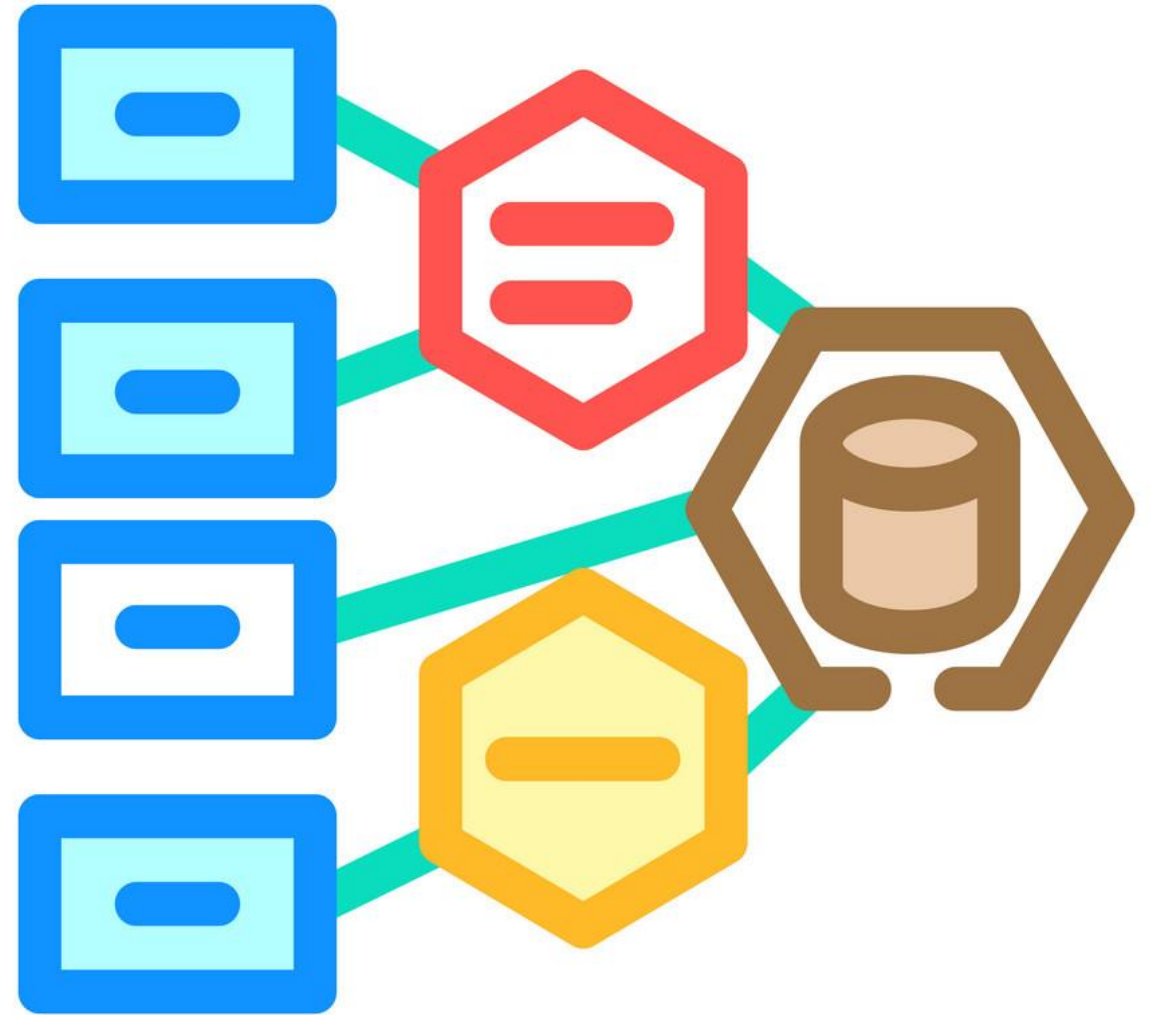


Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Creational patterns

Factory
Abstract factory
Builder
Prototype
Singleton

Structural patterns

Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

Behavioral patterns

Chain of responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Template method
Visitor

