1. Classification of Design Patterns

12 January 2024 21:46

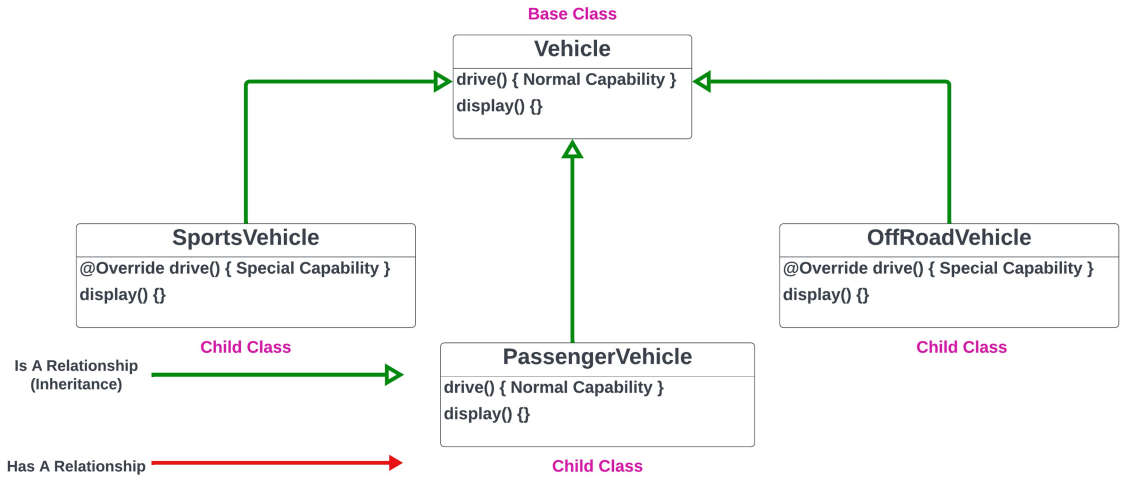
- **Creational Design Patterns**
    - **Abstract Factory**
    - **Builder**
    - **Factory Method**
    - **Prototype**
    - **Singleton**
- **Structural Design Patterns**
    - **Adapter**
    - **Composite**
    - **Bridge**
    - **Decorator**
    - **Façade**
    - **Flyweight**
    - **Proxy**
- **Behavioral Design Patterns**
    - **Chain of Responsibility**
    - **Interpreter**
    - **Iterator**
    - **Mediator**
    - **Memento**
    - **Observer**
    - **State**
    - **Strategy**
    - **Template Method**
    - **Visitor**

2. Strategy Design Pattern

12 January 2024 22:39

Pre-requisites:

1. We have a Vehicle Base class which have drive and display methods
2. **SportsVehicle** and **OffRoadVehicle** classes extending Vehicle class need Special Drive Capabilities so overrides drive method
3. **PassengerVehicle** class needs Normal Drive Capability so does not override drive method

