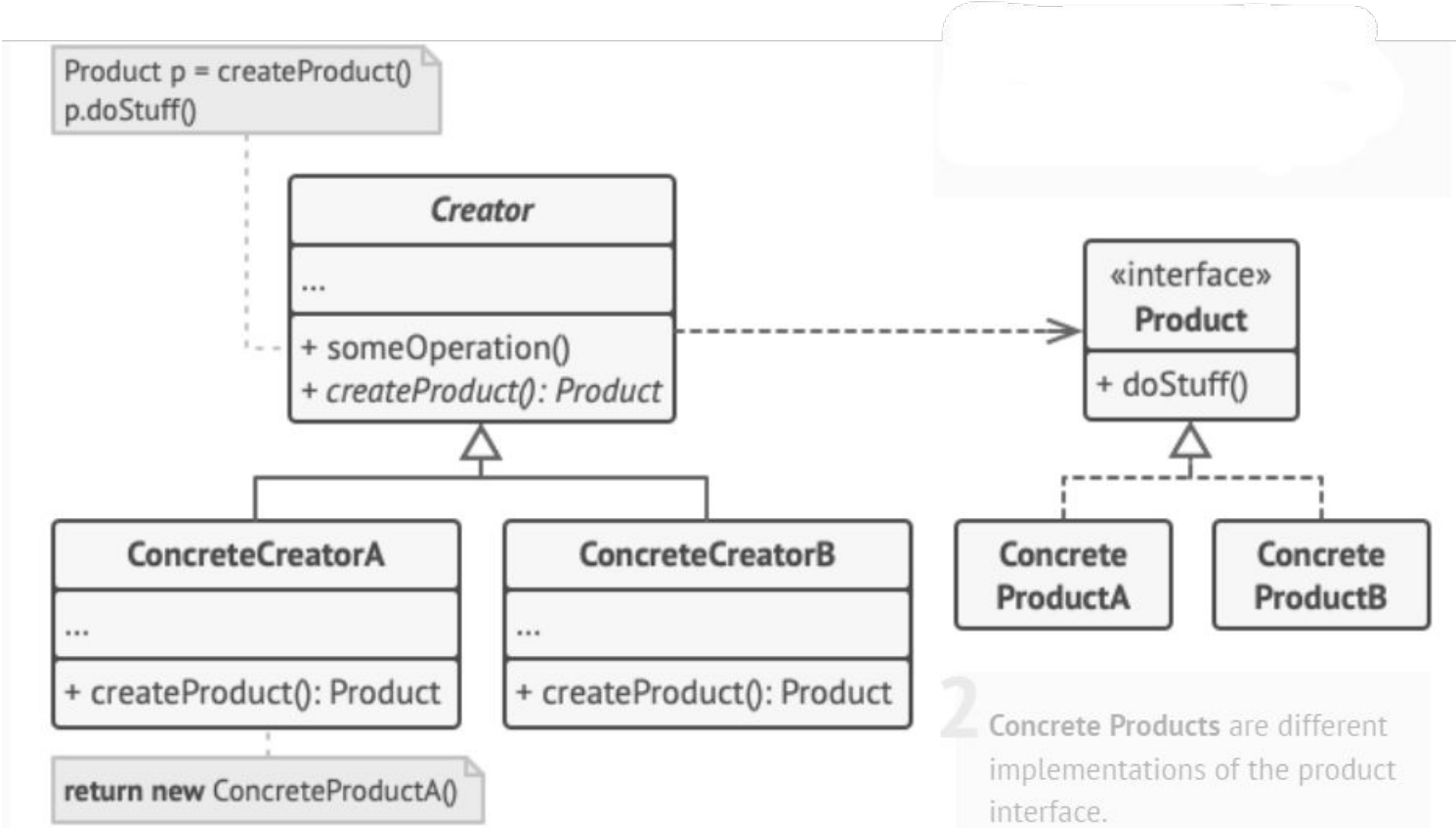
1 The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2 **Concrete Products** are different implementations of the product interface.

4 **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Structure


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
«interface»
Product
+ doStuff()
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
Concrete ProductA
Concrete ProductB
return new ConcreteProductA()
2 Concrete Products are different implementations of the product interface.
4 Concrete Creators override the base factory method so it returns a different type of product.
Note that the factory method doesn't have to create new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Example

```
class Pizza {
public:
    void prepare() const {
        cout << "Preparing a pizza...\n";
    }
};

class Pasta {
public:
    void prepare() const {
        cout << "Preparing a pasta...\n";
    }
};

class Waiter {
public:
    void takeOrder(const string& dishType) {
        if (dishType == "Pizza") {
            Pizza pizza;
            pizza.prepare();
        } else if (dishType == "Pasta") {
            Pasta pasta;
            pasta.prepare();
        } else {
            cout << "Sorry, we don't serve that dish.\n";
        }
    }
};
```

Example

```
int main() {
    Waiter waiter;
    waiter.takeOrder("Pizza");
    waiter.takeOrder("Pasta");
    waiter.takeOrder("Burger");

    return 0;
}
```

Issues

- **Tight Coupling**: Waiter directly depends on Pizza and Pasta classes, making it difficult to add new dishes without modifying Waiter.
- **Violation of Open/Closed Principle:** New dish types require modification in Waiter

Example (Applying Factory Method Pattern)

```cpp
// Dish Interface
class Dish {
public:
    virtual void prepare() const = 0;
};

// Concrete Dishes
class Pizza : public Dish {
public:
    void prepare() const override {
        cout << "Preparing a pizza...\n";
    }
};

class Pasta : public Dish {
public:
    void prepare() const override {
        cout << "Preparing a pasta...\n";
    }
};
```

Example (Applying Factory Method Pattern)

```
// Chef Interface
class Chef {
public:
    virtual Dish* createDish() const = 0;
};

// Concrete Chefs
class PizzaChef : public Chef {
public:
    Dish* createDish() const override {
        return new Pizza();
    }
};

class PastaChef : public Chef {
public:
    Dish* createDish() const override {
        return new Pasta();
    }
};
```

Example (Applying Factory Method Pattern)

```
// Waiter function that uses Chef objects
void waiterTakesOrder(const Chef& chef) {
    Dish* dish = chef.createDish();
    dish->prepare();
}

int main() {
    PizzaChef pizzaChef;
    PastaChef pastaChef;

    cout << "Waiter taking order for pizza:\n";
    waiterTakesOrder(pizzaChef);

    cout << "Waiter taking order for pasta:\n";
    waiterTakesOrder(pastaChef);

    return 0;
}
```

Advantages

- **Reduced Coupling:** Waiter no longer needs to know about specific dishes like Pizza or Pasta. It only interacts with the Chef interface, which makes it easy to add new dishes without modifying Waiter.
- **Scalability:** To add a new dish (e.g., Burger), simply create a new Burger class and BurgerChef class without changing the Waiter.

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.
- Helping Links:
- https://refactoring.guru/design-patterns/
- https://www.geeksforgeeks.org/factory-method-for-designing-pattern/
- https://sourcemaking.com/design_patterns/