Creational patterns

- ✓ **Factory**
- ✓ Abstract factory
- ✓ Builder
- ✓ Prototype
- ✓ Singleton

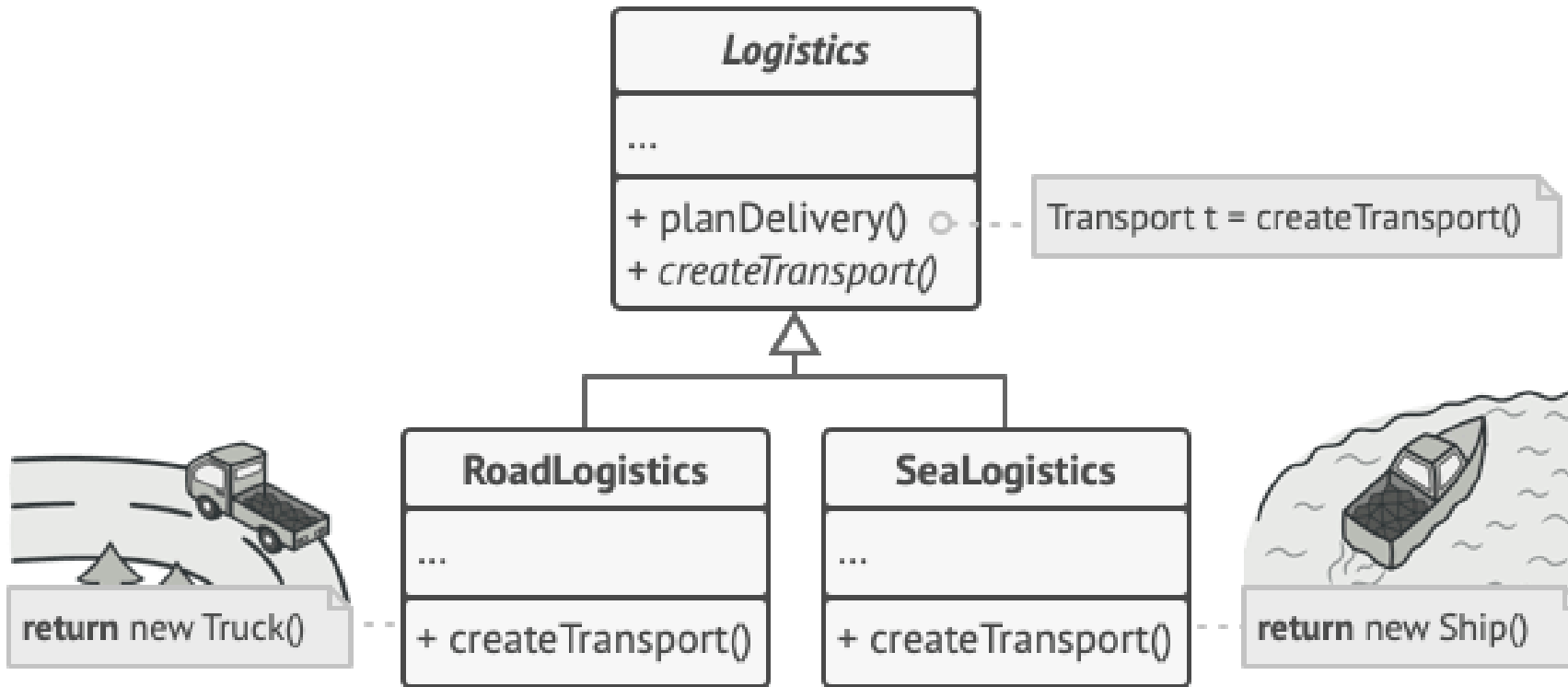
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

- 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 pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