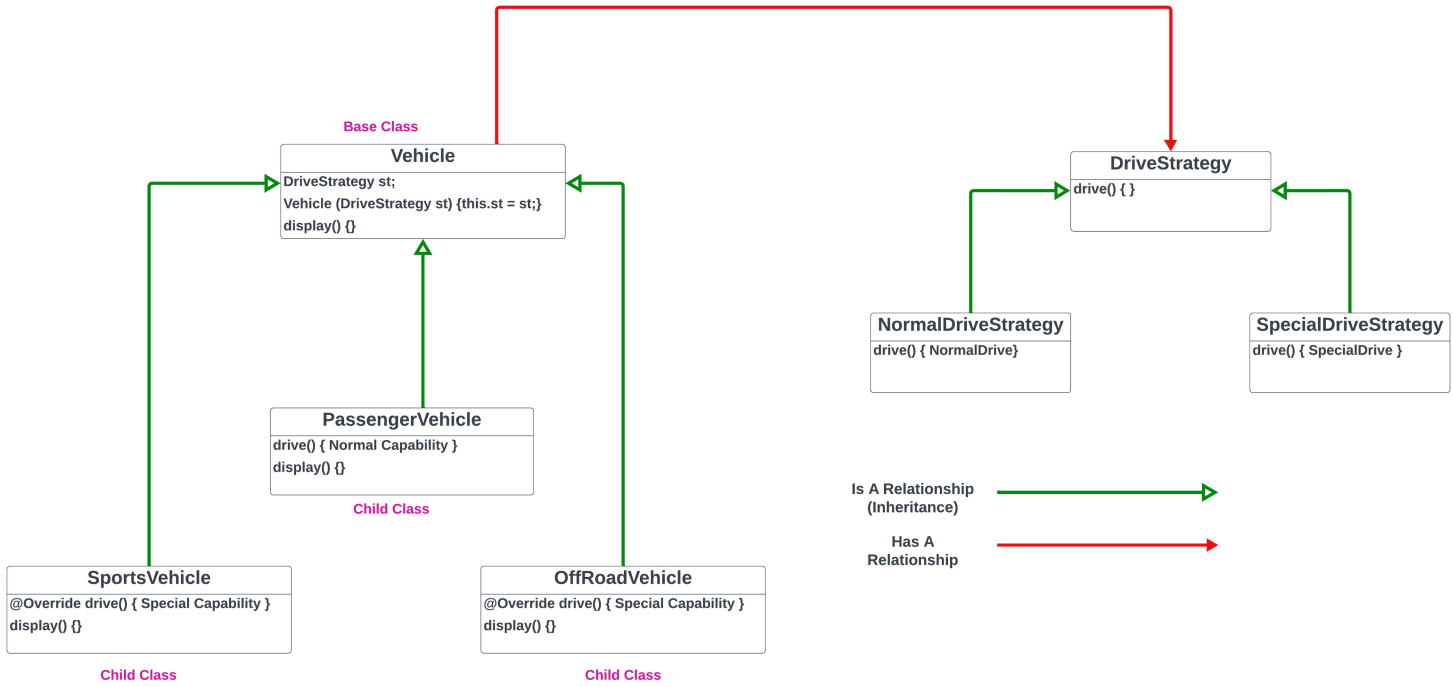


Problem:

1. Here both **SportsVehicle** and **OffRoadVehicle** class needs Special Drive capabilities and their functionality is different from Base class functionality so may result into duplication of code.
2. This duplication of code may increase when we need additional TypesVehicle classes

Solution:

1. This can be resolved using Strategy Design Pattern
2. Create a **DriveStrategy** Interface with concrete classes implementing the same as **NormalDriveStrategy** and **SpecialDriveStrategy**
3. In Vehicle class, use a variable of **DriveStrategy**
4. Now the individual classes have the responsibility to pass the Strategy to **Vehicle** class



Strategy Designs Implementation

```
public interface DriveStrategy {
    public void drive();
}
```

```
public class NormalDriveStrategy implements DriveStrategy {

    @Override
    public void drive() {
        System.out.println(x:"Normal Drive Capability");
    }

}
```

```
public class SportsDriveStrategy implements DriveStrategy {

    @Override
    public void drive() {
        System.out.println(x:"Special Drive Capability");
    }

}
```

Vehicle Classes Modification to use the above Strategy

```java
public class Vehicle {

    private DriveStrategy strategy;

    public Vehicle(DriveStrategy strategy) {
        this.strategy = strategy;
    }

    public void drive() {
        strategy.drive();
    }

    public DriveStrategy getDriveStrategy() {
        return this.strategy;
    }
}
```

```java
public class OffRoadVehicle extends Vehicle {
    public OffRoadVehicle() {
        super(new SportsDriveStrategy());
    }
}
```

```java
public class PassengerVehicle extends Vehicle {
    public PassengerVehicle() {
        super(new NormalDriveStrategy());
    }
}
```

```java
public class SportsVehicle extends Vehicle {
    public SportsVehicle() {
        super(new SportsDriveStrategy());
    }
}
```

