Design Principles & Patterns

Sujeet Kumar Jaiswal

Principal Engineer

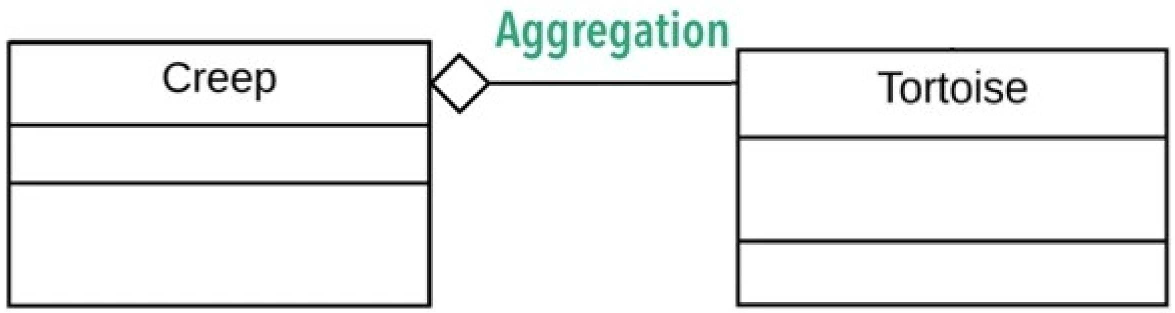
Logistics Team

UML Basics

- Inheritance notation
- Interface implementation notation

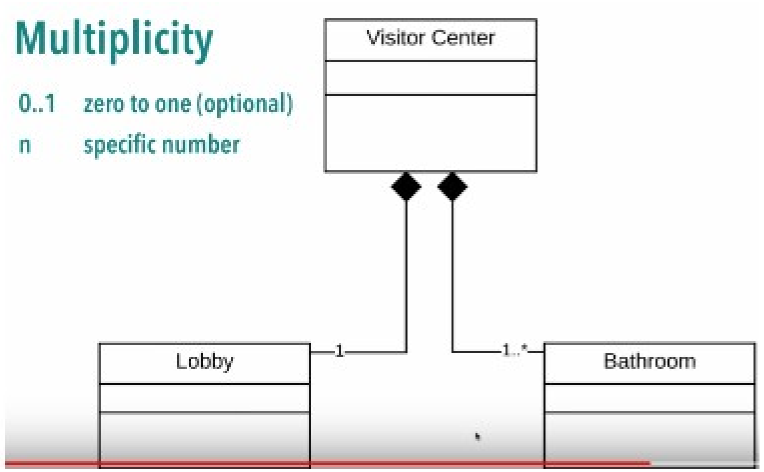
UML Basics - Aggregation notation

- A creep comprises of group of tortoises. Tortoise object (part of whole) can exist without Creep object (whole).



UML Basics - Composition notation

- Child object cannot exist without parent object. Orderdetails object cannot exist outside of order object. Another e.g. below -



Introduction – Design Principles

- These are core abstract principles which we are supposed to follow while designing software applicable irrespective of on any language/platform. Eg below -
  - Strive for loosely coupled designs between objects that interact (low coupling)
  - Single Responsibility Principle - A class should have only reason to change (high cohesion).
    - e.g. checkinventory() and calculatePrice() in one class should be avoided.

Design Principles - Encapsulate what varies

- Identify the parts of your application that varies and encapsulate them, so that later you can alter or extend those parts that vary without affecting those that don’t.
- Let's say you are developing an application for pets (dog, cats, duck etc...)
- Examine this client code -

```
if (pet.type() == dog) {
  pet.bark();
} else if (pet.type() == cat) {
  pet.meow();
} else if (pet.type() == duck) {
  pet.quack()
}
```

- Encapsulate the speaking behavior of each pet (pet.speak())

Design Principles - Favour composition over inheritance

- Creating systems using composition gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime
- Let's say you have a Car with children depending on its mode of power.
- Car
  - ElectricCar
  - DieselCar
  - GasolineCar
- Now, you'd like to extend this hierarchy to include the different types of cars: SUVs, sedans, and trucks.
  - Bad design - make nine different classes: ElectricSUV, ElectricSedan, ElectricTruck, DieselSUV, etc.This gets worse when you add even more dimensions, which is a common need in software design.
  - Good design - use composition: a Car might have an Engine object that defines how it generates power and a VehicleShape object that defines its shape. Engines and VehicleShapes can be subclassed independently of each other, and there's no combinatoric explosion of classes.