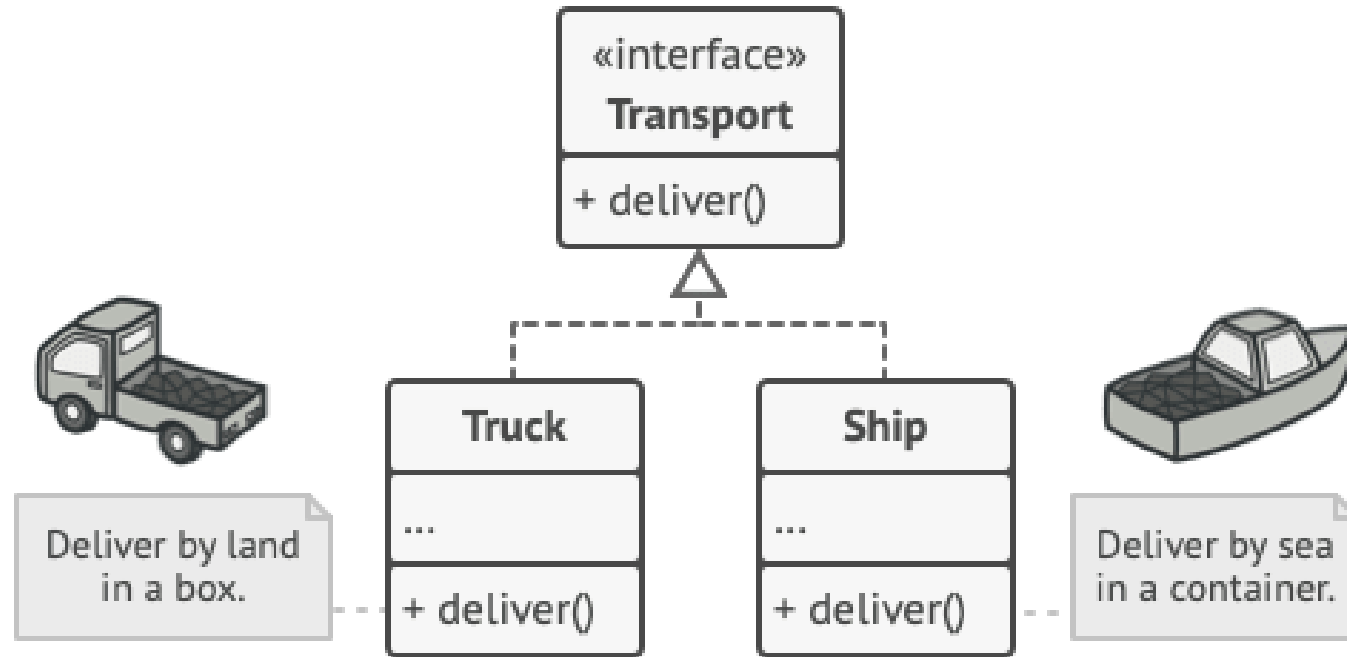
- At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.
- As a result, you will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



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



- 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.



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 handles **object creation**, while polymorphism focuses on **behavior at runtime**. Factory method delegates object creation to subclasses while ensuring flexibility and reuse.

What's new? It decouples object creation from the main logic, enabling flexibility and scalability

Steps to Implement Factory Method:

1. Create an interface/abstract class for the product.

```
class Shape:  
    def draw(self):  
        pass
```

2. Define concrete implementations of the product.

```
class Circle(Shape):  
    def draw(self):  
        print("Drawing a Circle.")
```

```
class Square(Shape):  
    def draw(self):  
        print("Drawing a Square.")
```

3. Create a factory class/method that uses polymorphism to create and return product objects.

```
class ShapeFactory:
    def create_shape(self, shape_type):
        if shape_type == "circle":
            return Circle()
        elif shape_type == "square":
            return Square()
        else:
            raise ValueError("Unknown shape type")
```

Client code

```
factory = ShapeFactory()  
shape1 = factory.create_shape("circle")  
shape1.draw() # Output: Drawing a Circle.
```

```
shape2 = factory.create_shape("square")  
shape2.draw() # Output: Drawing a Square.
```



- You avoid tight coupling between the creator and the concrete products.
- **Single Responsibility Principle.** You can move the product creation code into one place in the program, making the code easier to support.
- **Open/Closed Principle.** You can introduce new types of products into the program without breaking existing client code.



The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern.

The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Creational patterns










- ✓ Factory
- ✓ **Abstract factory**
- ✓ Builder
- ✓ Prototype
- ✓ Singleton

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

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

Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.

