What is Design Patterns

- **Design patterns** are typical solutions to commonly occurring problems in software design.
- They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.
- A design pattern isn't a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.

Design Patterns

- **Patterns deal with**
  - Application and system design
  - Abstractions on top of code
  - Speed up the development process by providing tested, proven development paradigms.
  - Relationships between classes and other collaborators
  - Problems that have been already solved
- **Patterns are not concerned with**
  - Algorithms
  - Specific implementations or classes

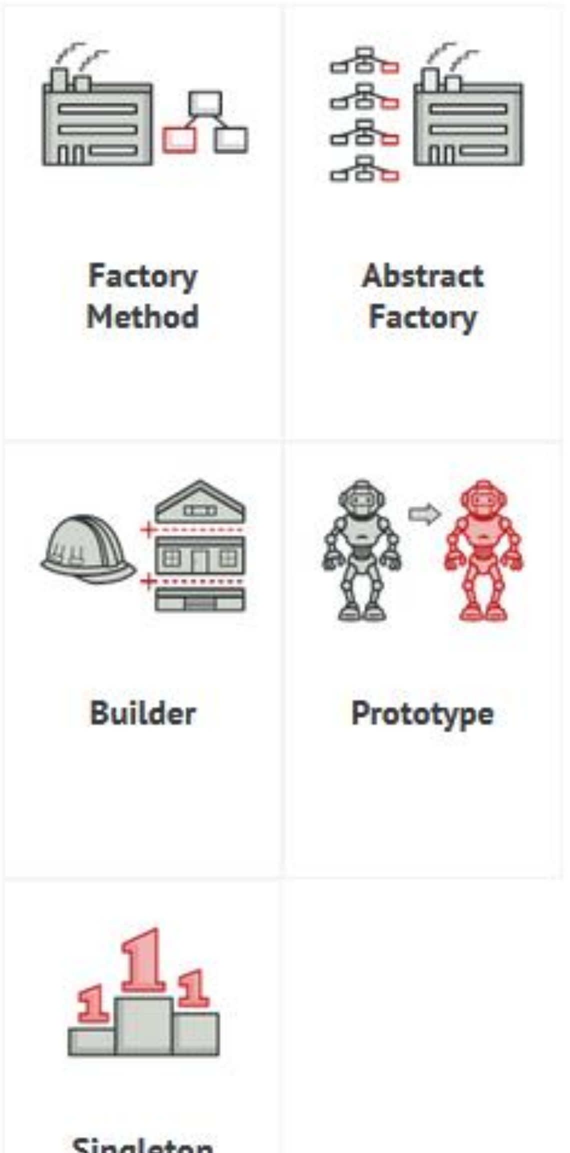
Design Patterns

- The idea was picked up by four authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm.
- In 1995, they published Design Patterns: Elements of Reusable Object-Oriented Software, in which they applied the concept of design patterns to programming.
- The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly.
- Due to its lengthy name, people started to call it "the book by the **gang of four**" which was soon shortened to simply "the GOF book".

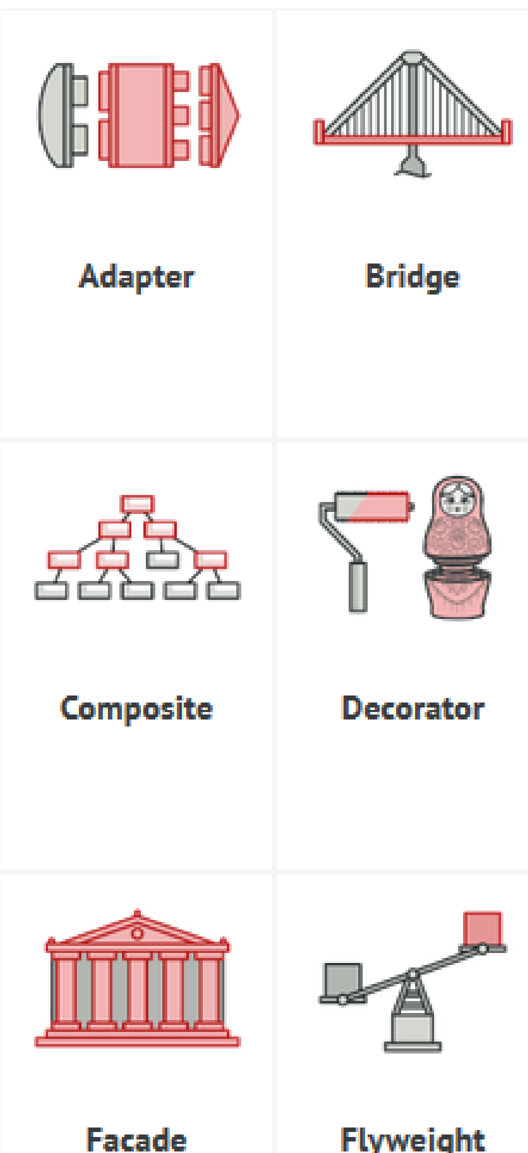
Design Patterns (Category)

