

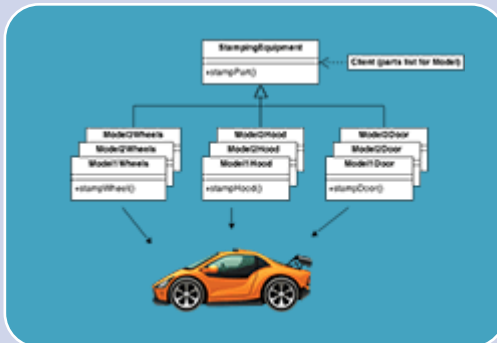
# Creational Design Patterns

Kuan-Ting Lai  
2020/4/6

# Design Patterns

## Creational Design Patterns

Initialize objects  
or create new  
classes



## Structural Design Patterns

Compose  
objects to get  
new functions



## Behavioral Design Patterns

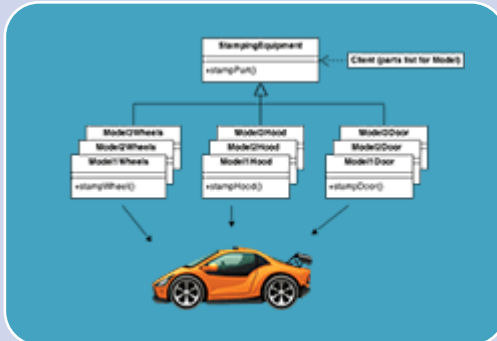
Communication  
between objects



# Creational Design Patterns

## Creational Design Patterns

Initialize objects  
or create new  
classes



## Structural Design Patterns

Compose  
objects to get  
new functions



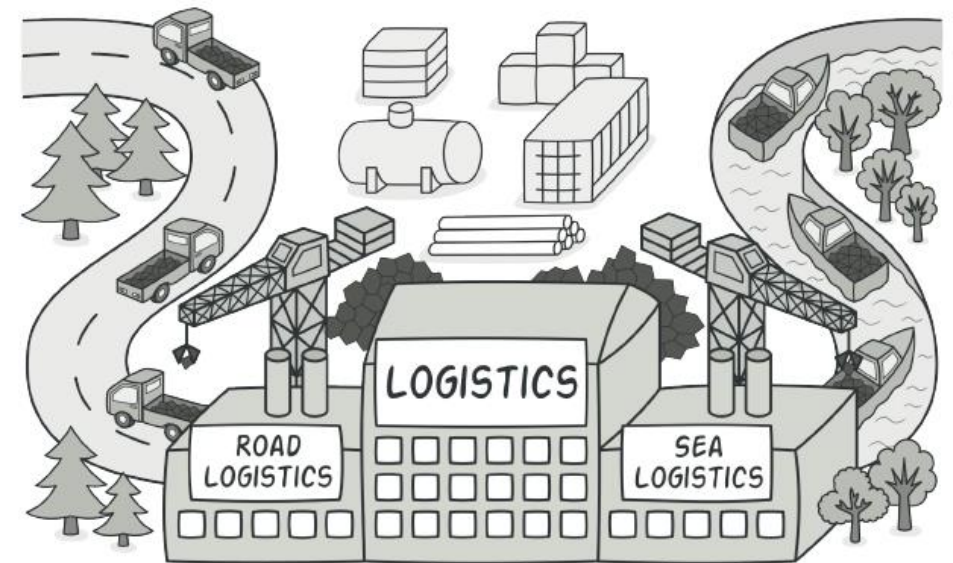
## Behavioral Design Patterns

Communication  
between objects



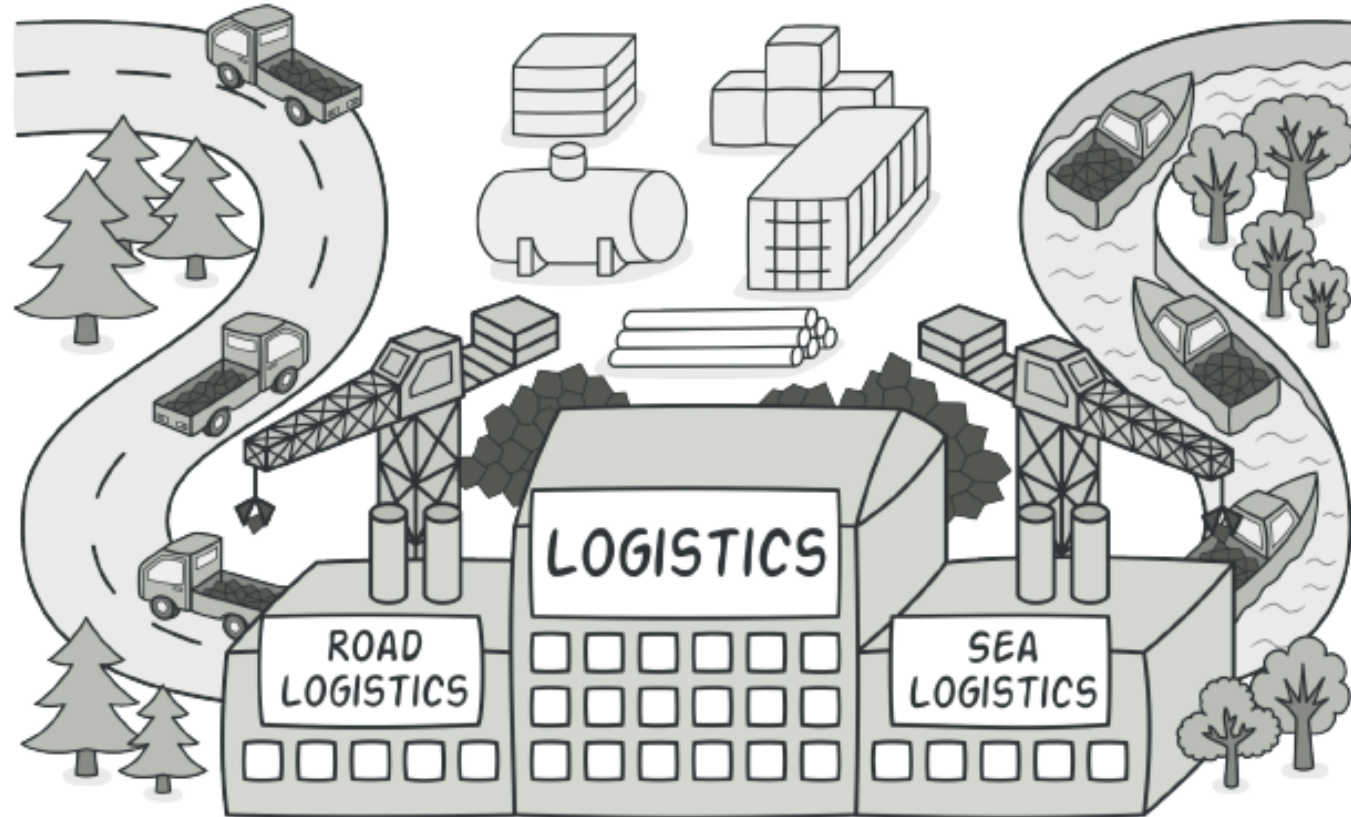
# Creational Design Patterns

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



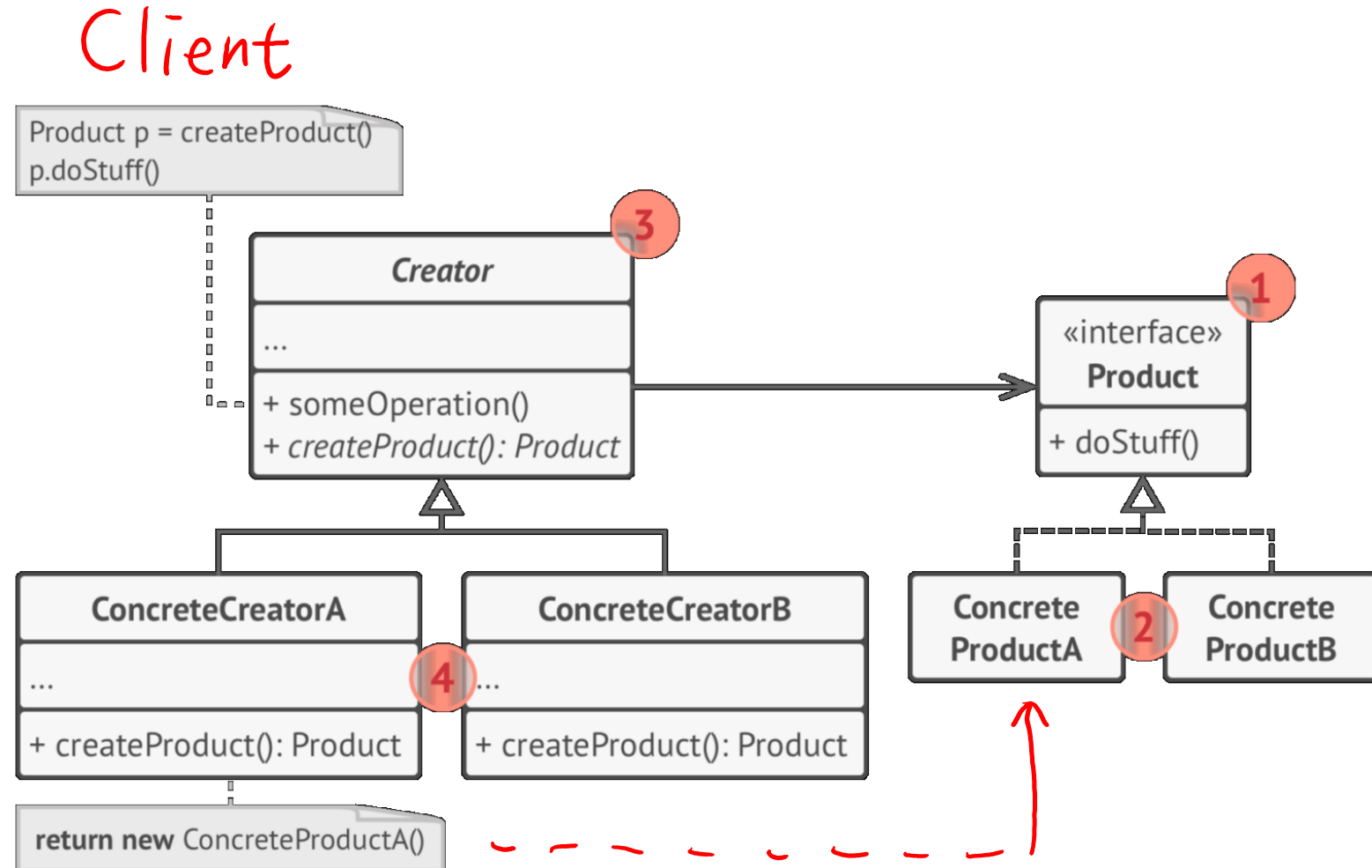
# 1. Factory Method

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



# Structure of Factory Method

1. **Product** declares the interface.
2. **Concrete Products** implements the product interface.
3. **Creator** declares factory method
4. **Concrete Creators** override the base factory method



# Example: Shape Factory

```
public interface Shape { void draw(); }

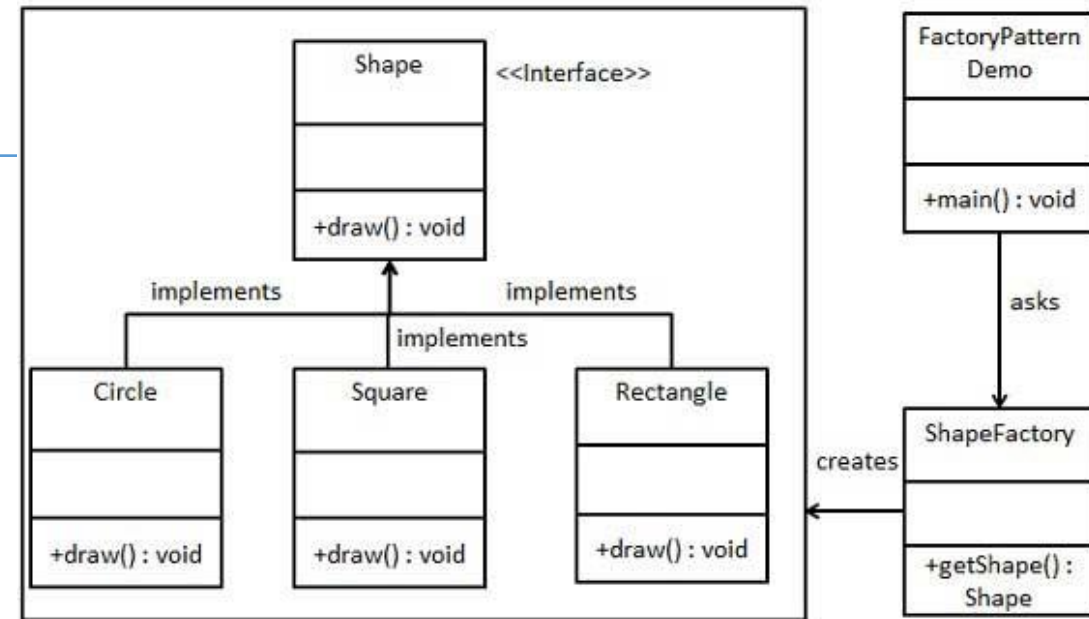
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

public class Square implements Shape {
    @Override
    public void draw() { System.out.println("Inside Square::draw() method."); }
}

public class ShapeFactory {
    //use getShape method to get object of type shape
    public Shape getShape(String shapeType){
        if(shapeType == null){ return null; }