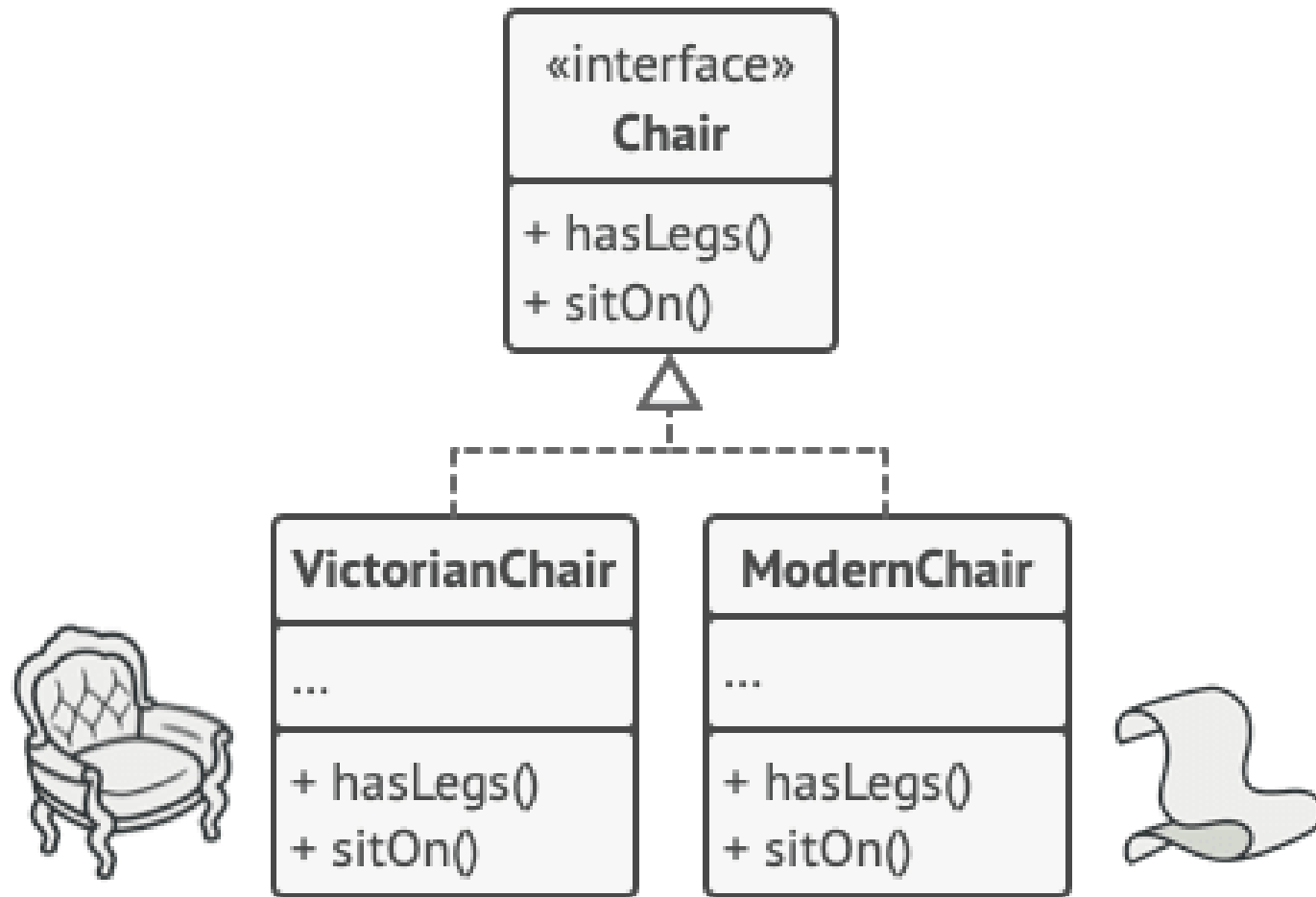
	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