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

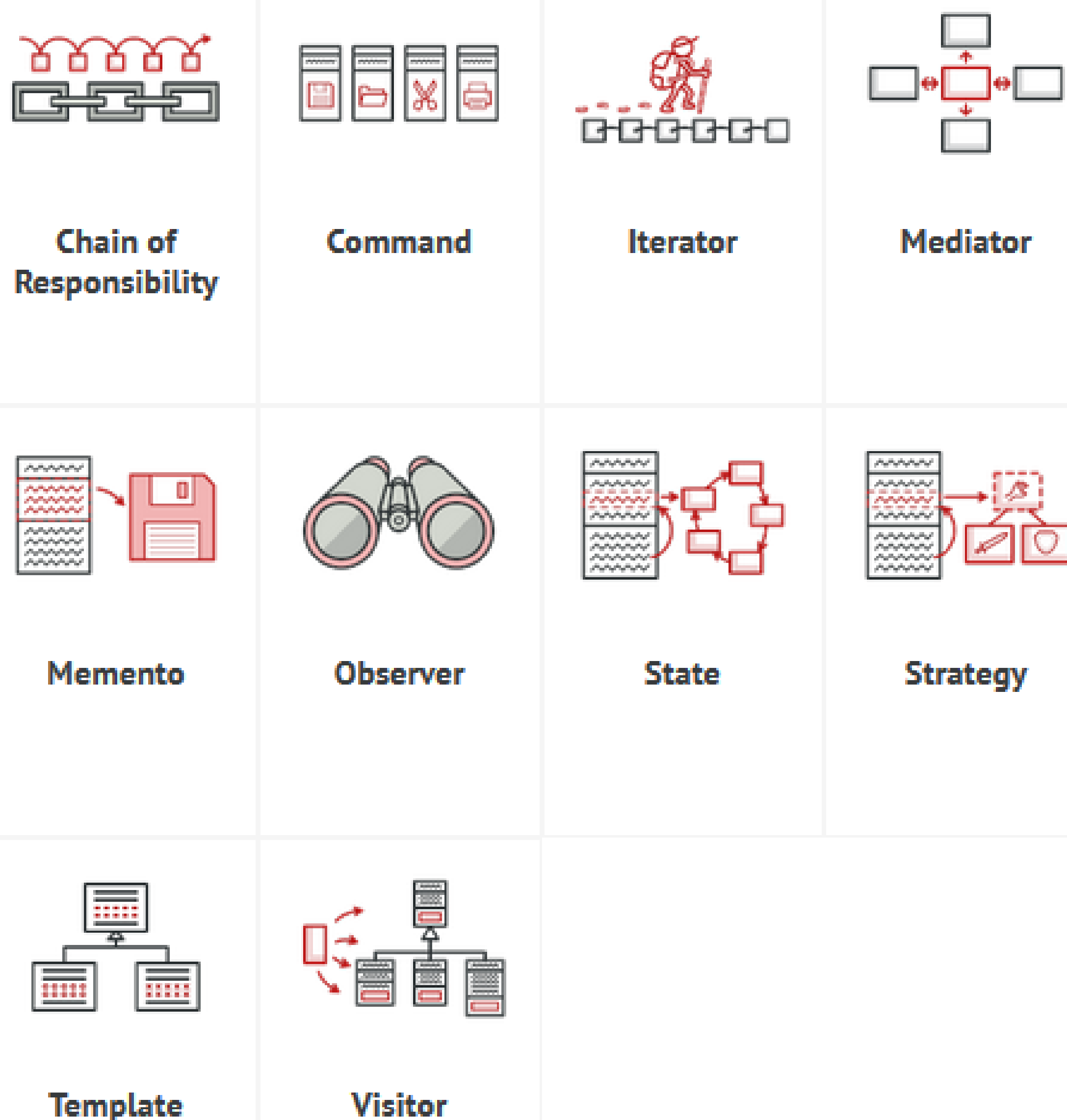
Creational patterns



Structural patterns



Behavioral patterns



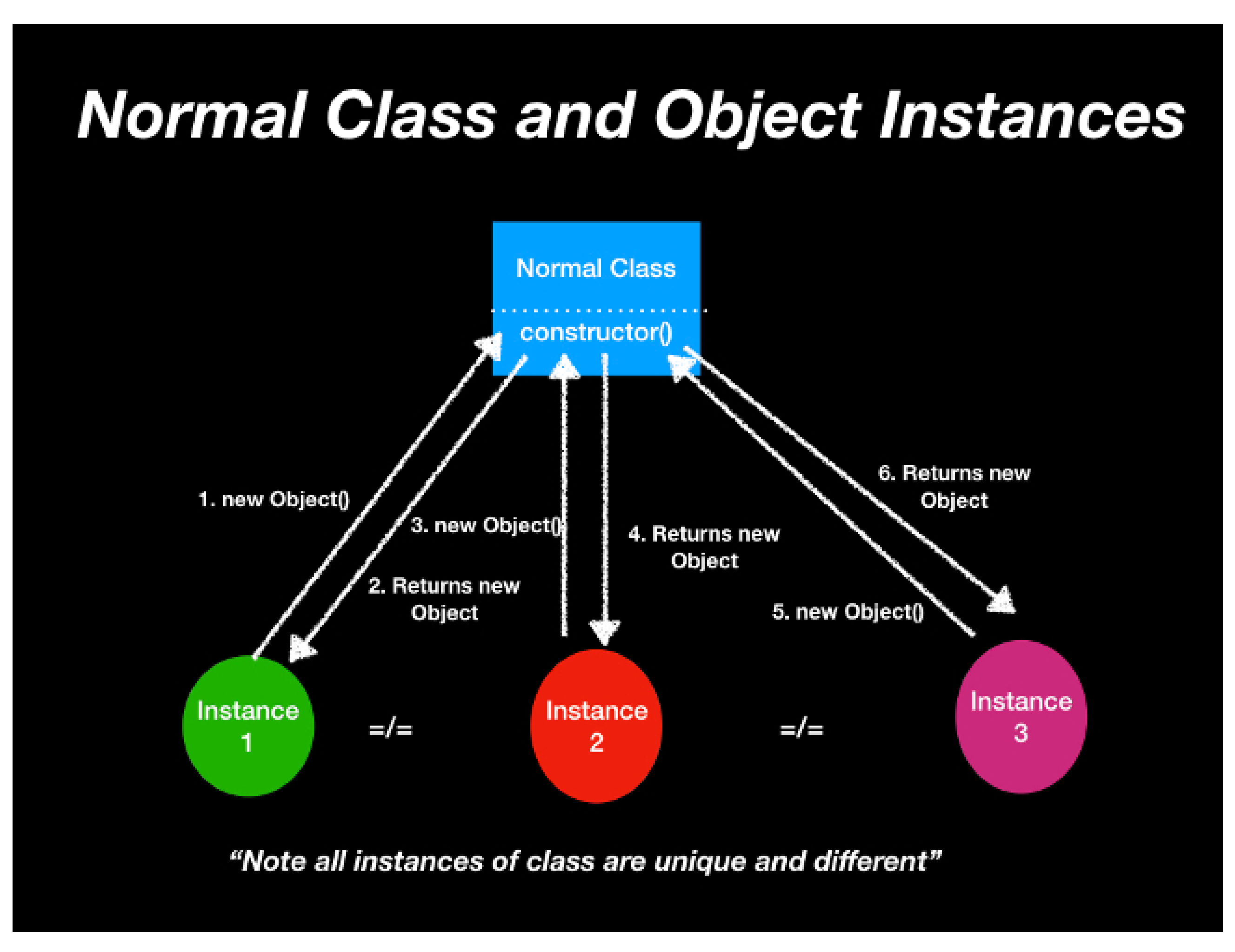
Singleton Pattern

Singleton Pattern

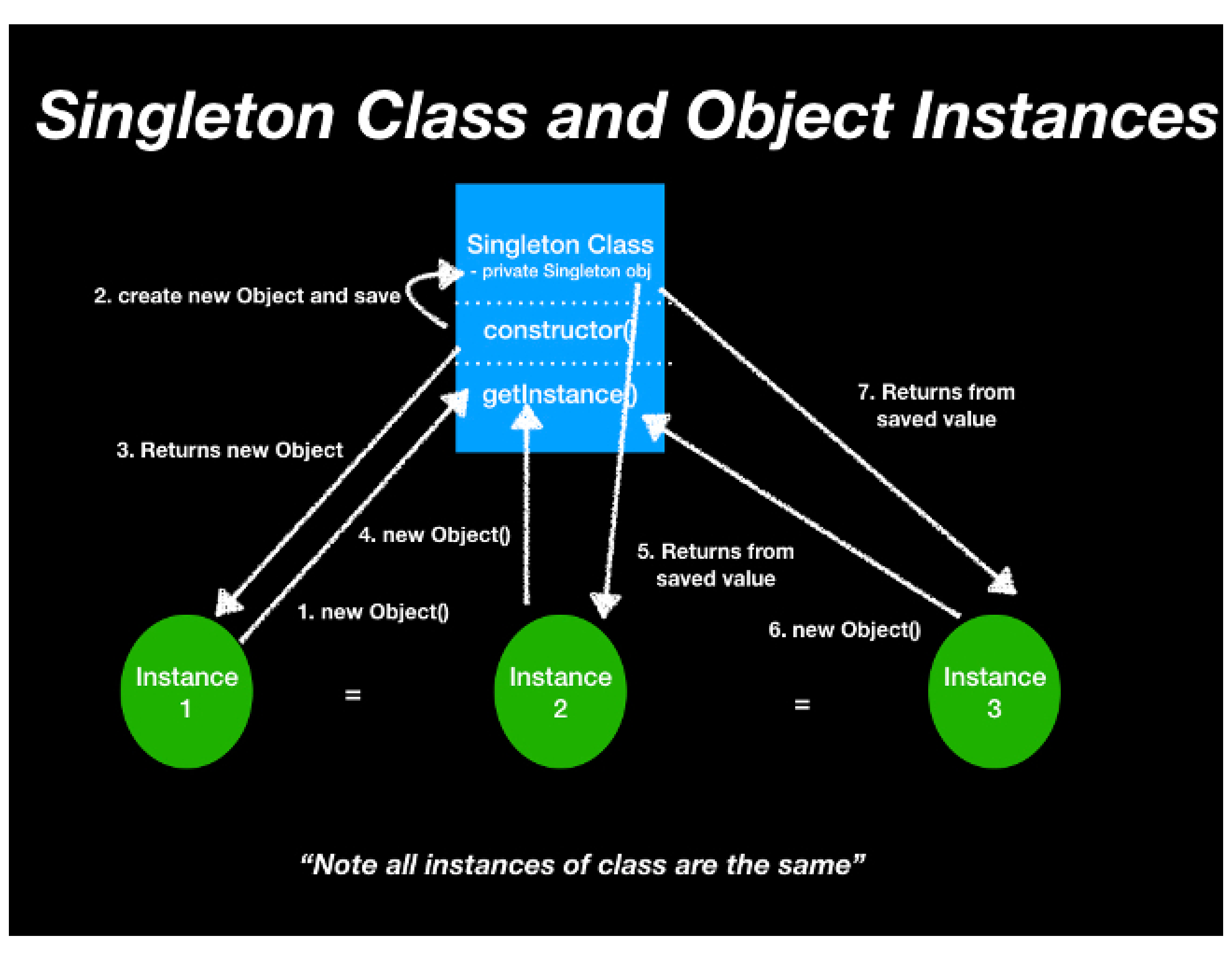
Singleton Pattern

- **Intent**
  - A creational design pattern that ensures only one instantiation (of the class) is created and avoids creating multiple instances of the same object.
  - Provides a global access point to this instance.
- **Problem**
  - Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Singleton Pattern



Singleton Pattern



Singleton Pattern

- The Singleton pattern solves two problems at the same time, violating the *Single Responsibility Principle*:

1. **Ensure that a class has just a single instance**. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

- Here's how it works:
  - imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.
  - Note that this behavior is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.