Design Principles – Low coupling

- Strive for loosely coupled designs between objects that interact (low coupling)
- When two objects are loosely coupled, they interact but have very little knowledge of each other. Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.
- eg Observer Pattern provides an object design where subjects and observers are loosely coupled.

Design Principles – Open/Closed principle

- Feel free to extend existing classes with any new behavior you like for your new requirement.
- We spent a lot of time getting this code correct and bug free, so we can’t let you alter the existing code. It must remain closed to modification. Eg Decorator Pattern, Observer Pattern
- You must concentrate on those areas that are most likely to change in your designs and apply the principles there.

Design Principles – Single Responsibility Principle

- A class should have only reason to change (high cohesion).
- Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change. This principle guides us to keep each class to a single responsibility. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

Introduction – Design Patterns

- Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.
- They are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Design Patterns Groups

- Creational Patterns
  - These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. Singleton, Builder, Prototype, Abstract Factory, Factory Method
- Structural Patterns
  - lets you compose classes or objects into larger structures. Decorator, Proxy, Facade, Composite, Bridge, Flyweight, Adapter
- Behavioral Patterns
  - is concerned with how classes and objects interact and distribute responsibility. Observer, Template Method, Visitor, Mediator, Iterator, Interpreter, Command, Memento, Chain of Responsibility, State, Strategy

Creational Pattern – Factory Method

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

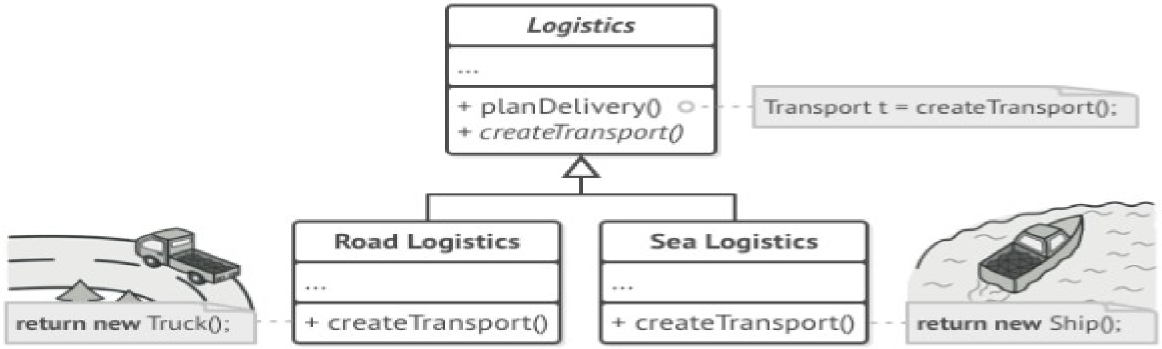
Creational Pattern – Factory Method : Scenario

- Imagine a logistics management application which handle transportation only by trucks. It becomes popular and now it needs to include sea transport as well.
- Bad news - most of the code is coupled with Truck class & adding Ships would require making changes to the entire codebase.
- We will end up with nasty code riddled with conditionals, which select behaviors depending on classes of transport objects.