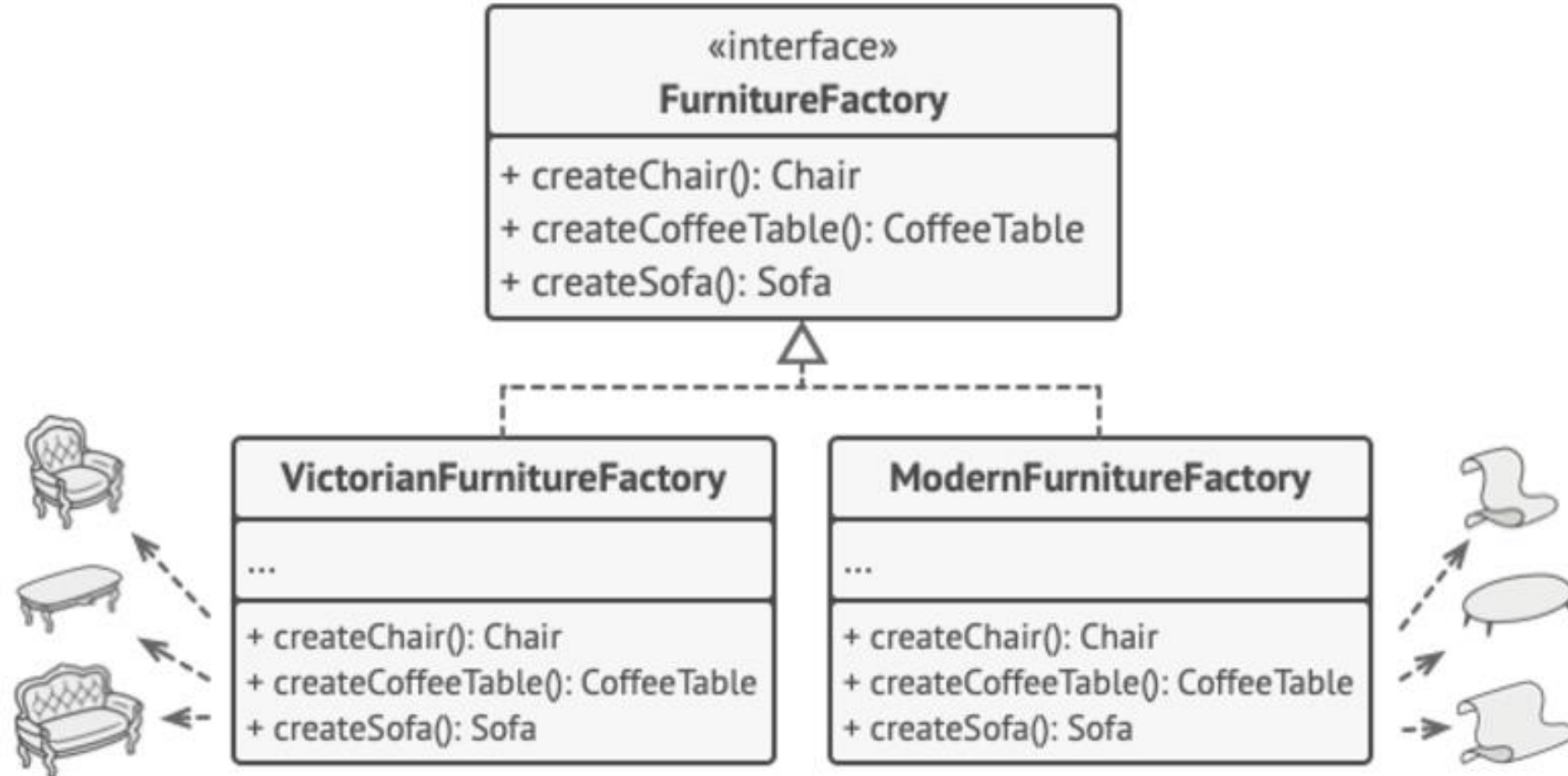
Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.



- The first thing the **Abstract Factory pattern** suggests is to explicitly 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 follow those interfaces.
- For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.



- The next move is to declare the Abstract Factory—an interface with a list of creation methods for all products that are part of the product family (for example, createChair, createSofa and createCoffeeTable).
- These methods must return abstract product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on.





- For each variant of a product family, we create a separate factory class based on the **AbstractFactory** interface.
- A factory is a class that returns products of a particular kind. For example, the **ModernFurnitureFactory** can only create **ModernChair**, **ModernSofa** and **ModernCoffeeTable** objects.
- The client code has to work with both factories and products via their respective abstract interfaces.
- This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

- The client doesn't have to be aware of the factory's class, nor does it matter what kind of chair it gets.
- Whether it's a Modern model or a Victorian-style chair, the client must treat all chairs in the same manner, using the abstract Chair interface.
- With this approach, the only thing that the client knows about the chair is that it implements the sitOn method in some way.
- Also, whichever variant of the chair is returned, it'll always match the type of sofa or coffee table produced by the same factory object.