Singleton Pattern

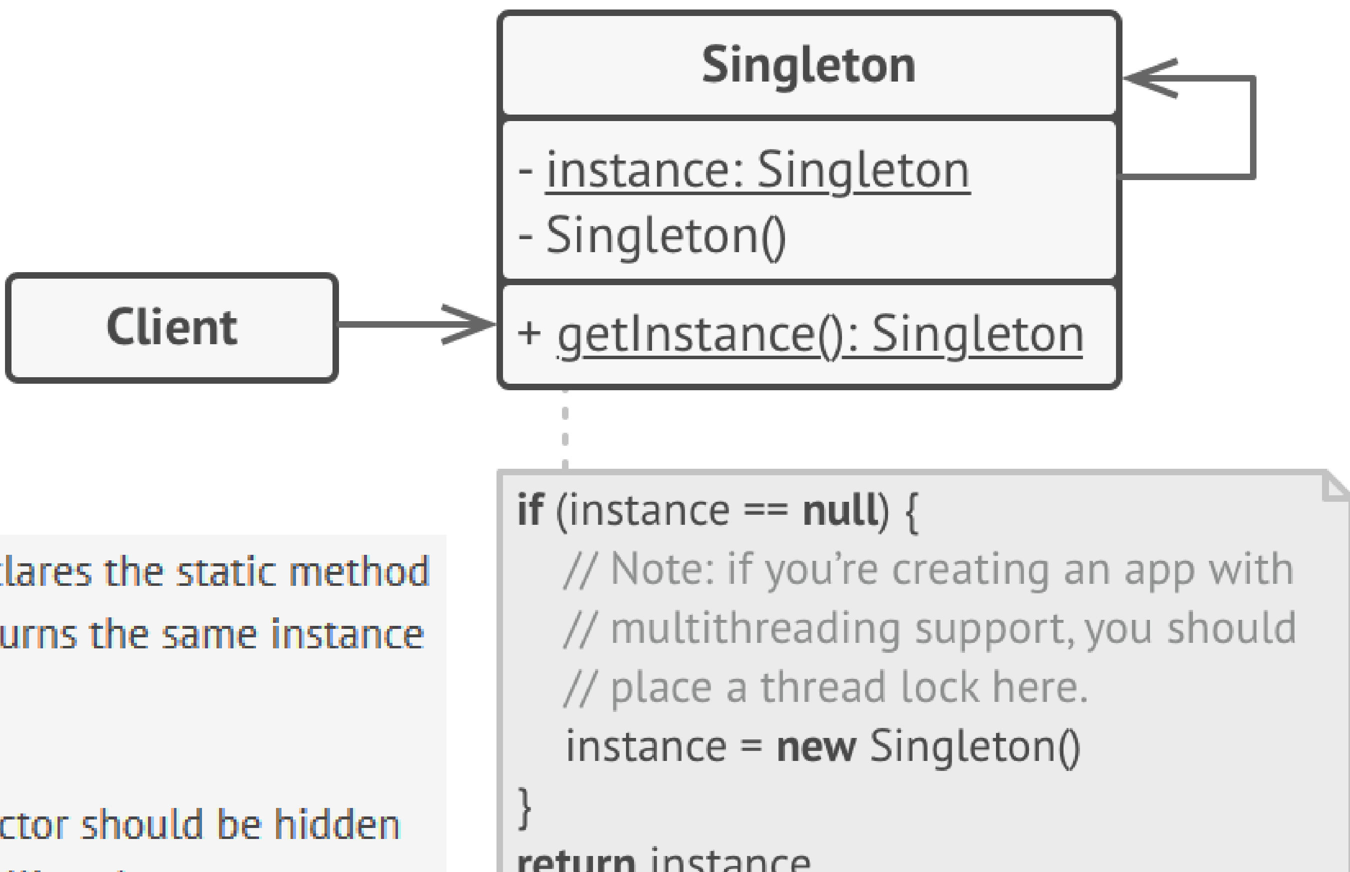
2. Provide a global access point to that instance. Remember those global variables that are used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

- Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

Singleton Pattern

- All implementations of the Singleton have these two steps in common:
  - Make the default constructor private, to prevent other objects from using the *new* operator with the Singleton class.
  - Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

Singleton Pattern (Structure)