3. Observer Design Pattern

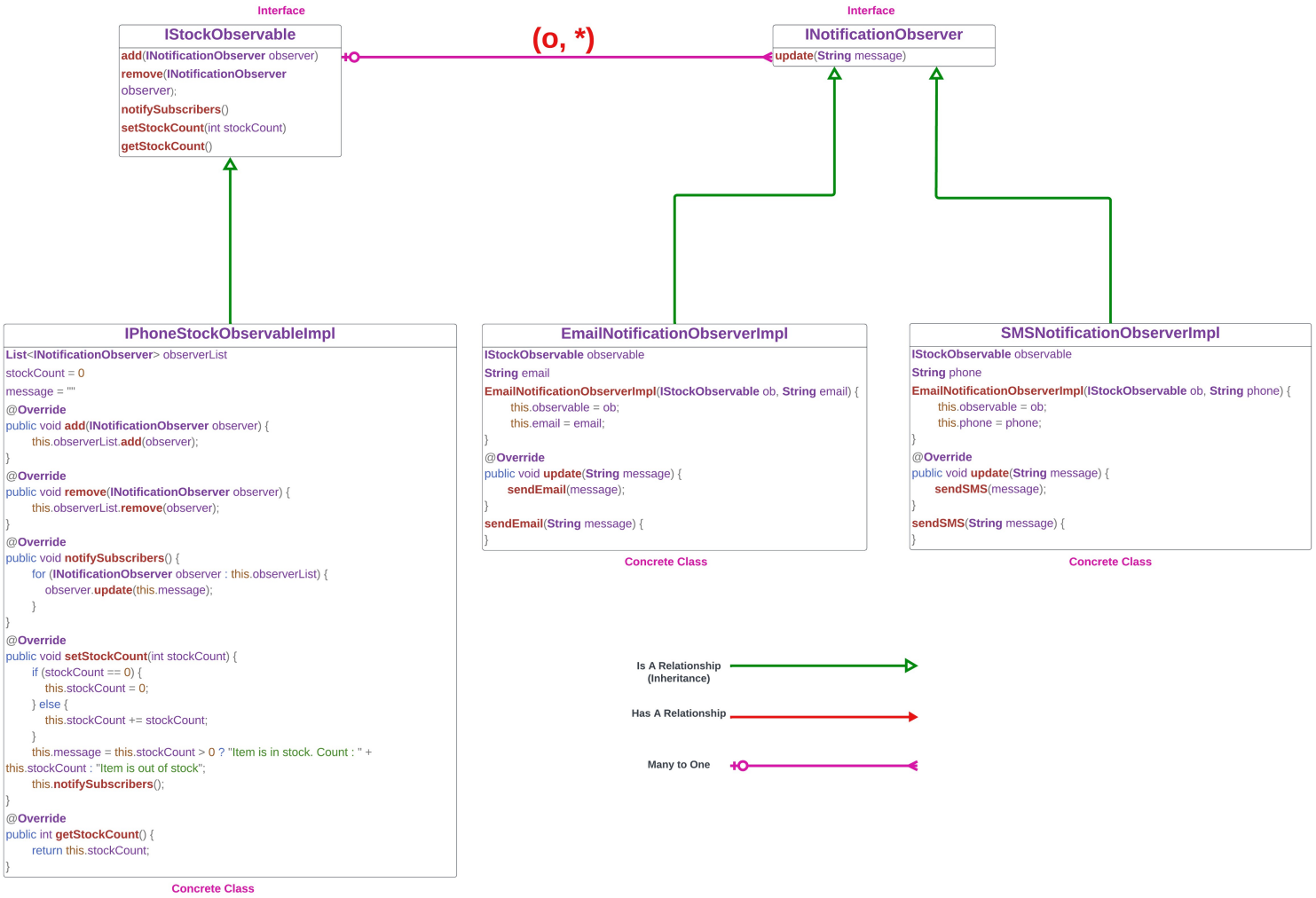
14 January 2024 22:25

Use-cases

- Amazon notifying customers once product is back in stock
- Weather App notifying users about change in temperature

Amazon Back in Stock Notification

The below ER Diagram shows how Observer design pattern would work in this scenario.



Observable Code:

This is mainly the service which is responsible to notify the subscribed customers.

Observable Interface:

```
public interface IStockObservable {
    public void add(INotificationObserver observer);

    public void remove(INotificationObserver observer);

    public void notifySubscribers();

    public void setStockCount(int stockCount);

    public int getStockCount();
}
```

Observable Concrete Class Implementation:

```
public class IPhoneStockObservableImpl implements IStockObservable {

    List<INotificationObserver> observerList = new ArrayList<INotificationObserver>();
    int stockCount = 0;
    String message = "";

    @Override
    public void add(INotificationObserver observer) {
        this.observerList.add(observer);
    }

    @Override
    public void remove(INotificationObserver observer) {
        this.observerList.remove(observer);
    }

    @Override
    public void notifySubscribers() {
        for (INotificationObserver observer : this.observerList) {
            observer.update(this.message);
        }
    }

    @Override
    public void setStockCount(int stockCount) {
        if (stockCount == 0) {
            this.stockCount = 0;
        } else {
            this.stockCount += stockCount;
        }
        this.message = this.stockCount > 0 ? "Item is in stock. Count : " + this.stockCount : "Item is out of stock";
        this.notifySubscribers();
    }

    @Override
    public int getStockCount() {
        return this.stockCount;
    }

}
```

Observer Code:

This is mainly the service on the client side which will be called in case any changes happen to Observable code.