Steps to Implement Abstract Factory:

1. Define abstract product interfaces for related products.

```
from abc import ABC, abstractmethod
```

```
class Button(ABC):  
    @abstractmethod  
    def render(self):  
        pass
```

```
class Checkbox(ABC):  
    @abstractmethod  
    def render(self):  
        pass
```

2. Create concrete implementations for each product type.

Windows products

```
class WindowsButton(Button):  
    def render(self):  
        print("Rendering a Windows Button.")
```

```
class WindowsCheckbox(Checkbox):  
    def render(self):  
        print("Rendering a Windows Checkbox.")
```

Mac products

```
class MacButton(Button):  
    def render(self):  
        print("Rendering a Mac Button.")
```

```
class MacCheckbox(Checkbox):  
    def render(self):  
        print("Rendering a Mac Checkbox.")
```

3. Define an **abstract factory** with methods to create each product.

```
class GUIFactory(ABC):  
    @abstractmethod  
    def create_button(self):  
        pass  
  
    @abstractmethod  
    def create_checkbox(self):  
        pass
```

4. **Implement concrete factories** that produce related product families.

```
class WindowsFactory(GUIFactory):
```

```
    def create_button(self):  
        return WindowsButton()
```

```
    def create_checkbox(self):  
        return WindowsCheckbox()
```

```
class MacFactory(GUIFactory):
```

```
    def create_button(self):  
        return MacButton()
```

```
    def create_checkbox(self):  
        return MacCheckbox()
```

Client code uses the factory to create products.

```
def client_code(factory: GUIFactory):  
    button = factory.create_button()  
    checkbox = factory.create_checkbox()  
  
    button.render()  
    checkbox.render()
```

```
# Use Windows factory  
windows_factory = WindowsFactory()  
print("Using Windows Factory:")  
client_code(windows_factory)
```

```
# Use Mac factory  
mac_factory = MacFactory()  
print("\nUsing Mac Factory:")  
client_code(mac_factory)
```



- You can be sure that the products you're getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.
- Single Responsibility Principle. You can extract the product creation code into one place, making the code easier to support.
- Open/Closed Principle. You can introduce new variants of products without breaking existing client code.



The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

Creational patterns

- ✓ Factory
- ✓ Abstract factory
- ✓ **Builder**
- ✓ Prototype
- ✓ Singleton

Builder is a creational design pattern that 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.

"It separates object construction logic from its representation" means:

- **Construction logic** (how the object is built, step by step) is handled by a **Builder**.
- **Representation** (the final product's structure or format) is independent of the construction process.

This allows creating different types or versions of an object (representations) using the same building steps.

Example:

A **Builder** for a house might define steps like:

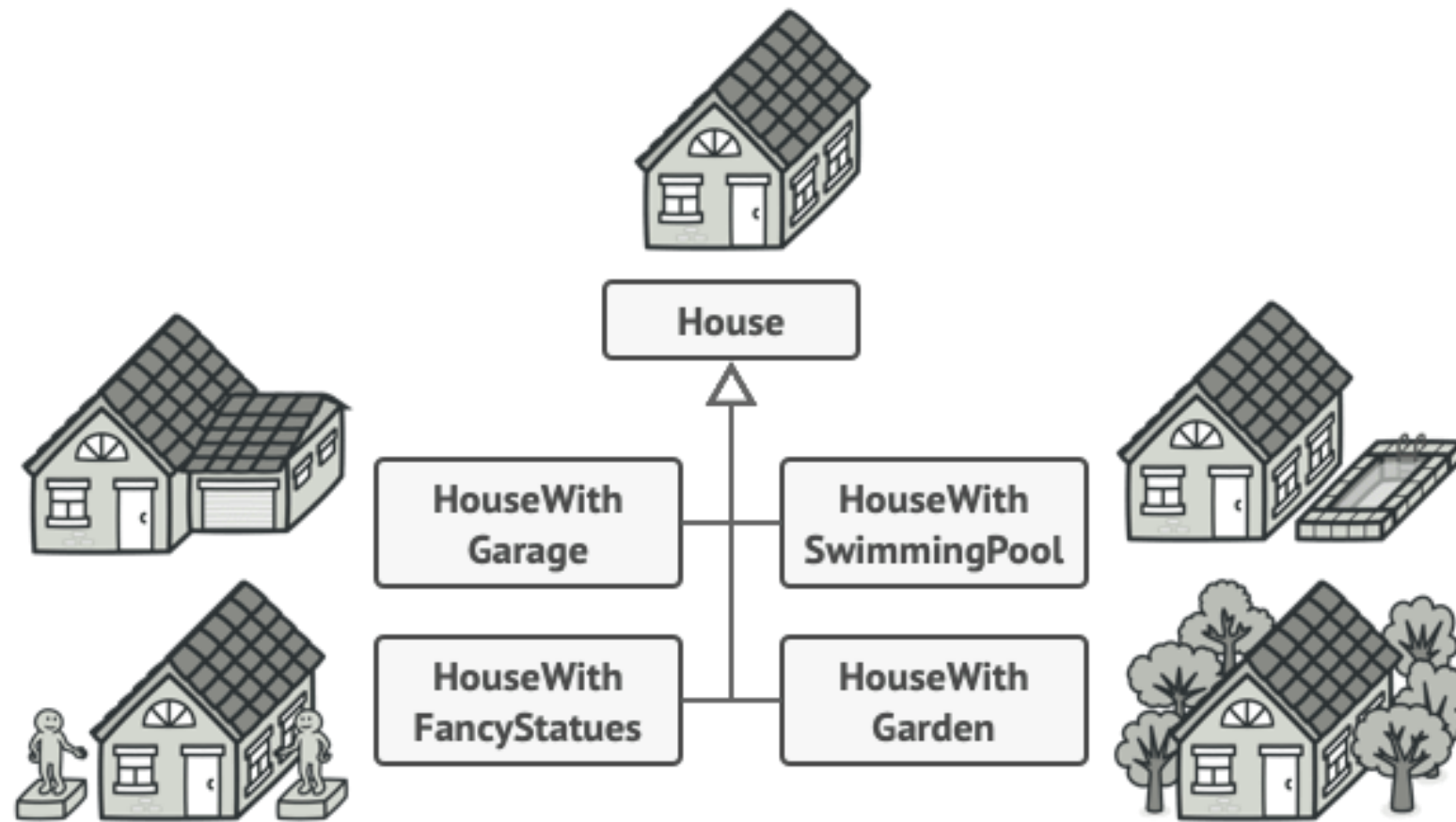
1. Build walls.
2. Add roof.
3. Paint interior.