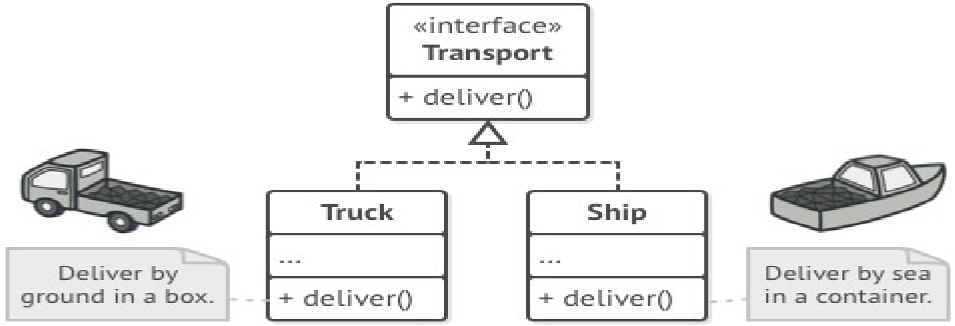
LogisticsApp

Creational Pattern – Factory Method : Solution

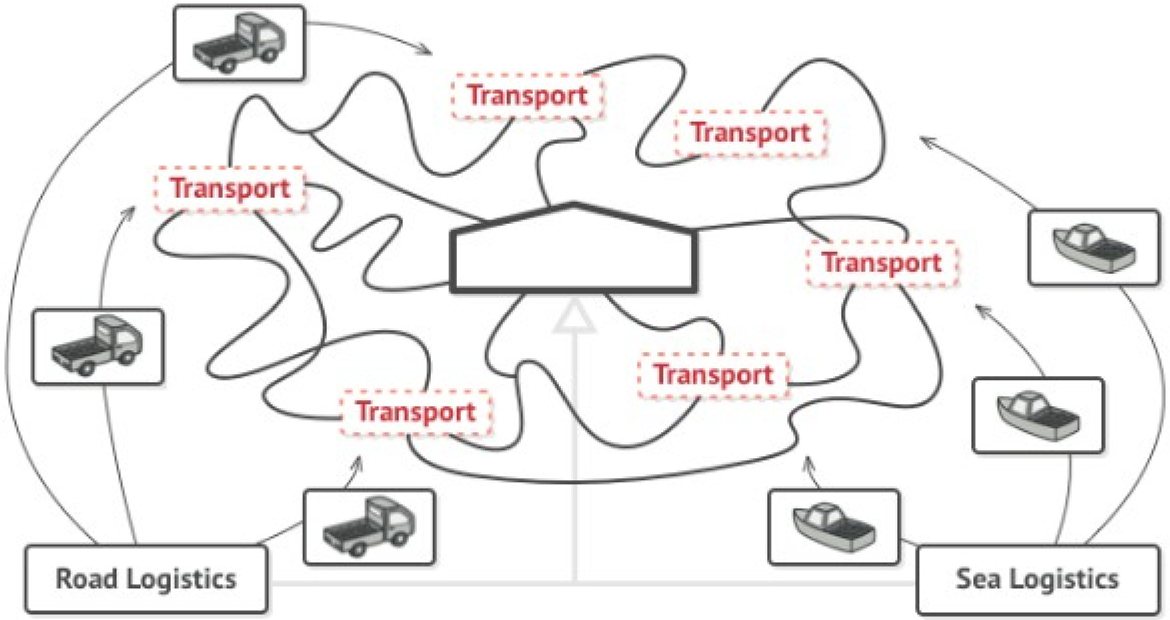
This pattern suggests replacing direct object creation with a call to a special "factory" method. The constructor call should be moved inside that method.