The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

Singleton Pattern

```java
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!");
   }
}
```

```java
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Applicability

- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.

- Use the Singleton pattern when you need stricter control over global variables.

Factory Patterns

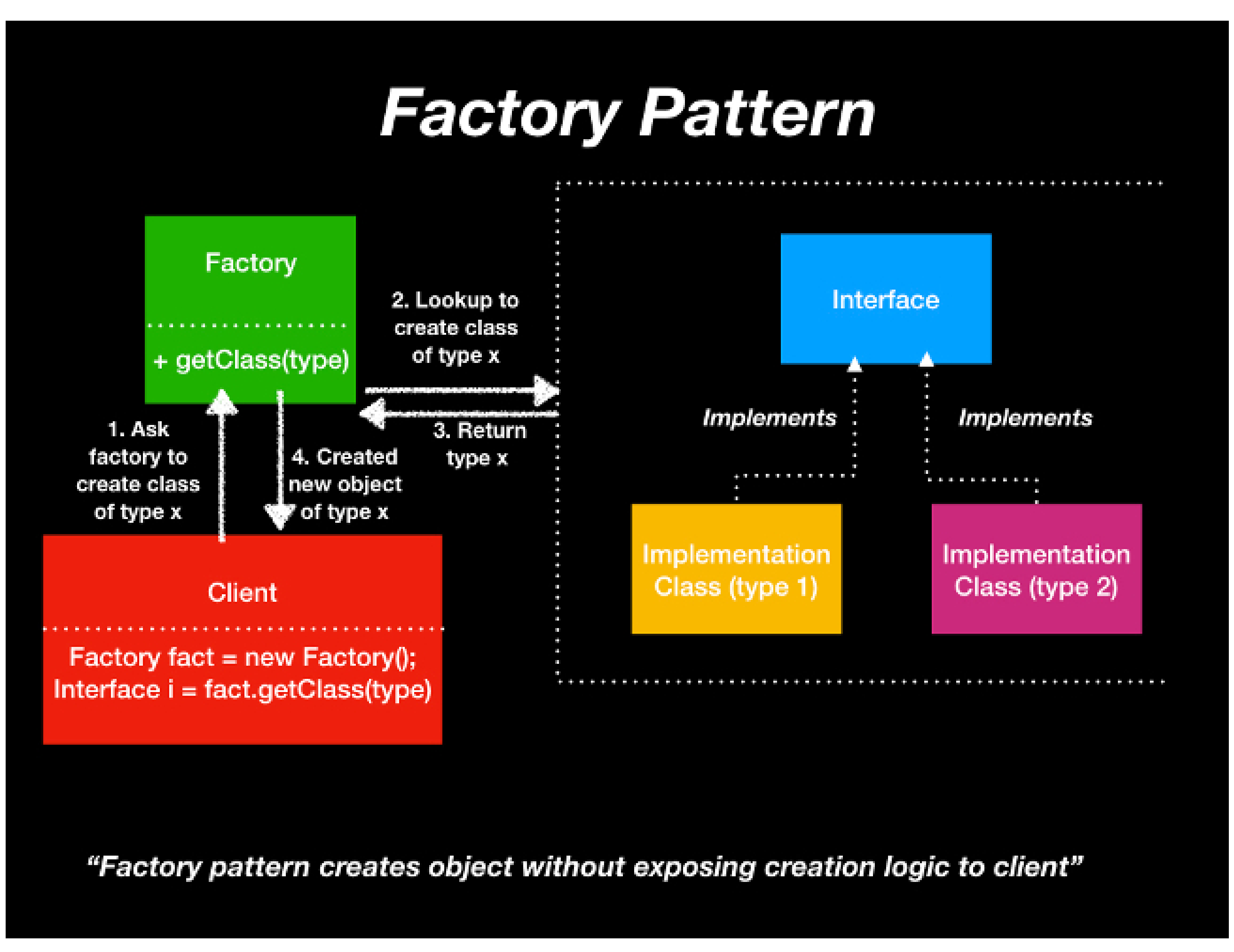
Factory Patterns

- **Intent**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - Defining a "virtual" constructor.
  - The *new* operator considered harmful.

Factory Patterns

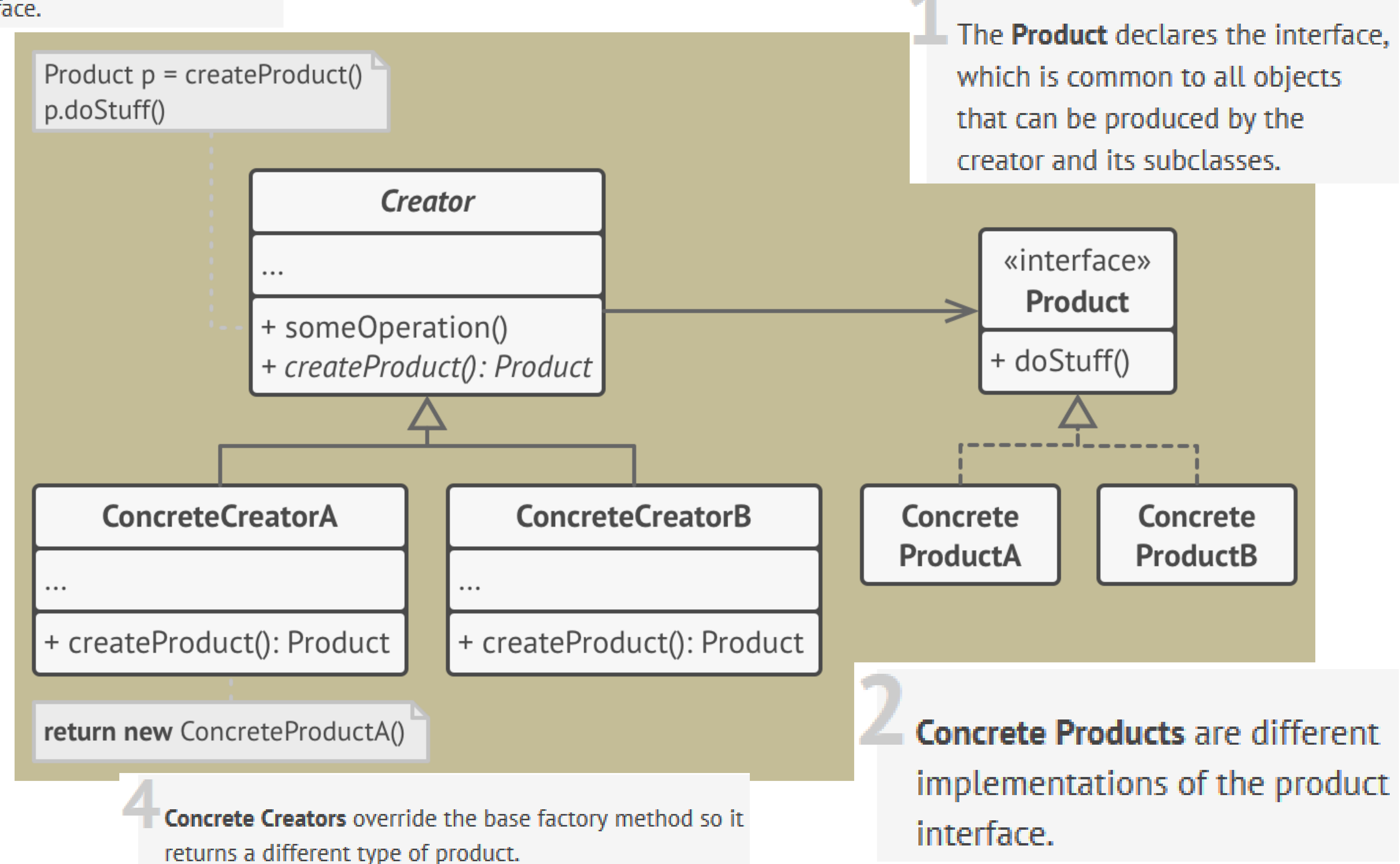
- **Problem**
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Factory Patterns

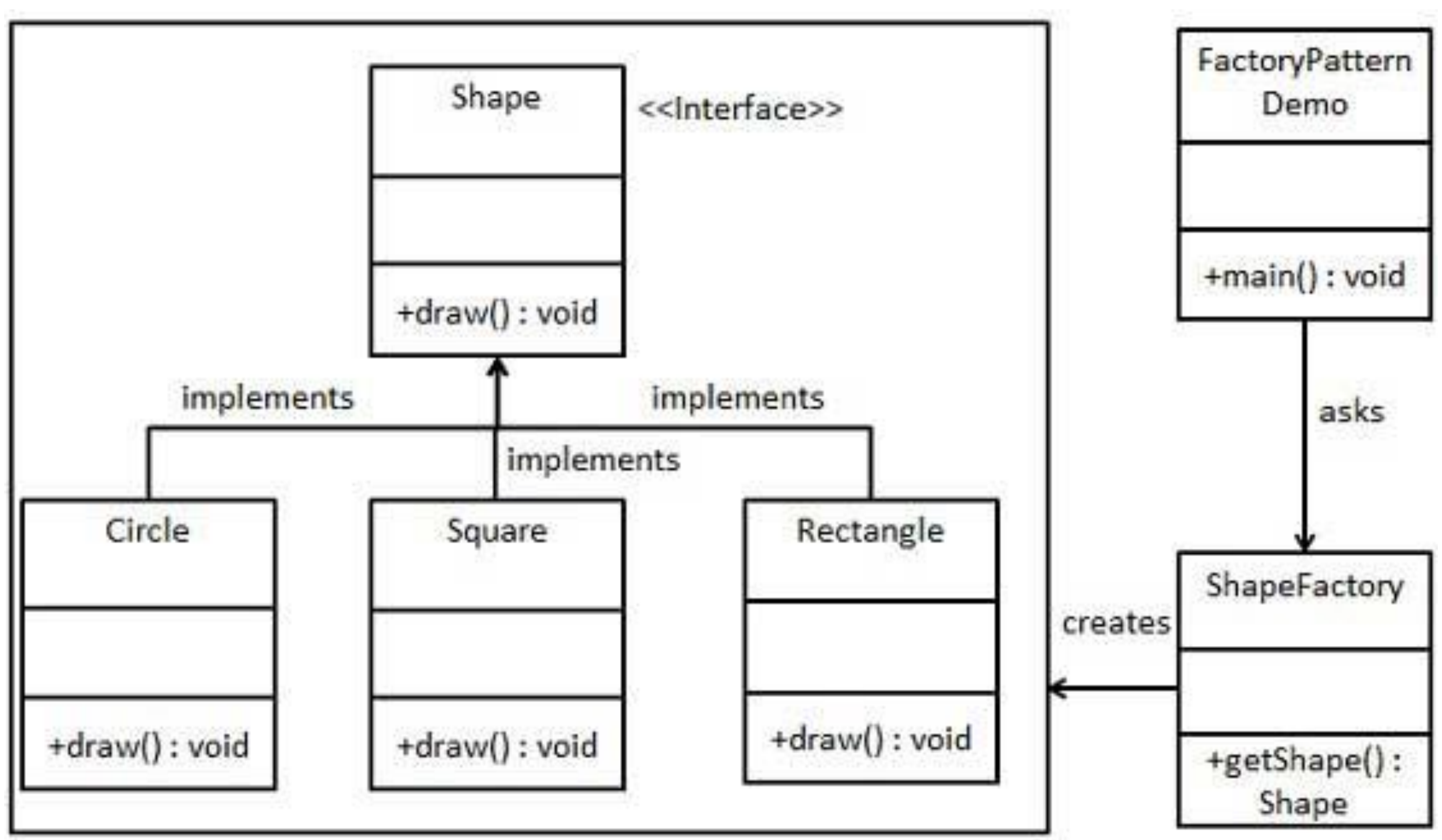


Factory Patterns

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.



Factory Patterns



Step 1

Create an interface.

Shape.java

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

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {

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

Circle.java

```
public class Circle implements Shape {

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

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
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("RECTANGLE")){
         return new Rectangle();