But the **representation** could differ:

- A **wooden house** or a **brick house** can be built with the same process but use different materials.

Imagine a 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. Or even worse: scattered all over the client code.

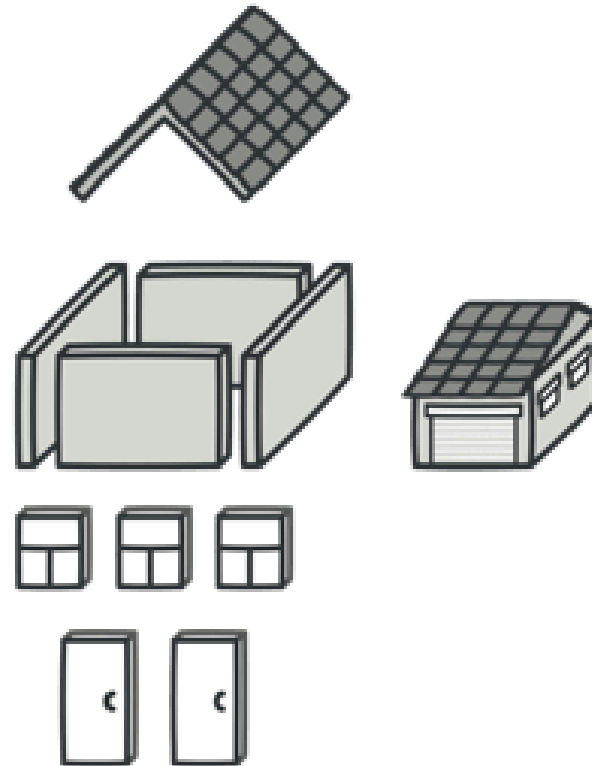
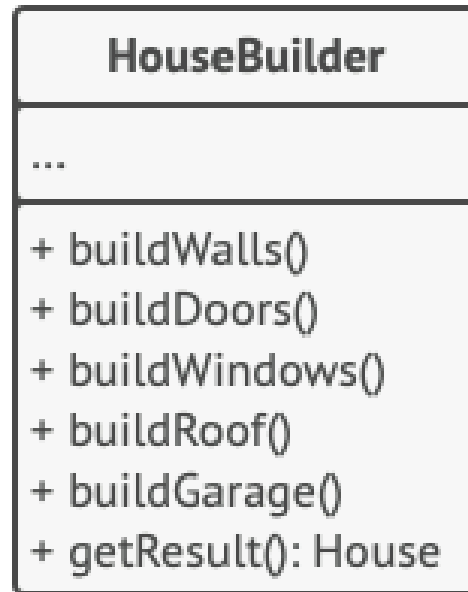


let's think about how to create a **House** object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

The simplest solution is to extend the base **House** class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.