Observer Interface:

```
public interface INotificationObserver {
    public void update(String message);
}
```

Observer Concrete Classes Implementation:

```java
public class EmailNotificationObserverImpl implements INotificationObserver {
    @SuppressWarnings("unused")
    private IStockObservable observable;
    private String email;

    public EmailNotificationObserverImpl(IStockObservable ob, String email) {
        this.observable = ob;
        this.email = email;
    }

    @Override
    public void update(String message) {
        sendEmail(message);
    }

    private void sendEmail(String message) {
        System.out.println("Notified via SMS to phone number : " + this.email + ". Message : " + message);
    }
}
```

```java
public class SMSNotificationObserverImpl implements INotificationObserver {
    @SuppressWarnings("unused")
    private IStockObservable observable;
    private String phoneNumber;

    public SMSNotificationObserverImpl(IStockObservable ob, String phoneNumber) {
        this.observable = ob;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public void update(String message) {
        sendSMS(message);
    }

    private void sendSMS(String message) {
        System.out.println("Notified via SMS to phone number : " + this.phoneNumber + ". Message : " + message);
    }

}
```

Main Store Class which will make changes to Observable

```java
public class Store {
    Run | Debug
    public static void main(String[] args) {
        IStockObservable iphoneObservable = new IPhoneStockObservableImpl();

        INotificationObserver observer1 = new EmailNotificationObserverImpl(iphoneObservable, email:"abc.xyz@ymail.com");
        INotificationObserver observer2 = new EmailNotificationObserverImpl(iphoneObservable, email:"pqr.xyz@ymail.com");
        INotificationObserver observer3 = new SMSNotificationObserverImpl(iphoneObservable, phoneNumber:"+91-8456325478");
        INotificationObserver observer4 = new SMSNotificationObserverImpl(iphoneObservable, phoneNumber:"+91-9569852633");

        iphoneObservable.add(observer1);
        iphoneObservable.add(observer2);
        iphoneObservable.add(observer3);
        iphoneObservable.add(observer4);

        iphoneObservable.setStockCount(stockCount:10);
        iphoneObservable.setStockCount(stockCount:0);
        iphoneObservable.setStockCount(stockCount:100);
        iphoneObservable.setStockCount(stockCount:20);
    }
}
```

4. Decorator Design Pattern

15 January 2024 21:46

Use-cases

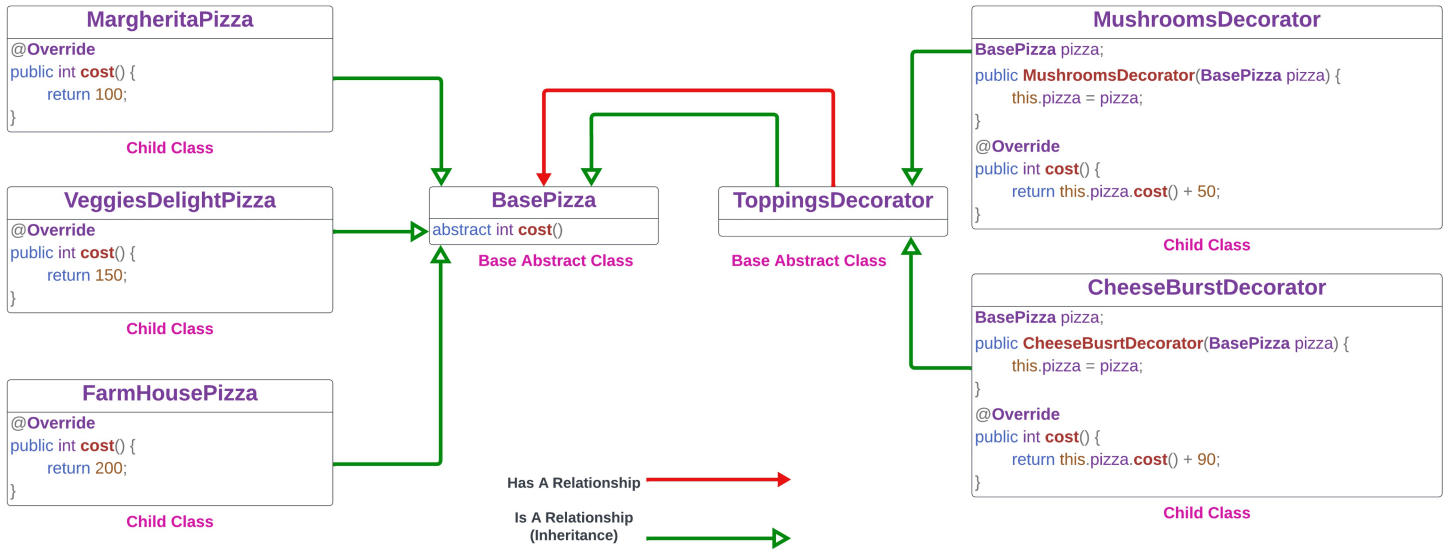
- Design to build Pizza builder and have functionality to add toppings like Extra Cheese and Mushrooms
- Design to build Coffee Machine and have functionality to add extra Milk and Creamlike