        if( shapeType.equalsIgnoreCase("CIRCLE") ){ return new Circle(); }
        else if( shapeType.equalsIgnoreCase("SQUARE") ){ return new Square(); }

        return null;
    }
}
```



# Using ShapeFactory

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        //get an object of Circle and call its draw method.  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
        shape2.draw();  
    }  
}
```





# C++ Factory Pattern: Creating Vehicles

```
enum VehicleType {VT_TwoWheeler, VT_ThreeWheeler};  
// Library classes  
class Vehicle {  
public:  
    virtual void printVehicle() = 0;  
    static Vehicle* Create(VehicleType type);  
};  
class TwoWheeler : public Vehicle {  
public:  
    void printVehicle() {cout << "I am two wheeler" << endl;}  
};  
class ThreeWheeler : public Vehicle {  
public:  
    void printVehicle() { cout << "I am three wheeler" << endl;}  
};  
// Factory method to create objects of different types.  
Vehicle* Vehicle::Create(VehicleType type) {  
    if (type == VT_TwoWheeler)  
        return new TwoWheeler();  
    else if (type == VT_ThreeWheeler)  
        return new ThreeWheeler();  
    else return NULL;  
}
```

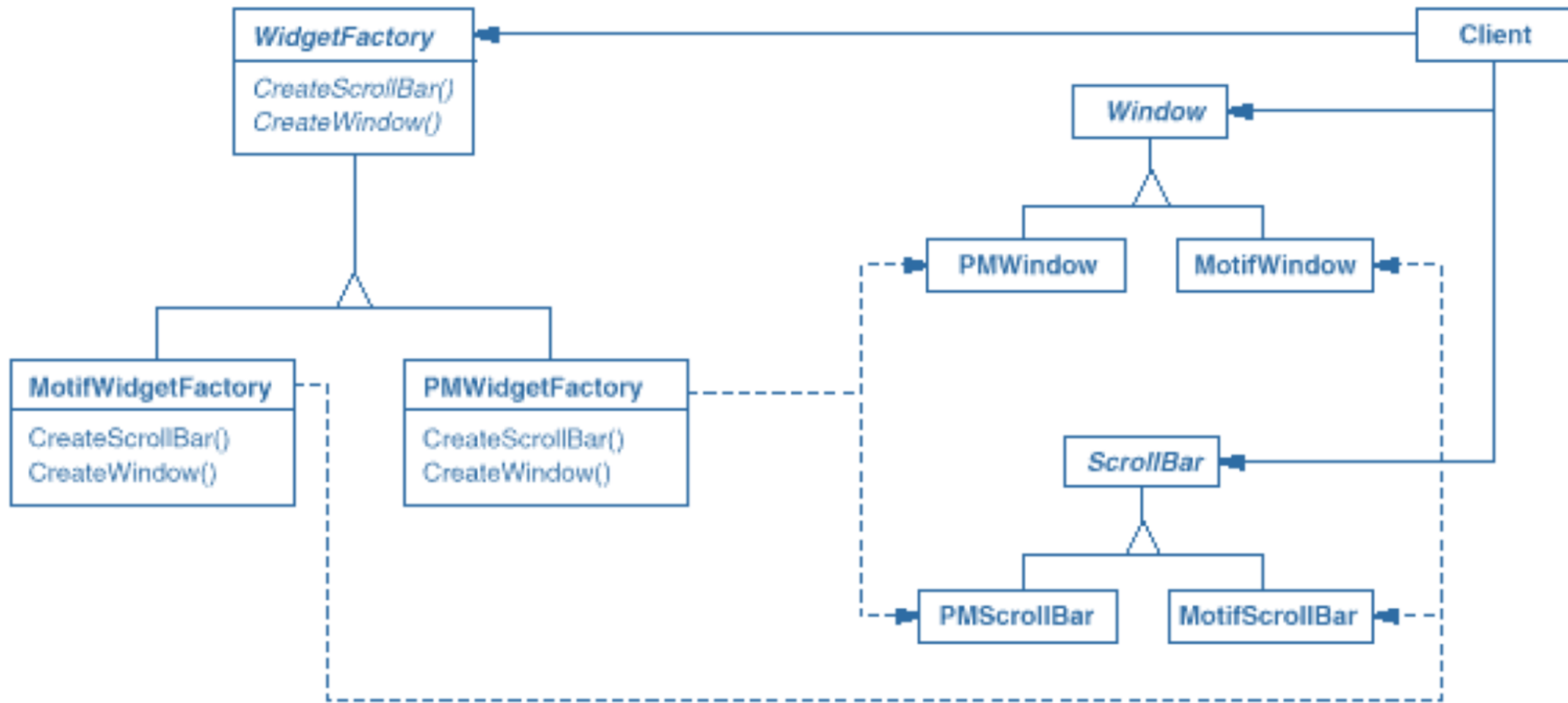


## 2. Abstract Factory Pattern

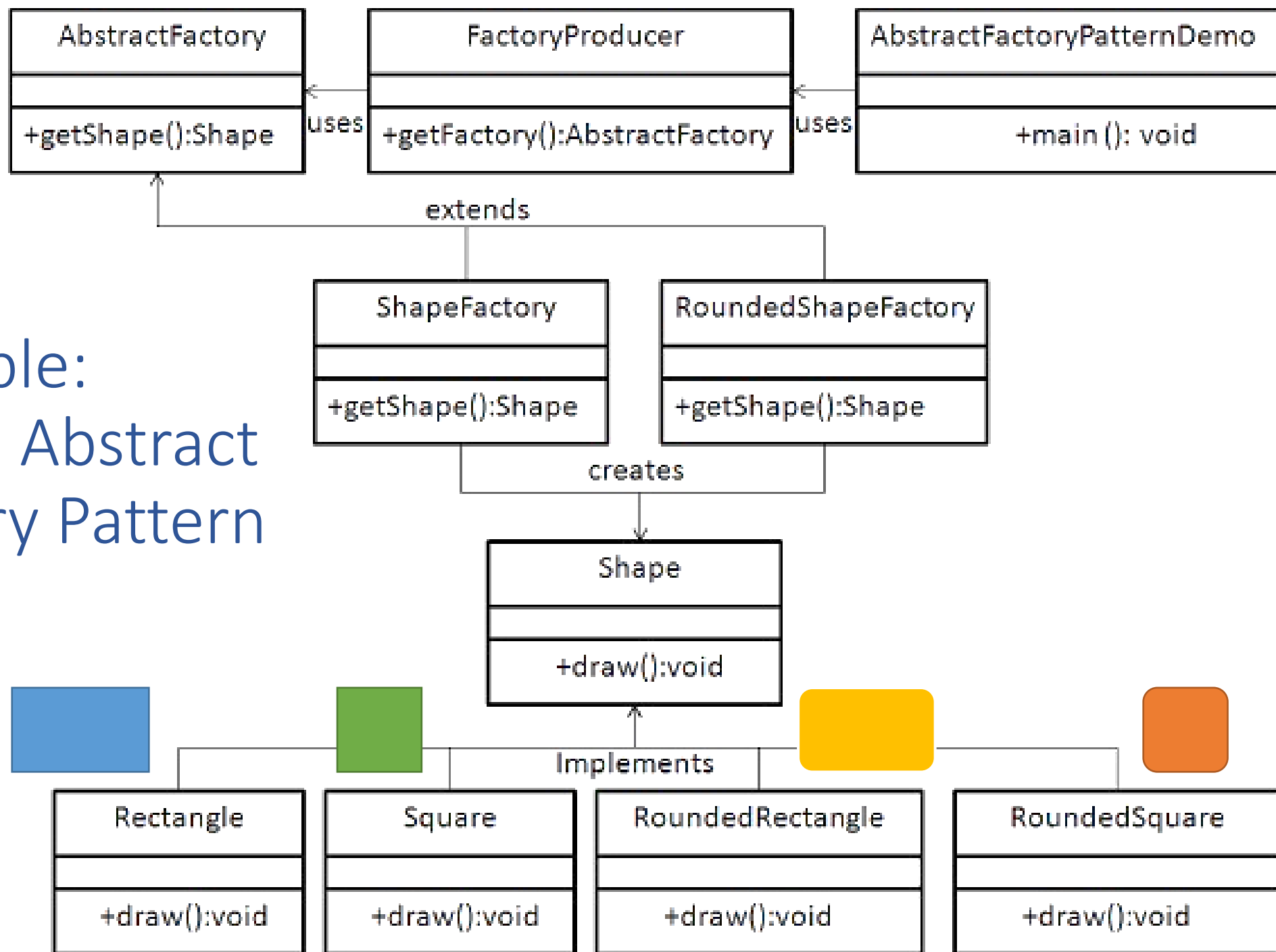
- Just define an interface (abstract class) for creating families of related objects, but doesn't specify their concrete sub-classes
- Abstract factory is also called as factory of factories



# Structure of Abstract Factory



## Example: Shape Abstract Factory Pattern