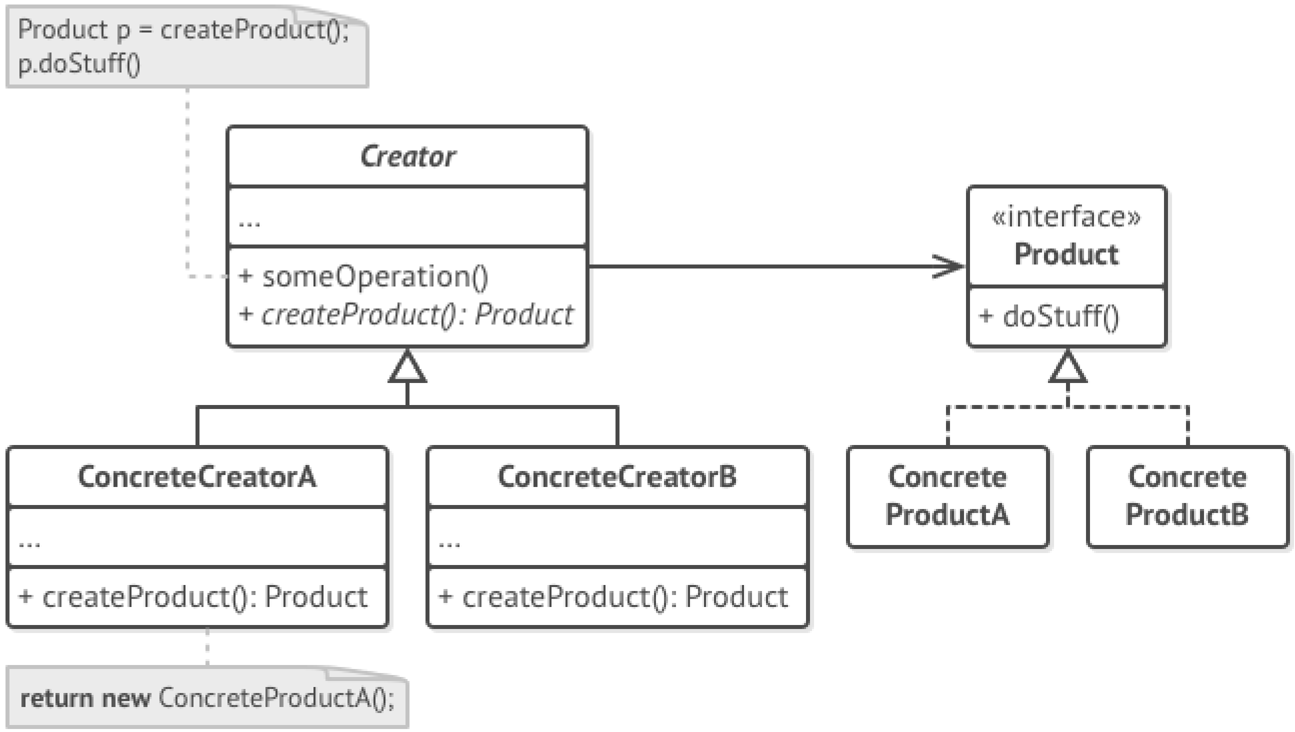
We can override the factory method in a subclass and change the class of an object that will be created.