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

      return null;
   }
}
```

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
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");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Applicability

- Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.
- Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
- Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

Observer Pattern

Also known as: ***Event-Subscriber, Listener***

Observer Pattern

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

Observer Pattern

- For the sake of simplicity, think about what happens when you follow someone on Twitter. You are essentially asking Twitter to send you (**the observer**) tweet updates of the person (**the subject**) you followed.

- The pattern consists of two actors:
  - *the **observer*** who is interested in the updates and
  - *the **subject*** who generates the updates.

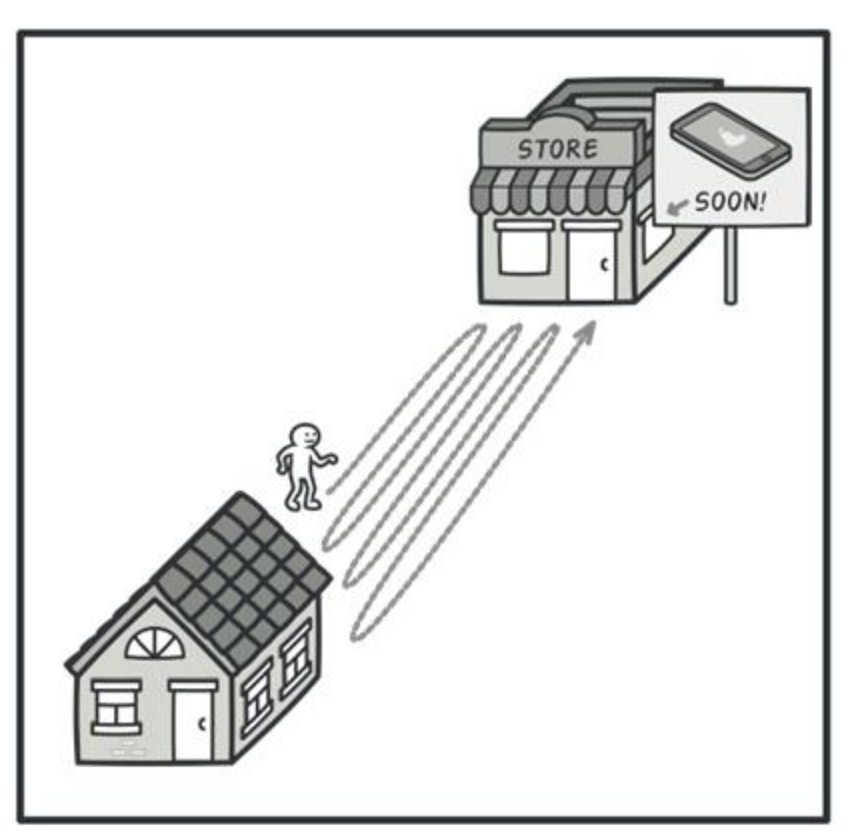
Observer Pattern

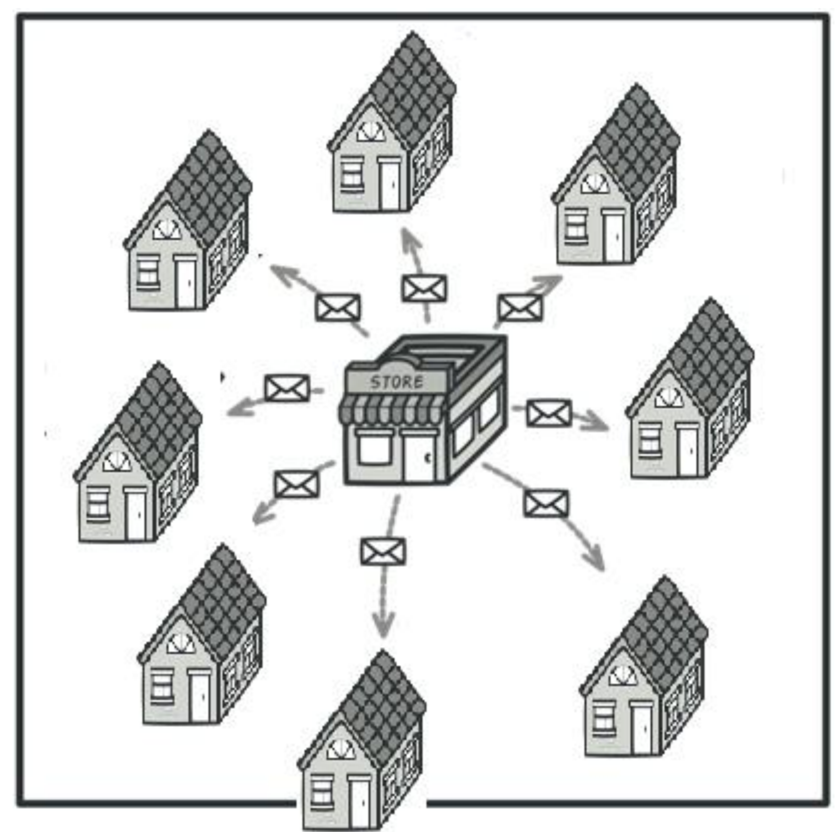
- **Problem**
  - Imagine that you have two types of objects: a **Customer** and a **Store**. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.
  - The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

Observer Pattern

- On the other hand, the store could send tons of emails (which might be considered **spam**) to all customers each time a new product becomes available.
  - This would save some customers from endless trips to the store.
  - At the same time, it'd upset other customers who aren't interested in new products.
- It looks like we've got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