```
public class FactoryProducer {
    public static AbstractFactory getFactory(boolean rounded) {
        if (rounded) {
            return new RoundedShapeFactory();
        }
        else {
            return new ShapeFactory();
        }
    }
}

public abstract class AbstractFactory {
    abstract Shape getShape(String shapeType);
}

// Extend Abstract Factory
public class RoundedShapeFactory extends AbstractFactory {
    @Override
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new RoundedRectangle();
        }
        else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new RoundedSquare();
        }
        return null;
    }
}
```

.....

[https://www.tutorialspoint.com/design\\_pattern/abstract\\_factory\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm)



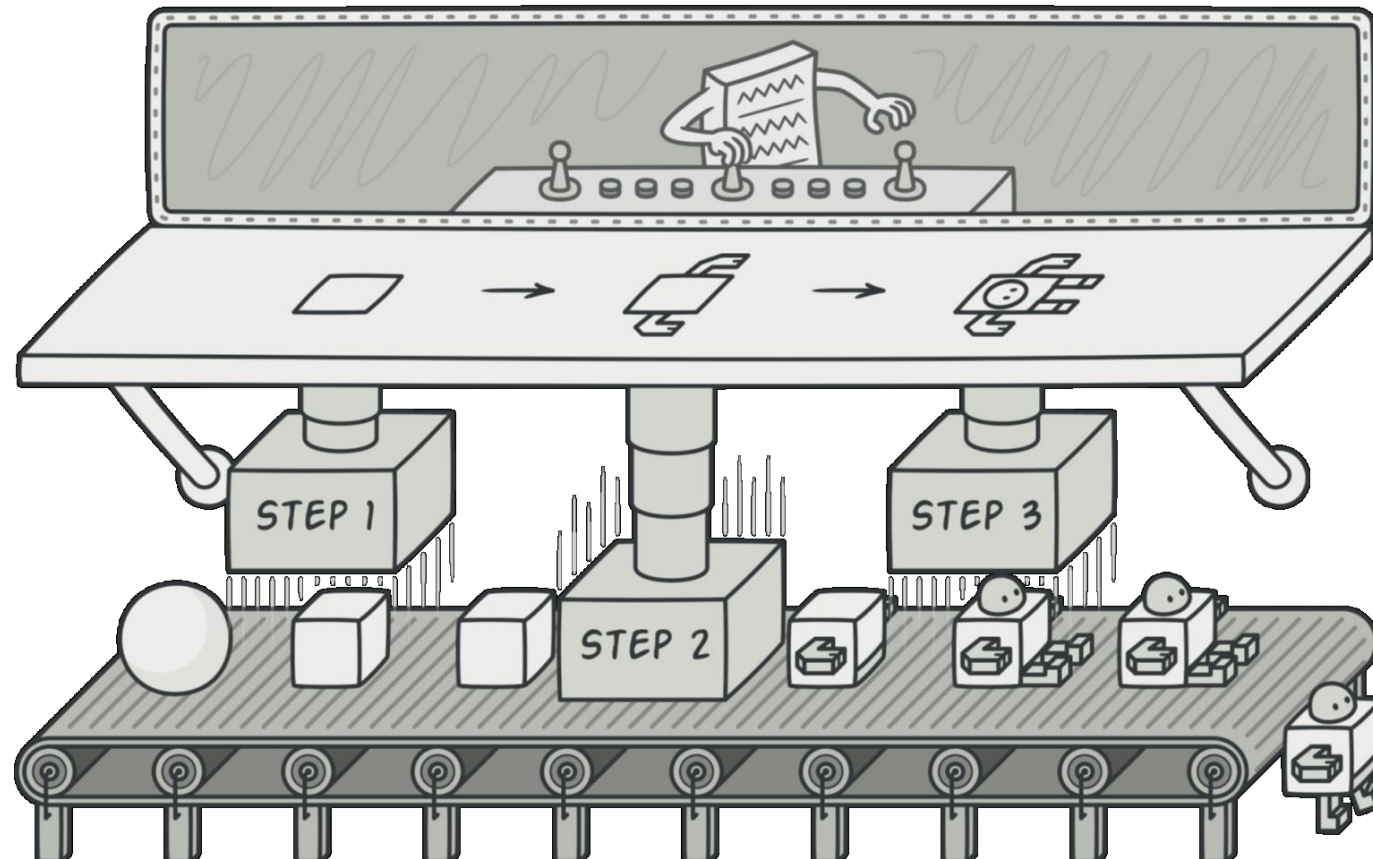
# Using AbstractFactory to Get Rounded Shape

```
public class AbstractFactoryPatternDemo {  
  
    public static void main(String[] args) {  
        //get shape factory  
        AbstractFactory shapeFactory = FactoryProducer.getFactory(false);  
        Shape shape1 = shapeFactory.getShape("RECTANGLE");  
        shape1.draw();  
  