Advantages

- **Flexibility:** It provides a flexible alternative to sub-classing for extending functionality. With decorators, you can mix and match on an individual basis at runtime.
- **Avoids Class Explosion:** It helps in avoiding the need for higher-level classes to have knowledge of all possible options. It keeps these classes simple while letting you combine decorators as you please.
- **Single Responsibility Principle:** Each decorator has its own specific responsibility. This separation means that each class or method has a single job, improving readability and maintainability.

Design to build Pizza builder

The below ER Diagram shows how Observer design pattern would work in this scenario.



Code Changes:

A. Pizza Classes:

```
public abstract class BasePizza {
    public abstract int cost();
}
```

```
public class MargheritaPizza extends BasePizza {
    @Override
    public int cost() {
        return 100;
    }
}
```

```
public class VeggieDelightPizza extends BasePizza {
    @Override
    public int cost() {
        return 150;
    }
}
```

```
public class FarmHousePizza extends BasePizza {
    @Override
    public int cost() {
        return 200;
    }
}
```

B. Decorator Classes:

```java
public abstract class ToppingsDecorator extends BasePizza {

}
```

```java
public class MushroomsDecorator extends ToppingsDecorator {

    BasePizza pizza;

    public MushroomsDecorator(BasePizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public int cost() {
        return this.pizza.cost() + 50;
    }
}
```

```java
public class CheeseBurstDecorator extends ToppingsDecorator {

    BasePizza pizza;

    public CheeseBurstDecorator(BasePizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public int cost() {
        return this.pizza.cost() + 90;
    }

}
```

C. Pizza Builder Main Class:

```java
public class PizzaBuilder {
    Run | Debug
    public static void main(String[] args) {
        BasePizza margheritaPizza = new MushroomsDecorator(new CheeseBurstDecorator(new MargheritaPizza()));
        int margheritaPizzaCost = margheritaPizza.cost();
        System.out.println("Cost of Margherita Pizza with Extra Cheese and Mushrooms : " + margheritaPizzaCost);

        BasePizza veggiePizza = new MushroomsDecorator(new VeggieDelightPizza());
        int veggiePizzaCost = veggiePizza.cost();
        System.out.println("Cost of Veggie Delight Pizza with Extra Mushrooms : " + veggiePizzaCost);

        BasePizza farmHousePizza = new CheeseBurstDecorator(new FarmHousePizza());
        int farmHousePizzaCost = farmHousePizza.cost();
        System.out.println("Cost of FarmHouse Pizza with Extra Cheese : " + farmHousePizzaCost);
    }
}
```

5. Factory Design Pattern

15 January 2024 23:47

Use-cases

- The Factory Method pattern abstracts the decision-making process from the calling class.