Observer Pattern



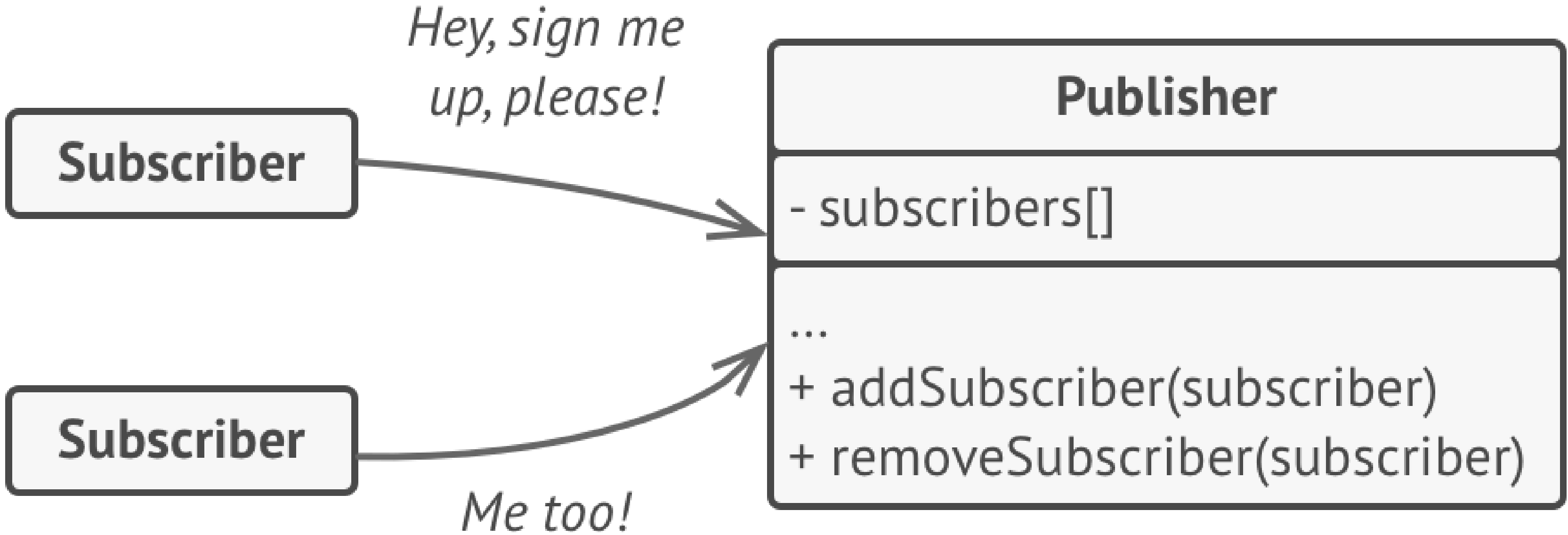


Visiting the store vs. sending spam

Observer Pattern

- **Solution**
  - The object that has some interesting state is often called ***subject***, but since it's also going to notify other objects about the changes to its state, we'll call it ***publisher***. All other objects that want to track changes to the publisher's state are called ***subscribers***.
  - The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

Observer Pattern

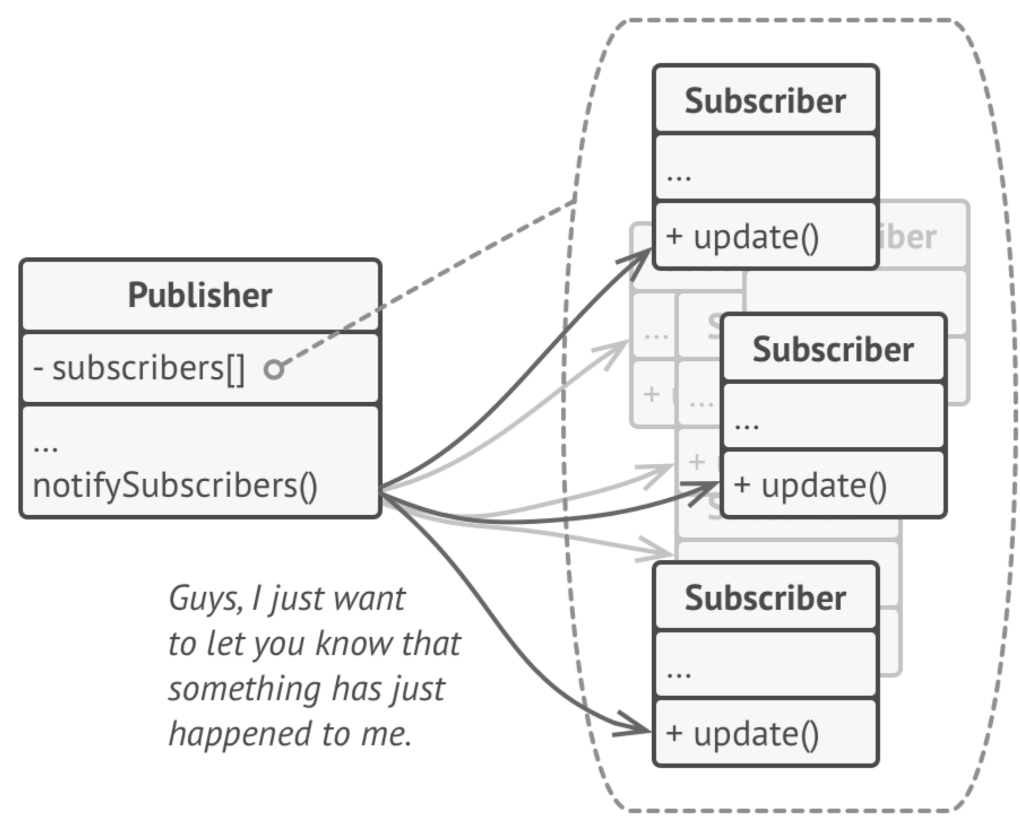


A subscription mechanism lets individual objects subscribe to event notifications.

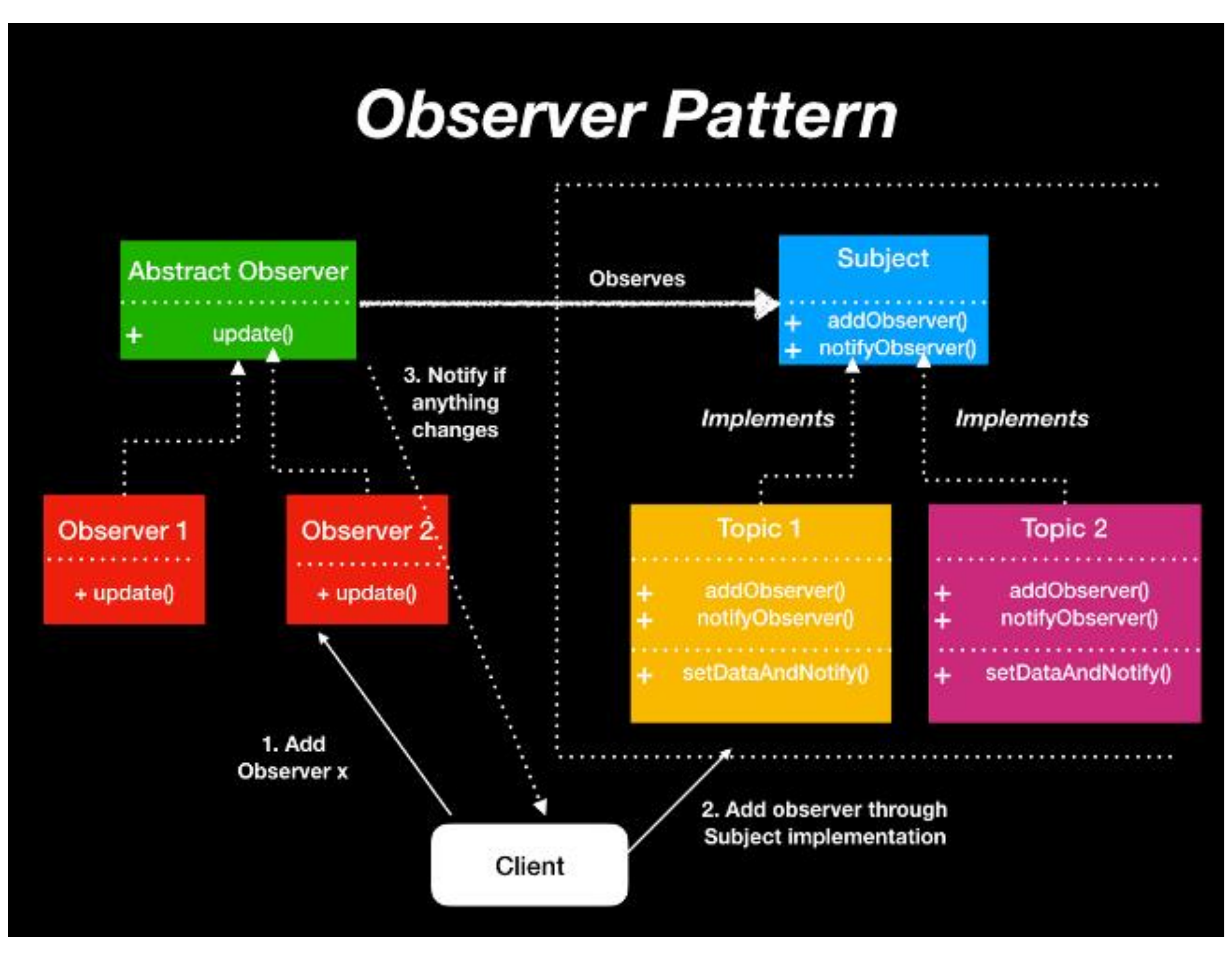
Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

Observer Pattern

- Real apps might have dozens of different subscriber classes that are **interested in tracking events** of the same publisher class.
- You wouldn't want to couple the publisher to all of those classes. Besides, you might not even know about some of them beforehand if your publisher class is supposed to be used by other people.
- That's why it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface.
- This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.



Observer Pattern



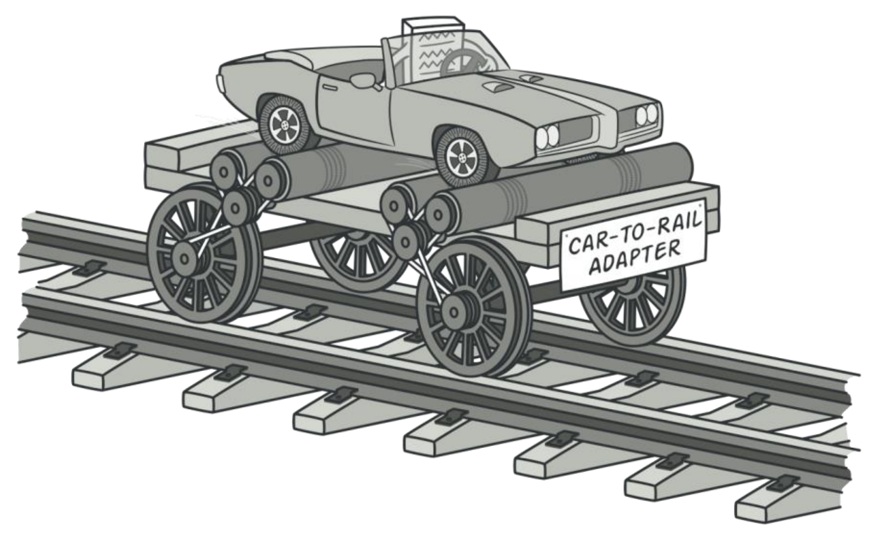
Applicability

- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
- Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

Adapter Pattern

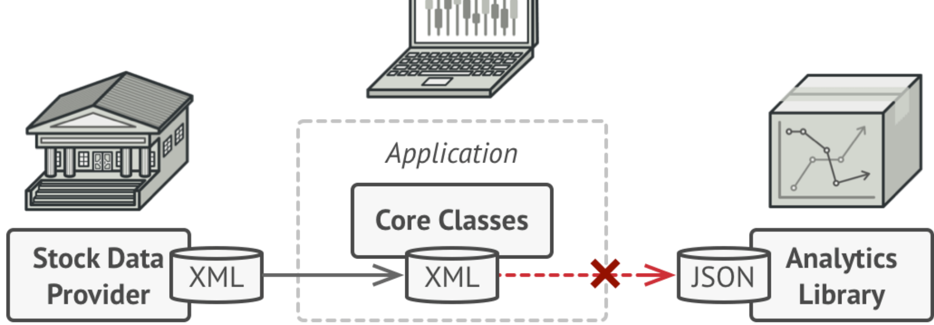
Adapter Pattern

- **Intent**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
  - Impedance match an old component to a new system