        //get rounded shape factory  
        AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);  
        //get rounded rectangle  
        Shape shape2 = shapeFactory1.getShape("RECTANGLE");  
        shape2.draw();  
    }  
}
```

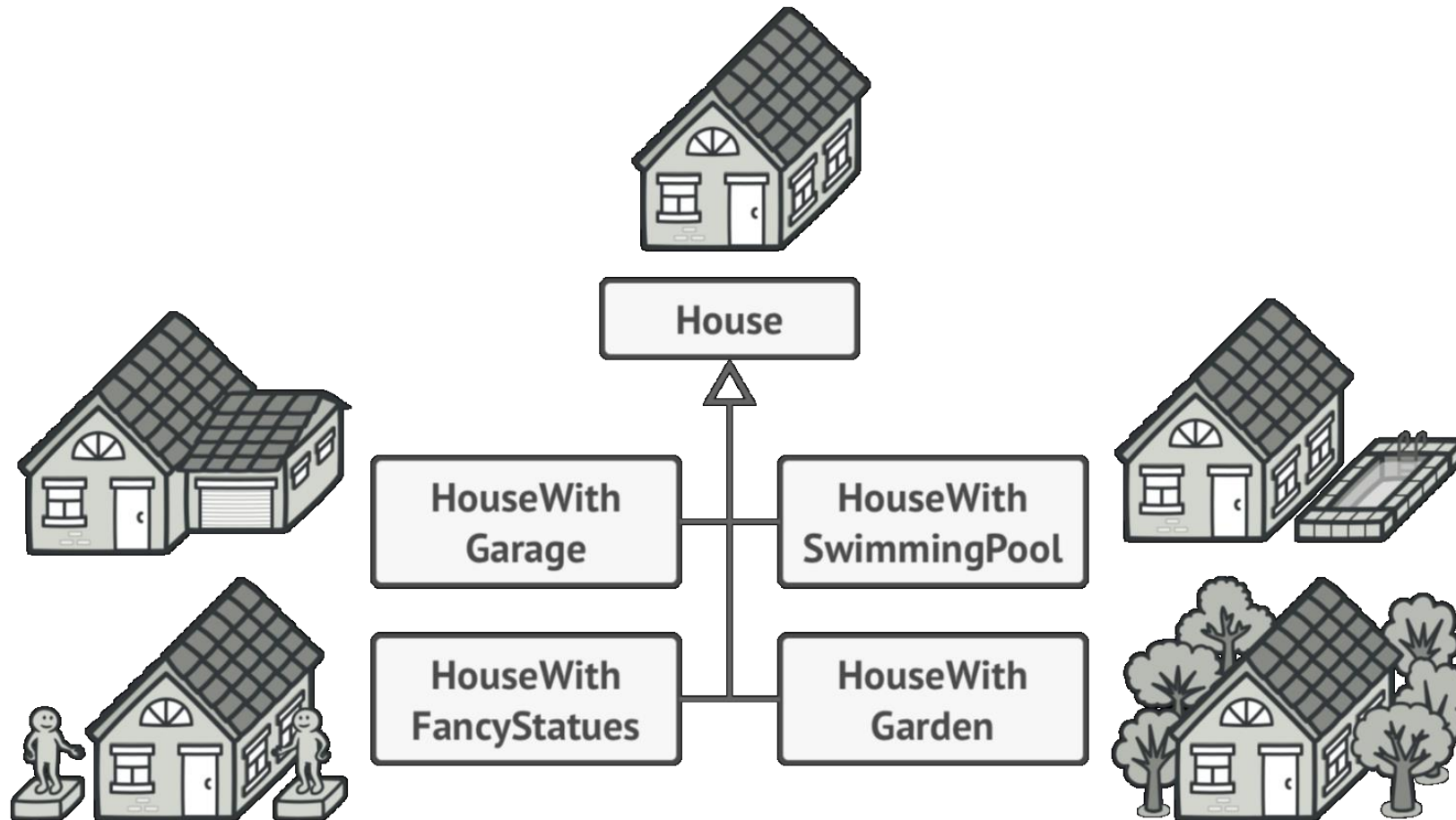


### 3. Builder

- Construct complex objects step by step.

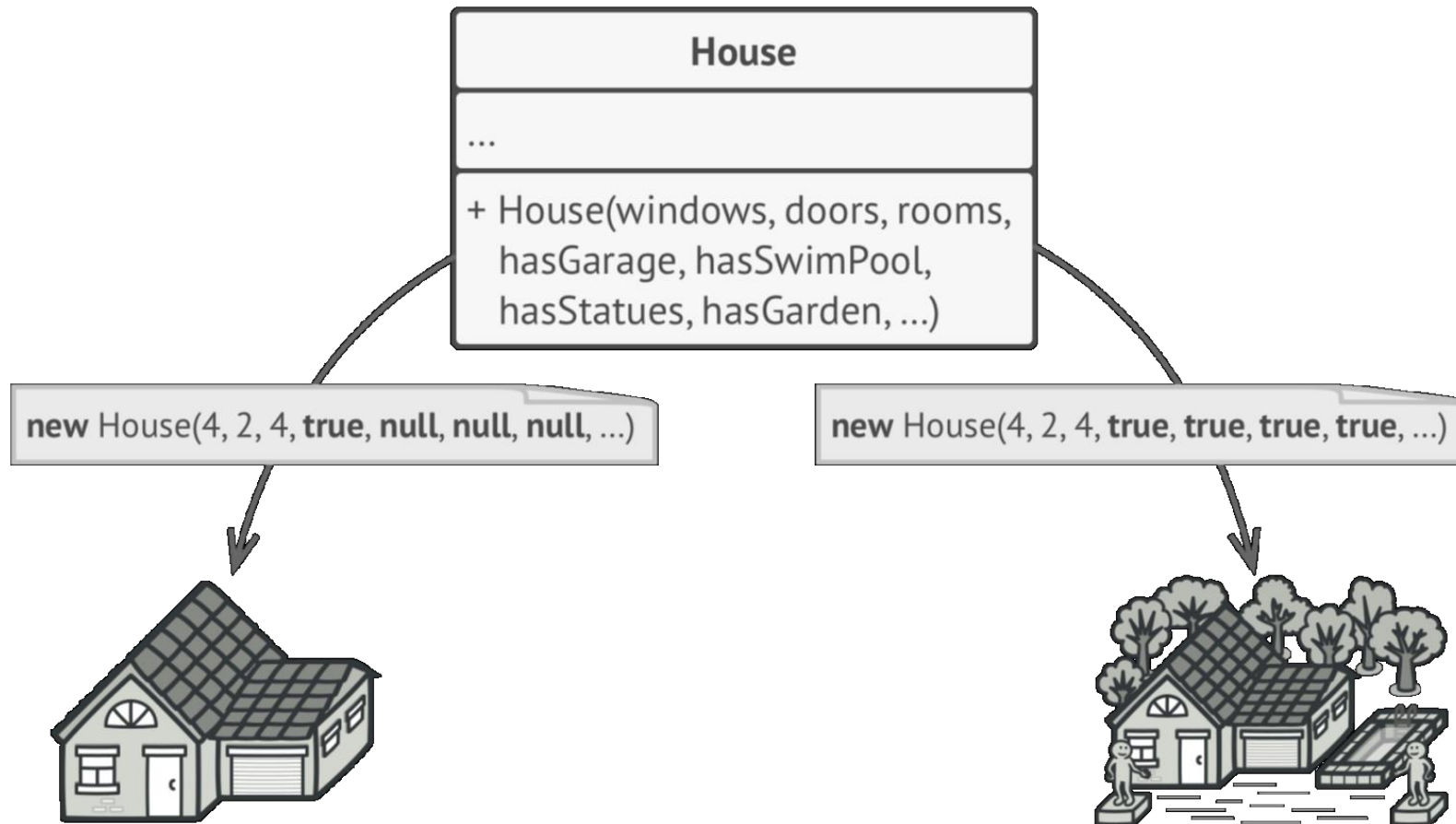