Advantages

- **Reuse:** If I want to instantiate in many places, I don't have to repeat my condition, so when I come to add a new class, I don't run the risk of missing one.
- **Unit-Testability:** I can write 3 tests for the factory, to make sure it returns the correct types on the correct conditions, then my calling class only needs to be tested to see if it calls the factory and then the required methods on the returned class. It needs to know nothing about the implementation of the factory itself or the concrete classes.
- **Extensibility:** When someone decides we need to add a new class D to this factory, none of the calling code, neither unit tests or implementation, ever needs to be told. We simply create a new class D and extend our factory method. This is the very definition of Open-Closed Principle.

Pros

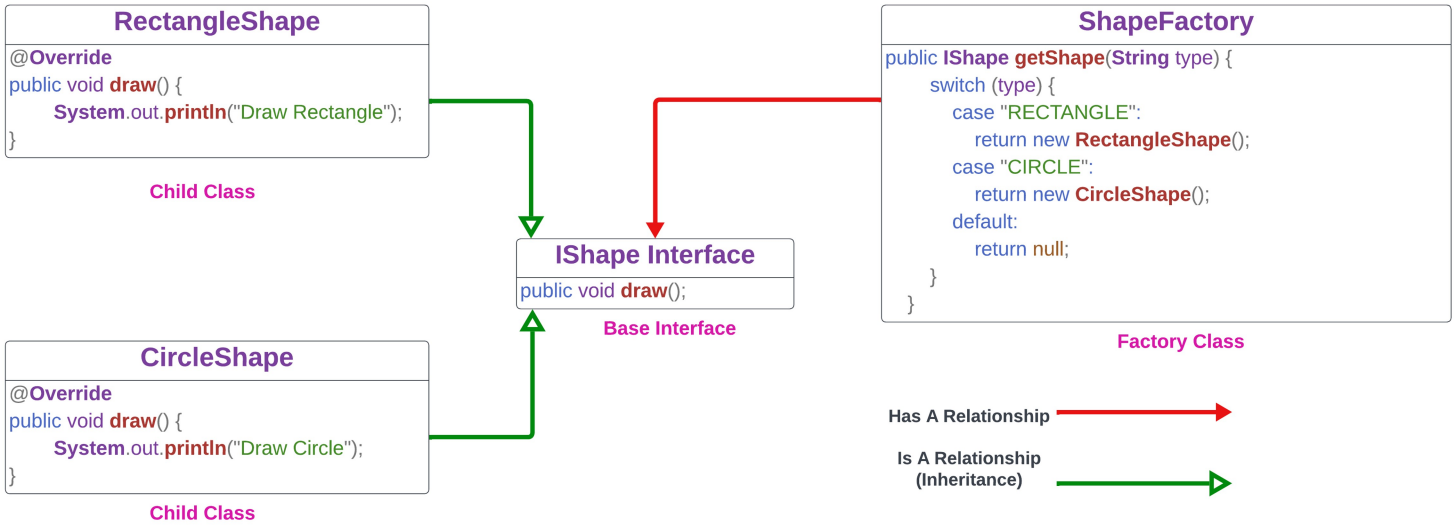
- Allows you to hide implementation of an application seam (the core interfaces that make up your application.
- Allows you to easily test the seam of an application (that is to mock/stub) certain parts of your application so you can build and test the other parts.
- Allows you to change the design of your application more readily, this is known as loose coupling.

Cons

- Makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions.
- Can be classed as an anti-pattern when it is incorrectly used, for example some people use it to wire up a whole application when using an IOC container, instead use Dependency Injection.

Example:

Create Shape classes based upon certain conditions as per ER diagram below:



Code:

Shape Interface and Concrete Classes:

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

```
public class RectangleShape implements IShape {
    @Override
    public void draw() {
        System.out.println(x:"Draw Rectangle");
    }
}
```

```
public class CircleShape implements IShape {
    @Override
    public void draw() {
        System.out.println(x:"Draw Circle");
    }
}
```

Factory Class:

```
public class ShapeFactory {
    public IShape getShape(String type) {
        switch (type) {
            case "RECTANGLE":
                return new RectangleShape();
            case "CIRCLE":
                return new CircleShape();
            default:
                return null;
        }
    }
}
```

CreateShapeMain Class:

```java
public class CreateShapeMain {
    Run | Debug
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        IShape rectangle = shapeFactory.getShape(type:"RECTANGLE");
        rectangle.draw();

        IShape circle = shapeFactory.getShape(type:"CIRCLE");
        circle.draw();
    }
}
```