Adapter Pattern

- **Problem**
  - Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
  - At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



  - You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible.

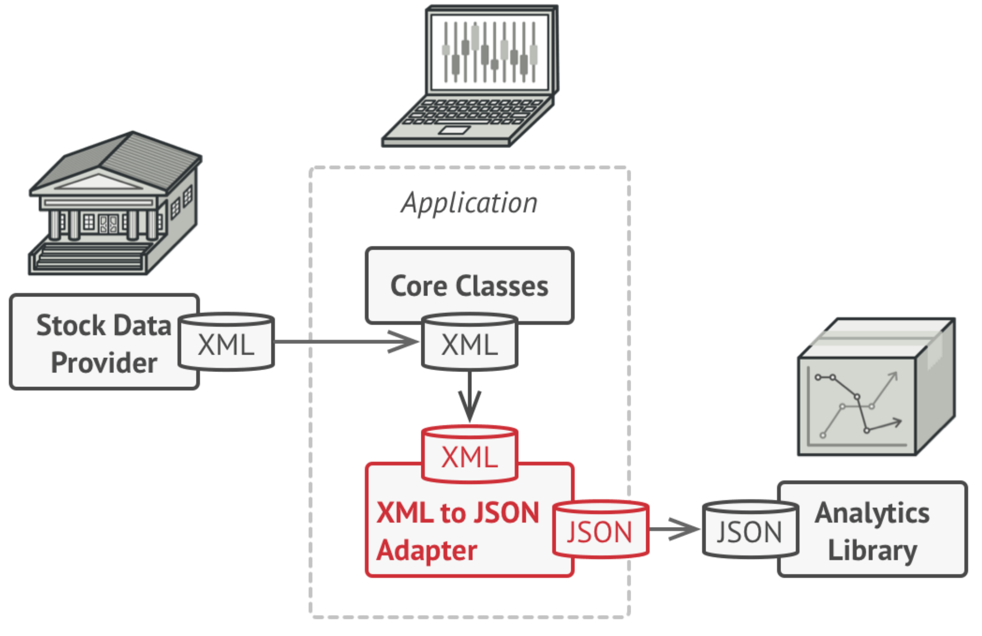
Adapter Pattern

- **Solution**
  - You can create an ***adapter***. This is a special object that converts the interface of one object so that another object can understand it.
  - An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn’t even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

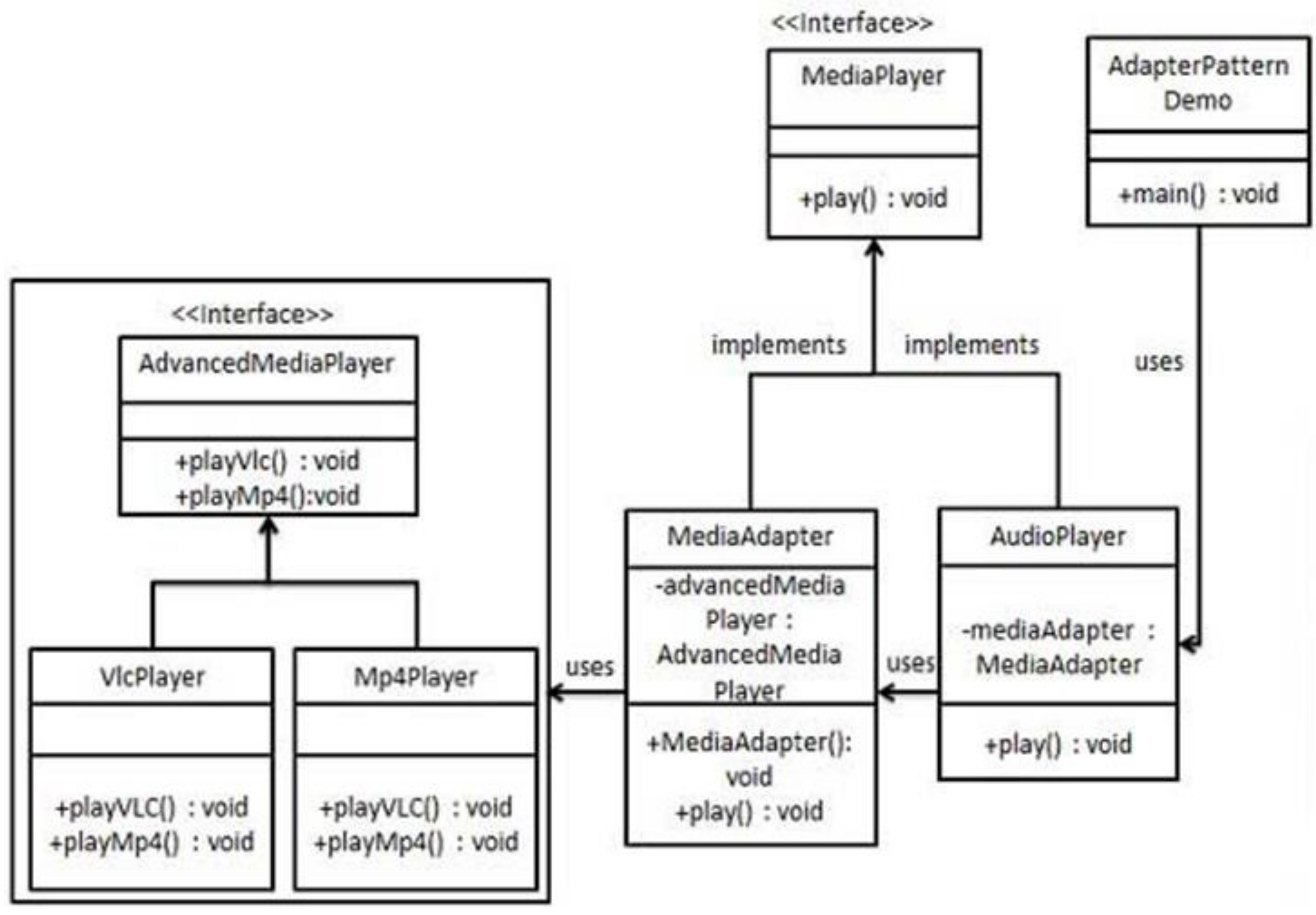
Adapter Pattern

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

1. The adapter gets an interface, **compatible** with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.



Adapter Pattern



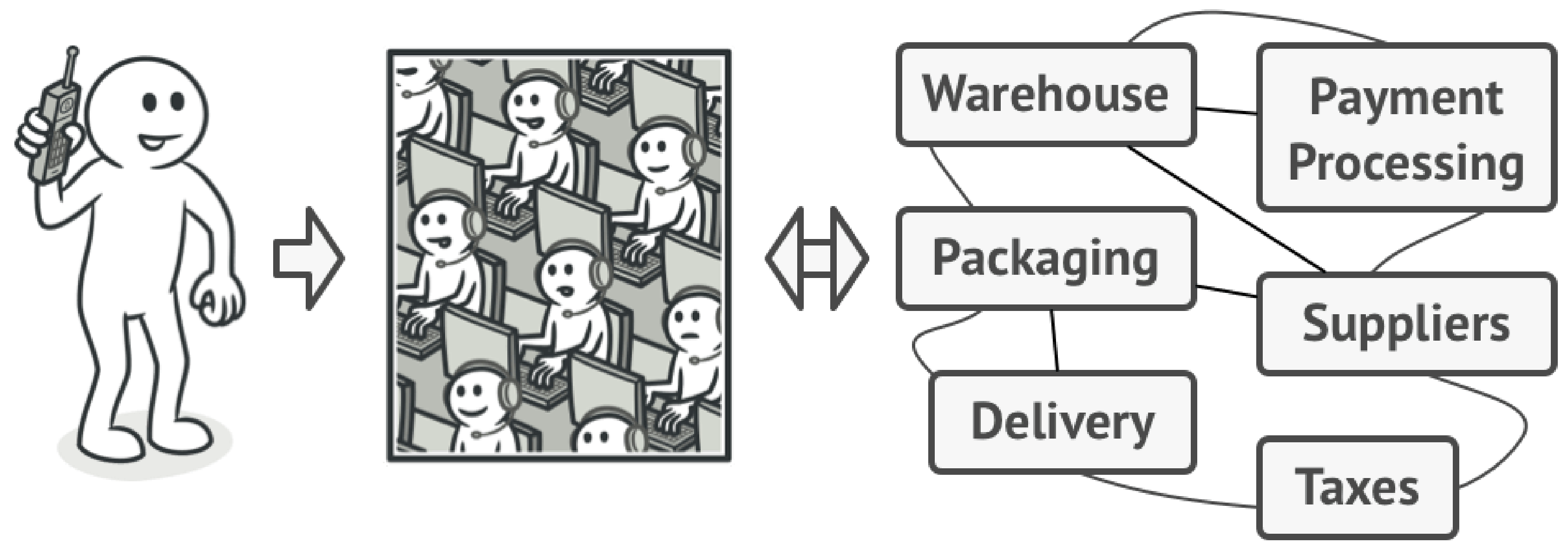
Applicability

- Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.
- Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

Façade Pattern

- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.
- **Problem**
  - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Façade Pattern

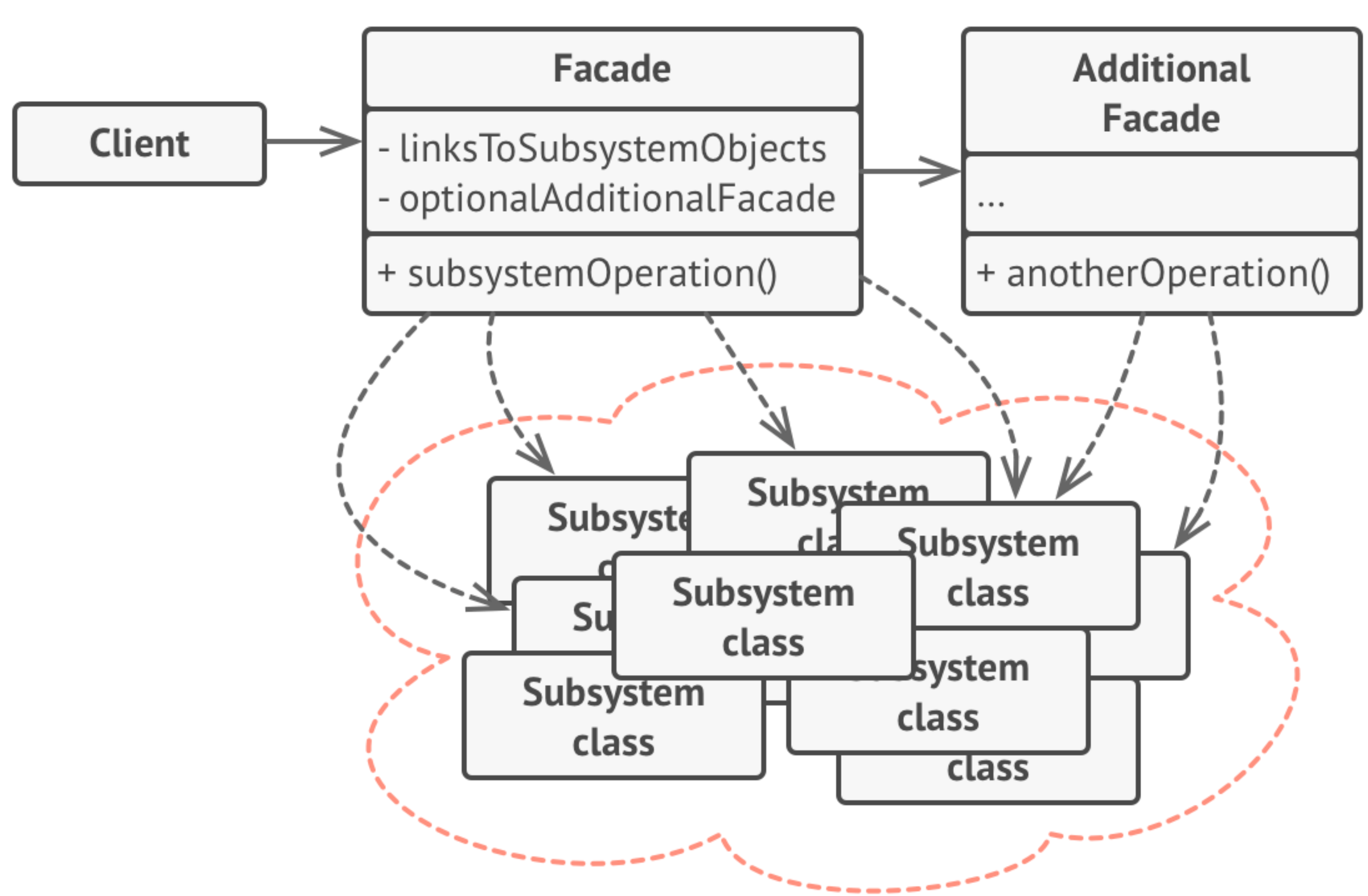


When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you with a simple voice interface to the ordering system, payment gateways, and various delivery services.

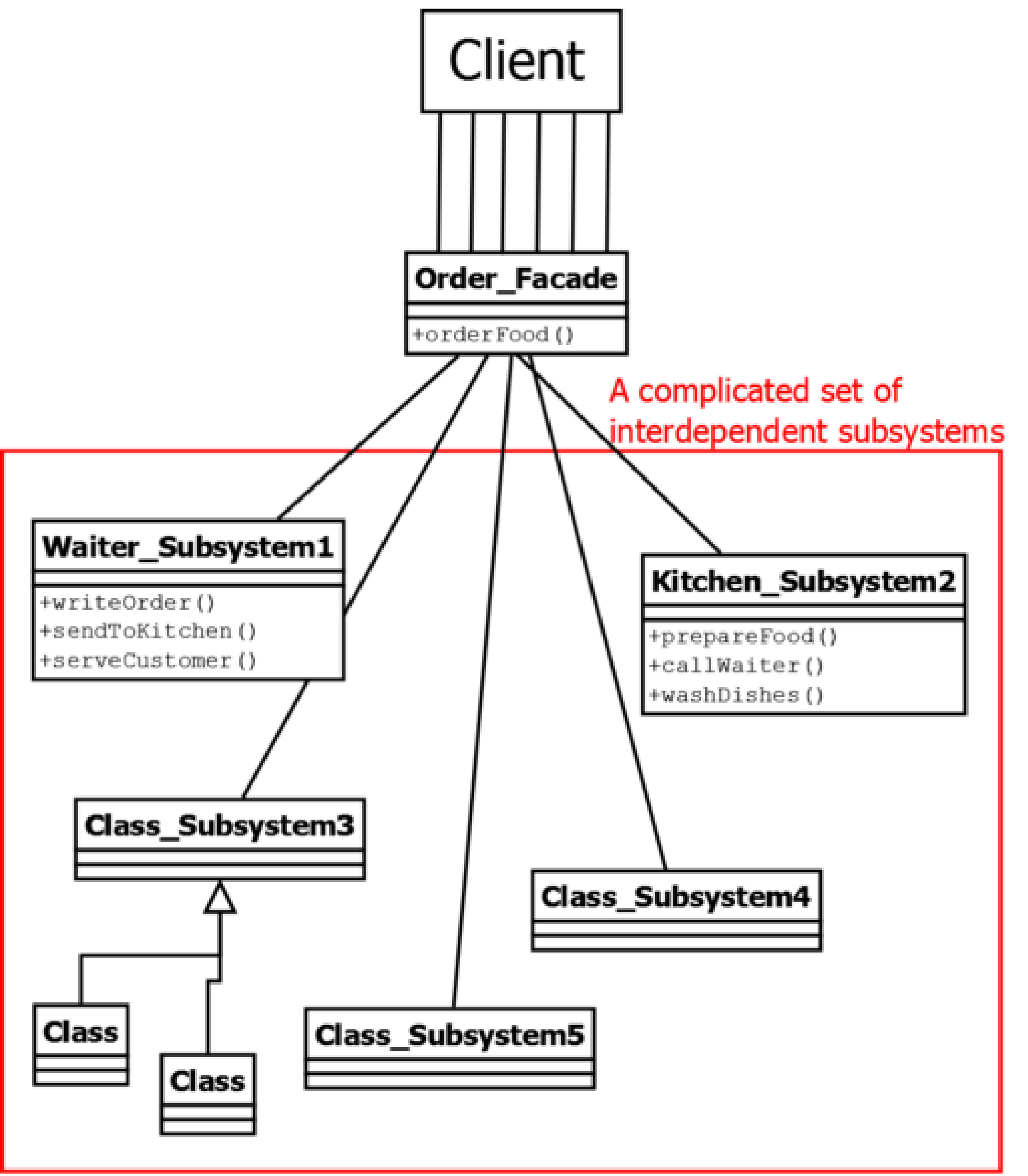
Façade Pattern

- Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem.
- It also promotes decoupling the subsystem from its potentially many clients.
- On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

Façade Pattern



Façade Pattern



Façade Pattern