## Example: Building a House

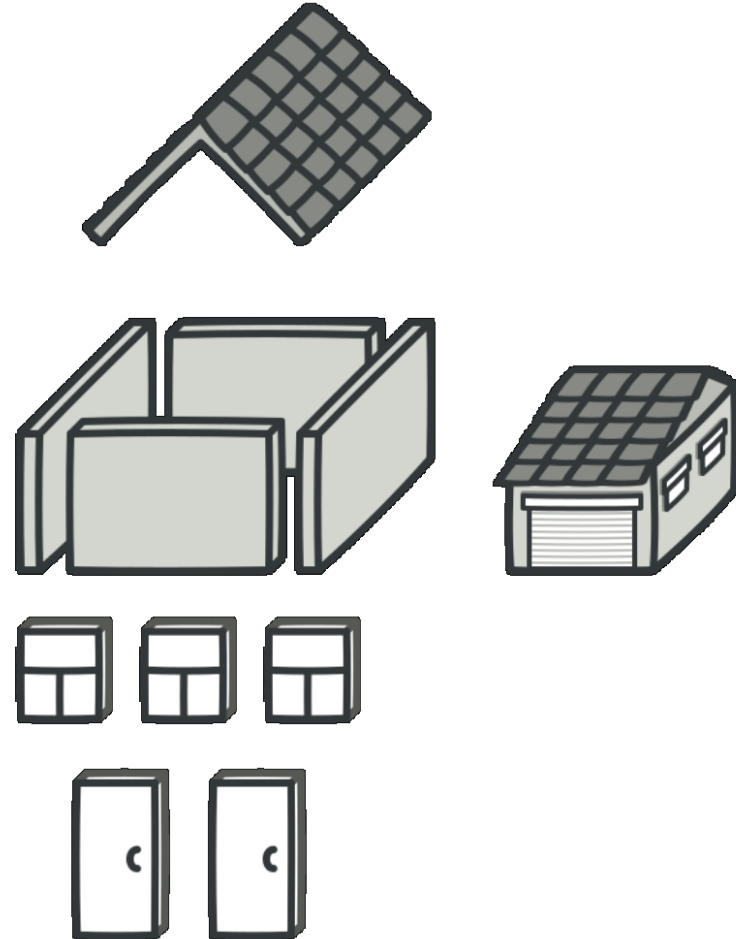
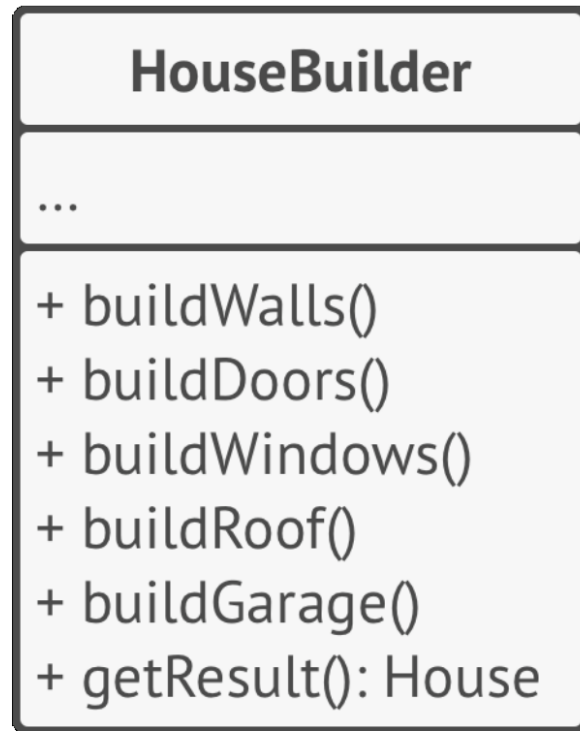




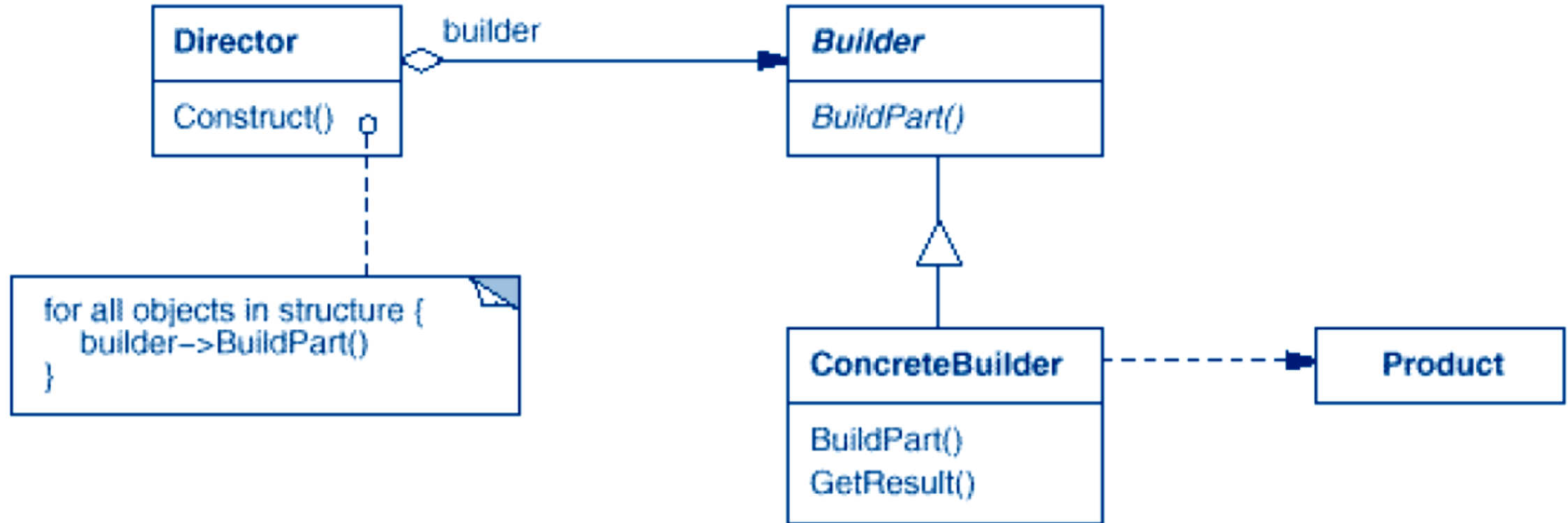
# Solution 1: constructors with many parameters?

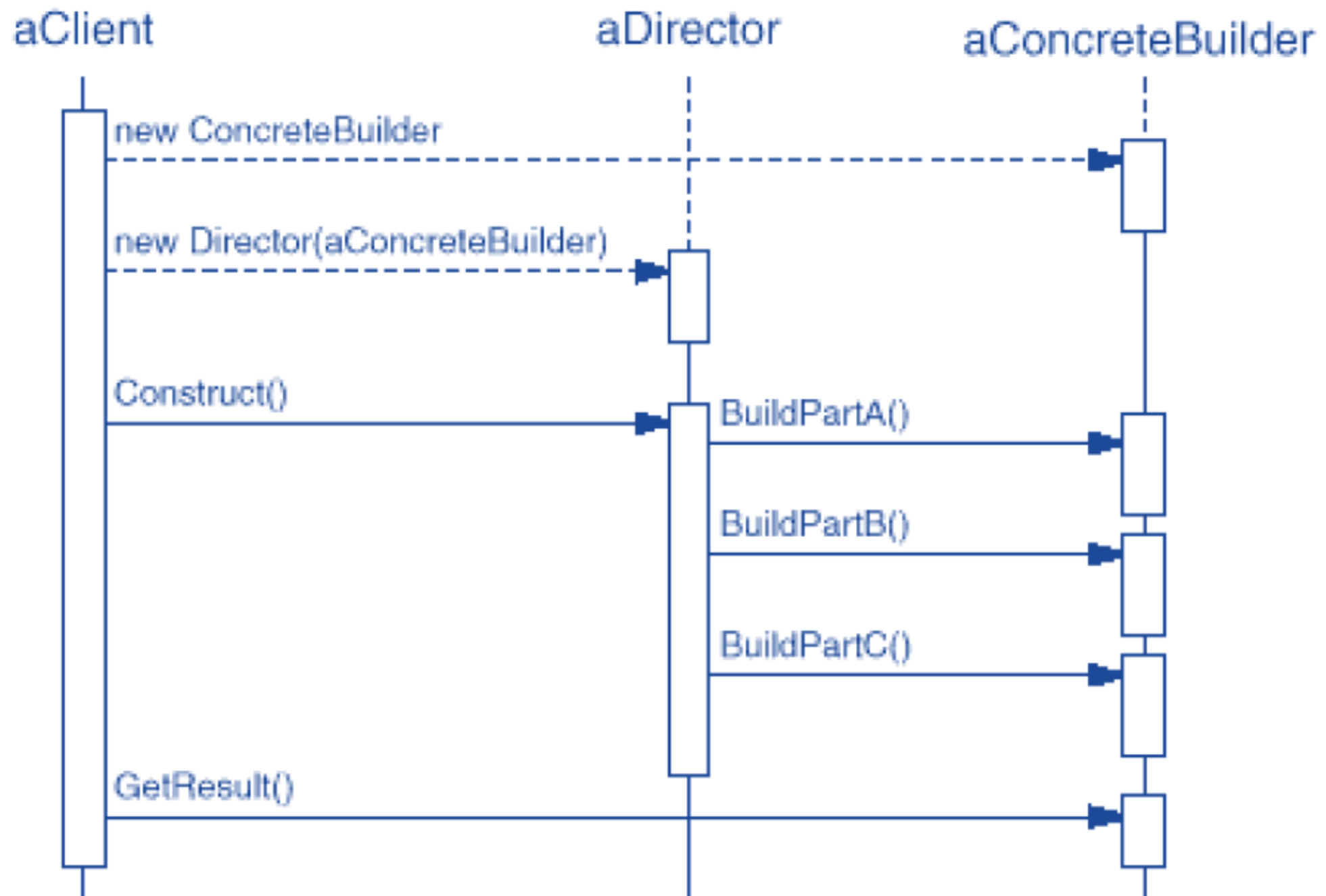


# Solution 2: Using Builder Design Pattern

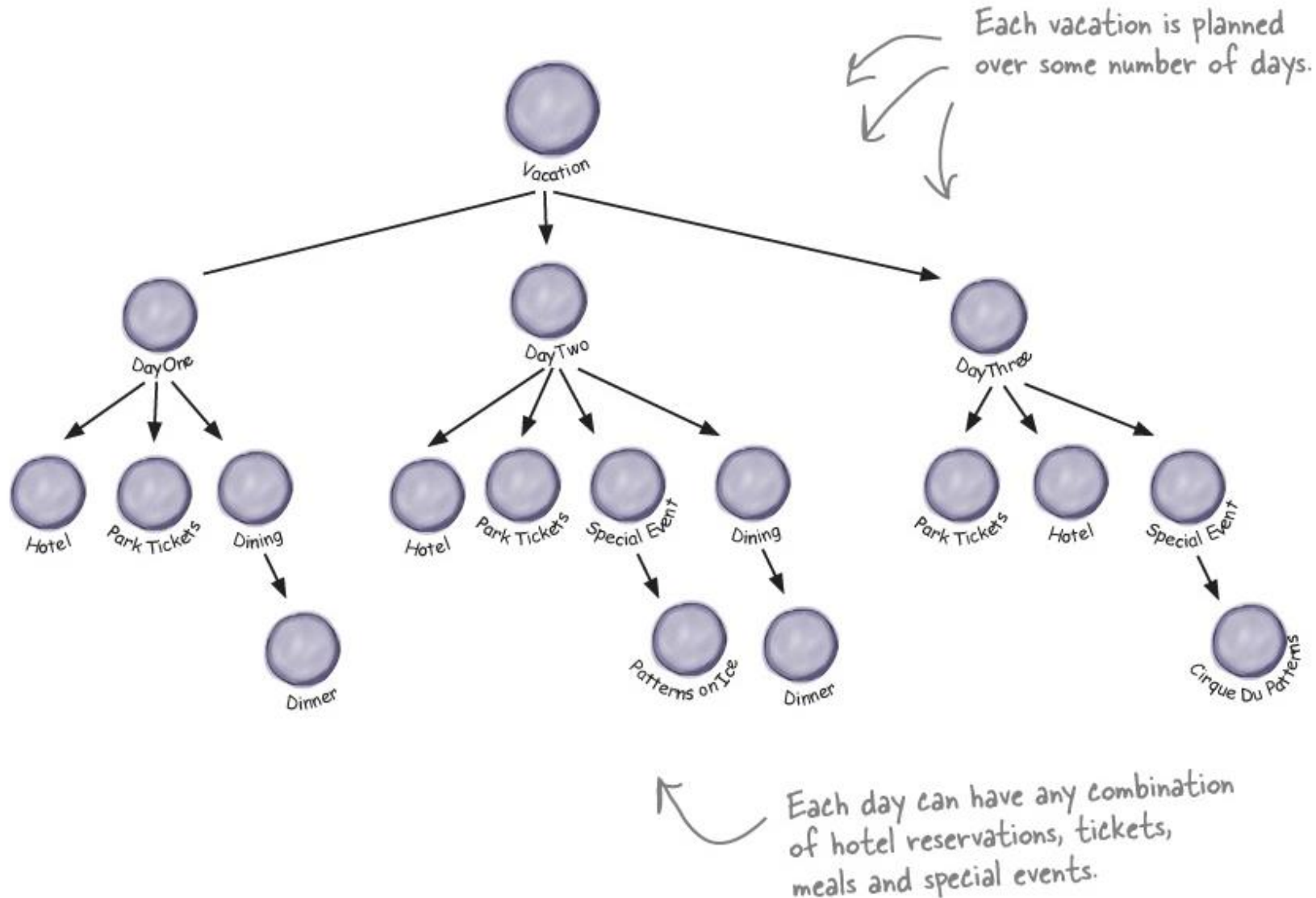