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



You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called **director**.

The ***director*** class defines the order in which to execute the building steps, while the *builder* provides the implementation for those steps.

Step 1: Define the Product (Complex Object)

```
class Computer:
    def __init__(self):
        self.cpu = None
        self.gpu = None
        self.ram = None
        self.storage = None

    def __str__(self):
        return f"Computer(CPU: {self.cpu}, GPU: {self.gpu},
RAM: {self.ram}GB, Storage: {self.storage}GB)"
```

Step 2: Create an Abstract Builder

```
from abc import ABC, abstractmethod
```

```
class ComputerBuilder(ABC):
    @abstractmethod
    def set_cpu(self, cpu):
        pass

    @abstractmethod
    def set_gpu(self, gpu):
        pass

    @abstractmethod
    def set_ram(self, ram):
        pass

    @abstractmethod
    def set_storage(self, storage):
        pass

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

Step 3: Create Concrete Builder

```
class GamingComputerBuilder(ComputerBuilder):  
    def __init__(self):  
        self.computer = Computer()  
  
    def set_cpu(self, cpu):  
        self.computer.cpu = cpu  
  
    def set_gpu(self, gpu):  
        self.computer.gpu = gpu  
  
    def set_ram(self, ram):  
        self.computer.ram = ram  
  
    def set_storage(self, storage):  
        self.computer.storage = storage  
  
    def get_computer(self):  
        return self.computer
```

Step 4: Define the Director

```
class ComputerDirector:  
    def __init__(self, builder):  
        self.builder = builder  
  
    def build_gaming_computer(self):  
        self.builder.set_cpu("Intel i9")  
        self.builder.set_gpu("NVIDIA RTX 4090")  
        self.builder.set_ram(32)  
        self.builder.set_storage(2000)
```

Client code

Create a builder

```
gaming_builder = GamingComputerBuilder()
```

Use the director to construct the product

```
director = ComputerDirector(gaming_builder)
```

```
director.build_gaming_computer()
```

Get the final product

```
gaming_computer = gaming_builder.get_computer()
```

```
print(gaming_computer)
```



- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.



The overall complexity of the code increases since the pattern requires creating multiple new classes.

Creational patterns

- ✓ Factory
- ✓ Abstract factory
- ✓ Builder
- ✓ **Prototype**
- ✓ Singleton

Prototype is a **creational design pattern** that allows objects to be cloned from a prototype instance instead of being created directly using constructors.

Say 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.

Nice! But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.

Since you have to know the object's class to create a duplicate, your code becomes dependent on that class.



- 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.

How It Works:

- Define a clone() method in your base class or interface.
- Implement clone() in concrete classes to duplicate objects.
- Use the prototype instance to create copies instead of new.

Define a Prototype Base Class

```
import copy
```

```
class Prototype:  
    def clone(self):  
        return copy.deepcopy(self)
```

Step 2: Create Concrete Prototype Classes

```
class Car(Prototype):  
    def __init__(self, model, color, engine):  
        self.model = model  
        self.color = color  
        self.engine = engine  
  
    def __str__(self):  
        return f"Car(Model: {self.model}, Color: {self.color}, Engine: {self.engine})"
```

Step 3: Use the Prototype

Create a prototype object

```
original_car = Car("Sedan", "Red", "V6")
```

Clone the prototype

```
cloned_car = original_car.clone()
```

Modify the clone

```
cloned_car.color = "Blue"
```

Display the results

```
print(original_car) # Output: Car(Model: Sedan, Color: Red, Engine: V6)
```

```
print(cloned_car)  # Output: Car(Model: Sedan, Color: Blue, Engine: V6)
```



- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.
- You get an alternative to inheritance when dealing with configuration presets for complex objects.



Cloning complex objects that have circular references might be very tricky.

Creational patterns

- ✓ Factory
- ✓ Abstract factory
- ✓ Builder
- ✓ Prototype
- ✓ **Singleton**

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

<<already covered>>

That's it