```
#include <iostream>
using namespace std;


class Waiter_Subsystem1
{
public:
  void writeOrder()    { cout << " Waiter writes client's order\n";}
  void sendToKitchen(){ cout << " Send order to kitchen\n";}
  void serveCustomer(){ cout << " Yeeei customer is served!!!\n";}
};


class Kitchen_Subsystem2
{
public:
    void prepareFood(){ cout << " Cook food\n";}
    void callWaiter() { cout << " Call Waiter\n";}
    void washDishes() { cout << " Wash the dishes\n";}
};
```

Façade Pattern

```
class Order_Facade
{
private:
    Waiter_Subsystem1waiter;
    Kitchen_Subsystem2 kitchen;
public:
    void orderFood()
    {
        cout << "A series of interdependent calls on various
subsystems:\n";
        waiter.writeOrder();
        waiter.sendToKitchen();
        kitchen.prepareFood();
        kitchen.callWaiter();
        waiter.serveCustomer();
        kitchen.washDishes();
    }
};
```

```
int main(int argc, char *argv[])
{
    // Simple for the client
    // no need to know the order or the
    // dependencies among various subsystems.
    Order_Facade facade;
    facade.orderFood();


return 0;
}


// Output
// A series of interdependent calls on various subsystems:
//  Waiter writes client's order
//  Send order to kitchen
//  Cook food
//  Call Waiter
//  Yeeei customer is served!!!
//  Wash the dishes
```

Applicability

- ❑ Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.
- ❑ Use the Facade when you want to structure a subsystem into layers.