# Structure of Builder

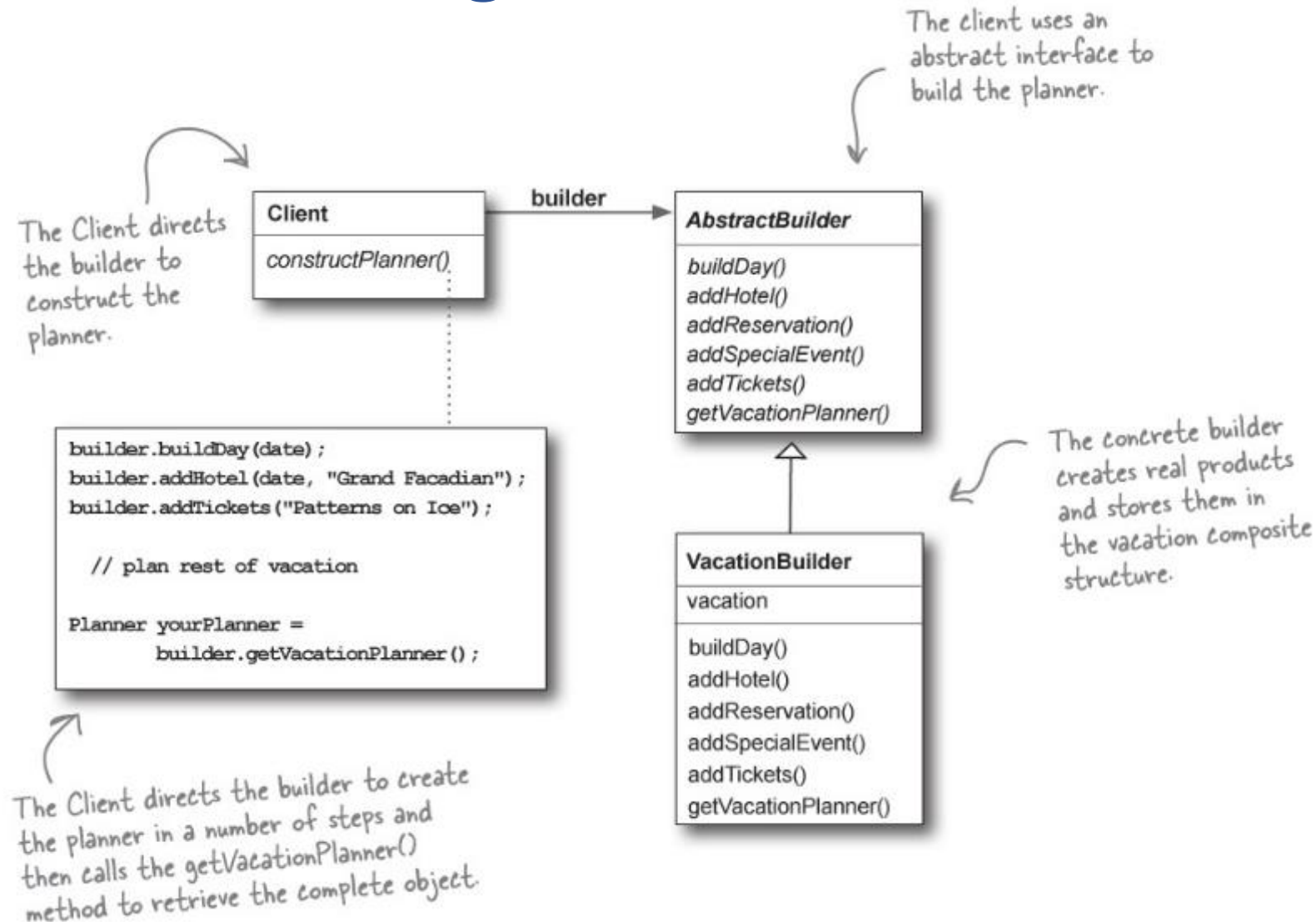




# Example 2: Building a Vacation Planner

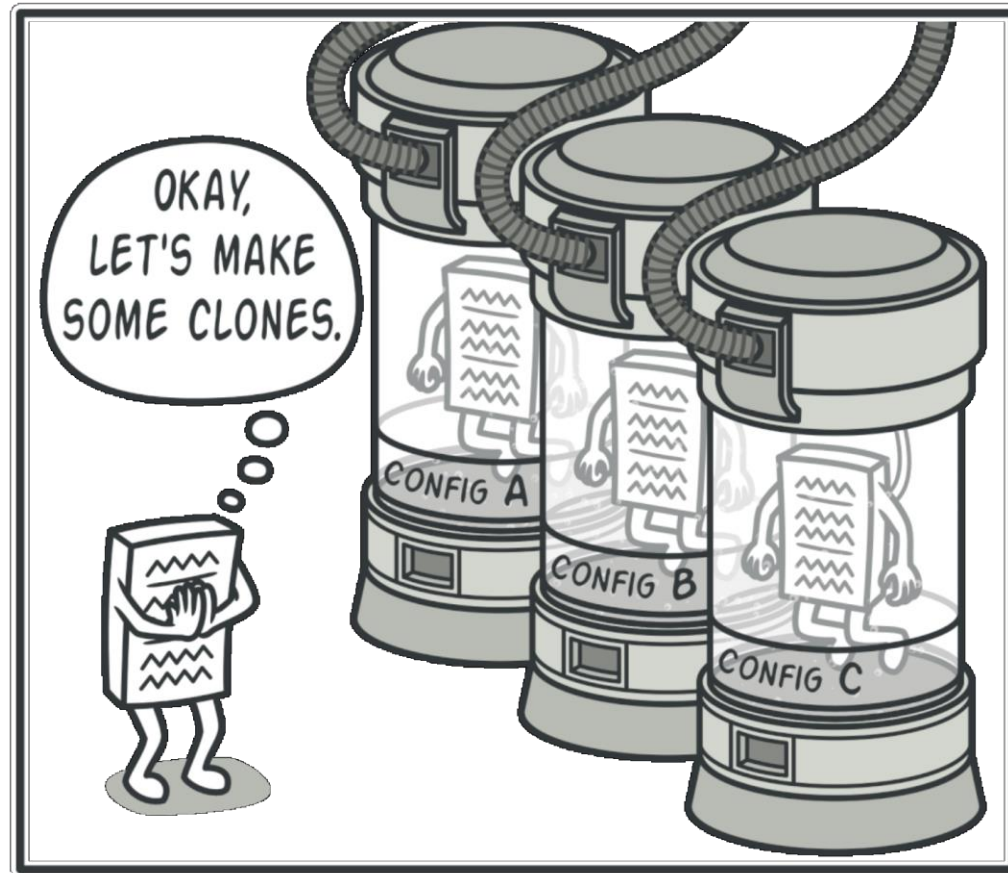


# Example 2: Building a Vacation Planner



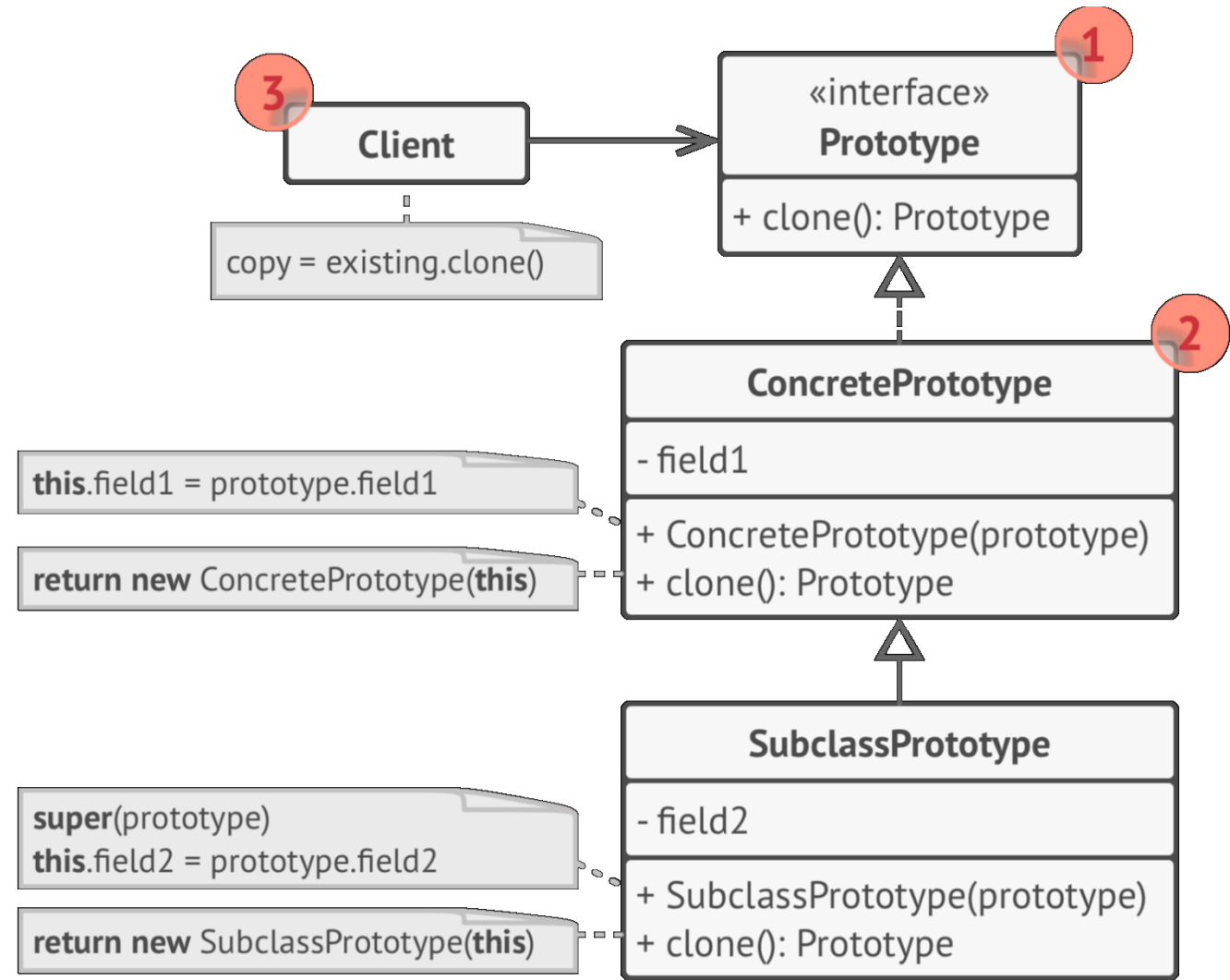
## 4. Prototype

- Allow copying existing objects without making your code dependent on their classes



# Prototype Structure

1. Define clone() method.
2. **Concrete Prototype** class implements the cloning method.
3. The **Client** can produce a copy of any object that follows the prototype interface.



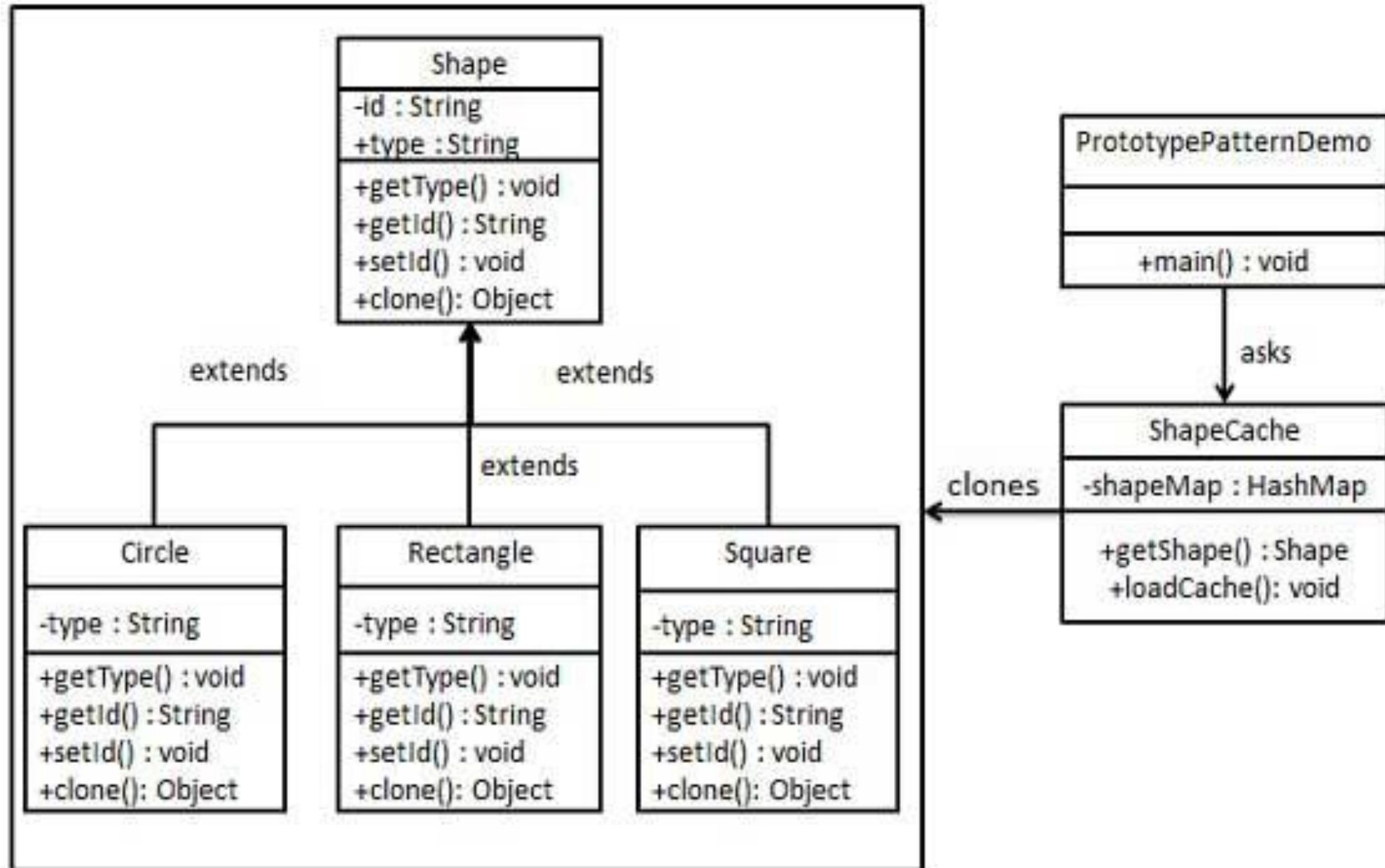


# Java Cloneable Interface

- A class implements the Cloneable interface to indicate that Object.clone() can be used to make a field-for-field copy of instances

```
public class DogName implements Cloneable {
    public String dname;
    // Overriding clone() method of Object class
    public Object clone()throws CloneNotSupportedException {
        return (DogName)super.clone();
    }
    public static void main(String[] args) {
        DogName obj1 = new DogName("Tommy");
        try {
            DogName obj2 = (DogName)obj1.clone();
            System.out.println(obj2.getName());
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

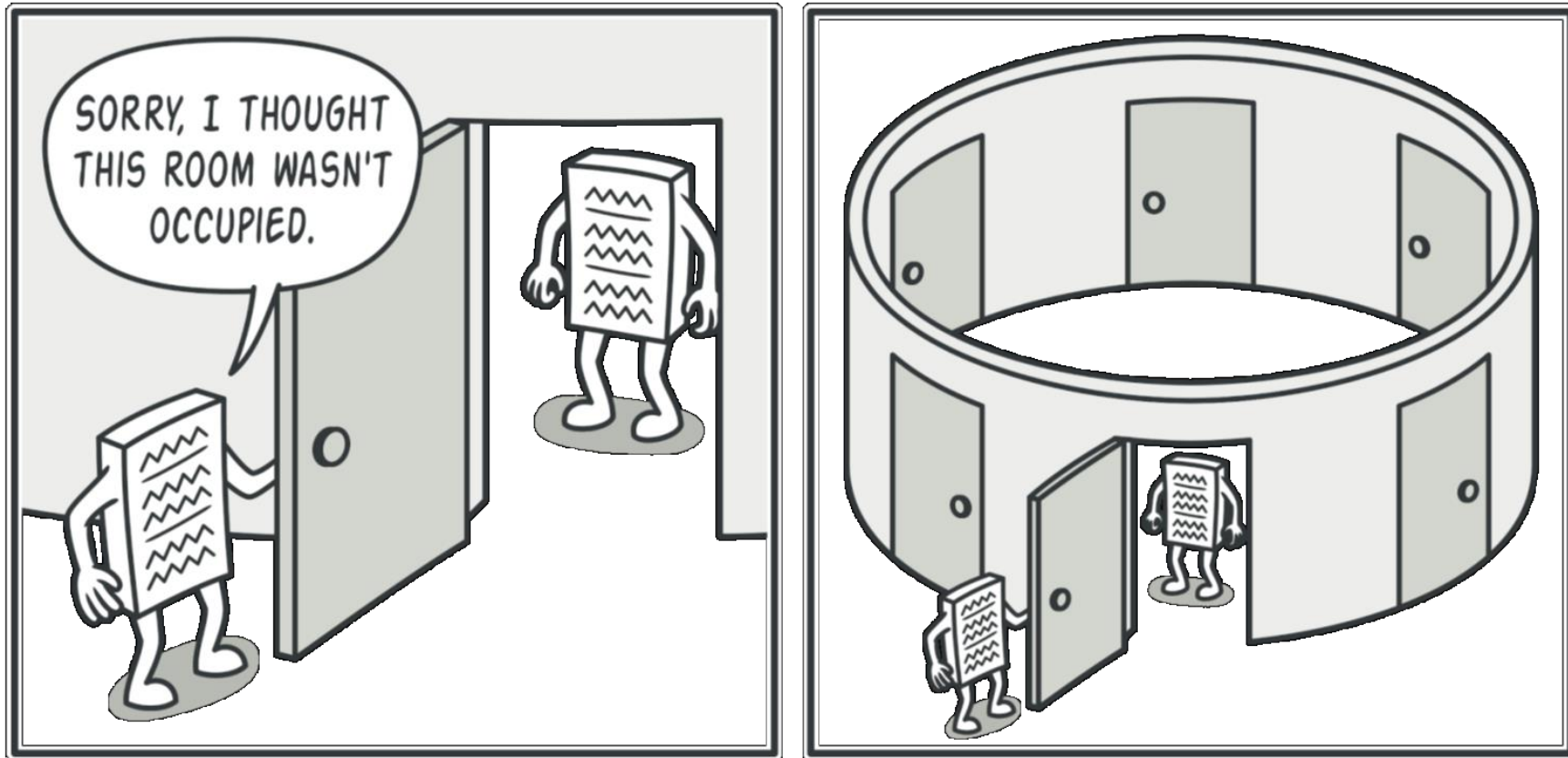
# Example: Shape Prototype



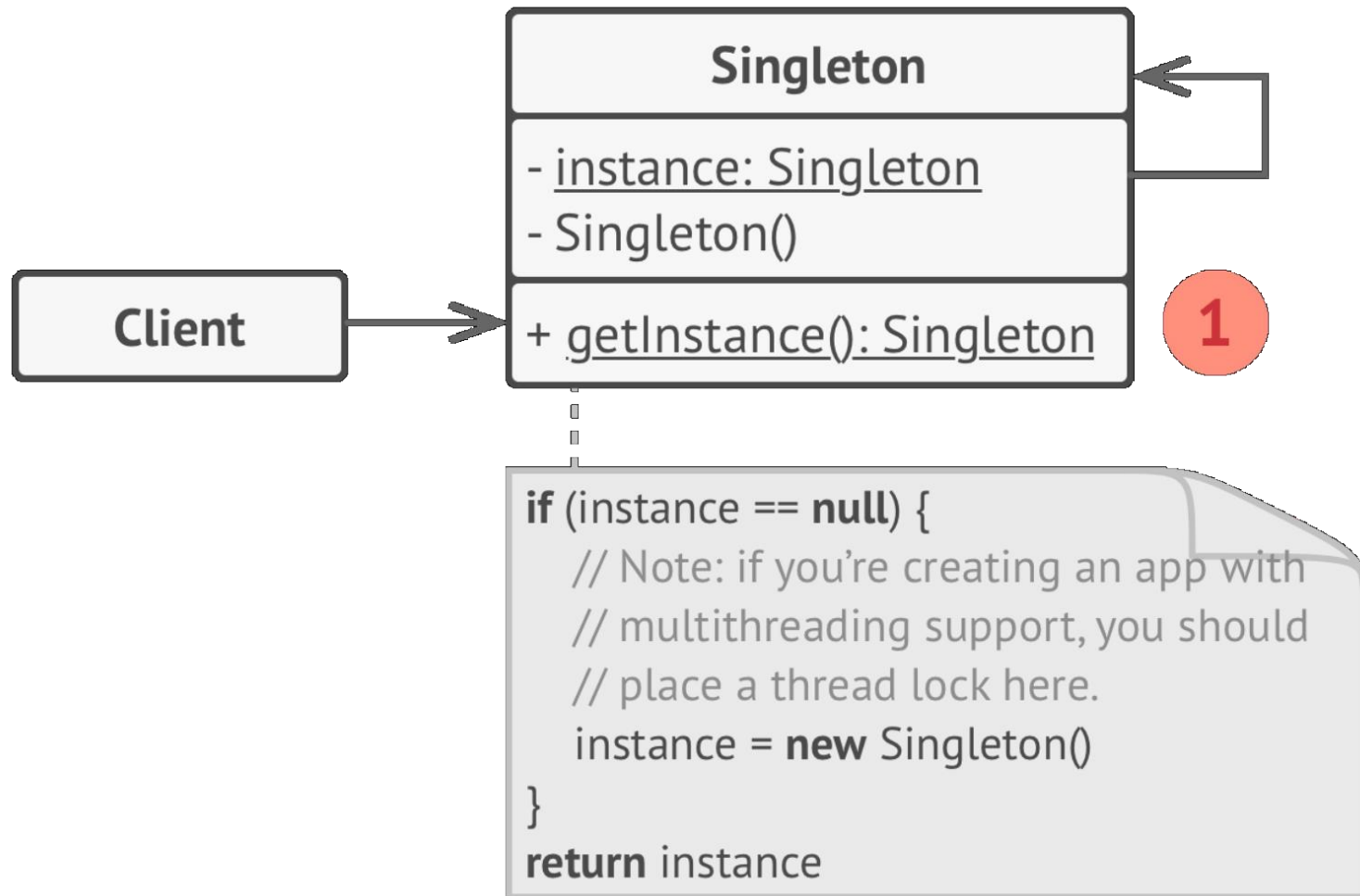
```
public abstract class Shape implements Cloneable {  
    private String id;  
    protected String type;  
  
    abstract void draw();  
  
    public String getType() { return type; }  
    public String getId() { return id; }  
    public void setId(String id) { this.id = id; }  
  
    public Object clone() {  
        Object clone = null;  
  
        try {  
            clone = super.clone();  
        }  
        catch (CloneNotSupportedException e) {  
            e.printStackTrace();  
        }  
        return clone;  
    }  
}
```

# Singleton

- **Singleton** is a creational design pattern that lets you ensure that a class has only one instance



## 5. Singleton Structure



# C++ Singleton Example

```
class Singleton
{
public:
    static Singleton* getInstance();

private:
    static Singleton* instance; // Here will be the instance stored.

    Singleton(); // Private constructor to prevent instantcing.
};

/* Null, because instance will be initialized on demand. */
Singleton* Singleton::instance = 0;

Singleton* Singleton::getInstance()
{
    if (instance == 0)
        instance = new Singleton();

    return instance;
}
```

# Java Singleton Example

```
public class SingleObject {  
  
    //create an object of SingleObject  
    private static SingleObject instance = new SingleObject();  
  
    //make the constructor private so that this class cannot be  
    //instantiated  
    private SingleObject() {}  
  
    //Get the only object available  
    public static SingleObject getInstance() {  
        return instance;  
    }  
  
    public void showMessage() {  
        System.out.println("Hello World!");  
    }  
}
```

# Summary

- Many designs start by using **Factory Method** and evolve toward **Abstract Factory**, **Prototype**, or **Builder**
- **Builder** focuses on constructing complex objects step by step.
- **Abstract Factory** specializes in creating families of related objects.
- **Prototype** is used to clone (copy) objects
- **Singleton** ensures that a class has only one instance



# Dive into Design Patterns

- Alexander Shvets



# References

- Alexander Shvets, “Dive into Design Patterns,” 2018
- [https://www.tutorialspoint.com/design\\_pattern/index.htm](https://www.tutorialspoint.com/design_pattern/index.htm)
- <https://www.javatpoint.com/design-patterns-in-java>