Creational Pattern – Factory Method : Solution



Structural Pattern – Factory Method : Structure



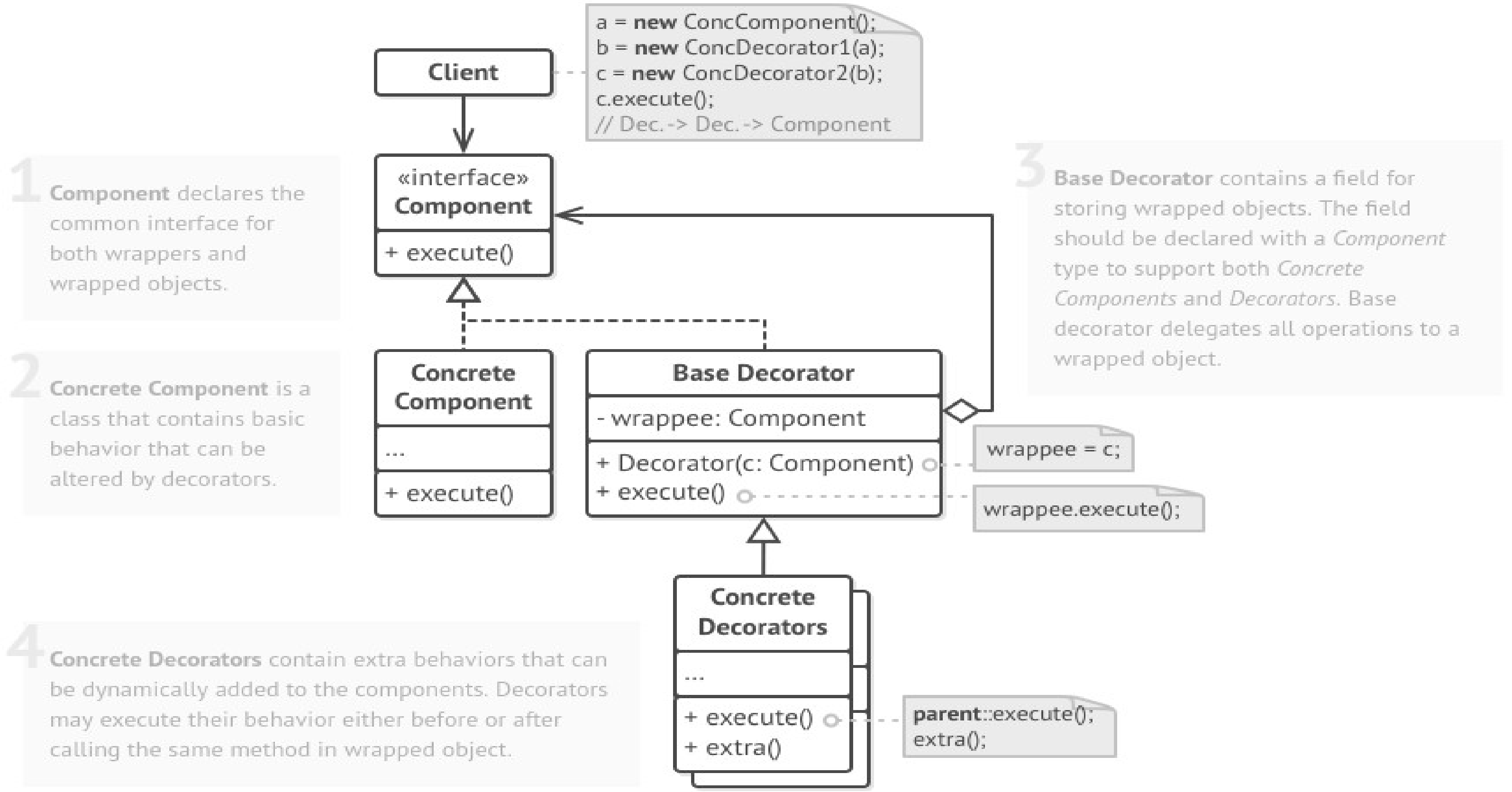
Structural Pattern – Factory Method : Code examples

- Checkout java code examples
- Checkout nodejs code examples

Structural Pattern – Decorator

- Intent - Decorator is a structural design pattern that lets you attach new behaviors to objects by placing them inside wrapper objects that contain these behaviors.
- Eg - GridServiceability cache key design
  - dest.pin, wids (sourcepin), pricerange, wtrange (high/low), volwtrange(high/low), shipperwisewtrange(high/low), category, merchantids, deliverytype etc..
- Checkout UML → decorator-pattern-gridserviceability-eg.jpg

Structural Pattern – Decorator : Structure



```
a = new ConcComponent();
b = new ConcDecorator1(a);
c = new ConcDecorator2(b);
c.execute();
// Dec. -> Dec. -> Component
```

1 **Component** declares the common interface for both wrappers and wrapped objects.

2 **Concrete Component** is a class that contains basic behavior that can be altered by decorators.

3 **Base Decorator** contains a field for storing wrapped objects. The field should be declared with a *Component* type to support both *Concrete Components* and *Decorators*. Base decorator delegates all operations to a wrapped object.

4 **Concrete Decorators** contain extra behaviors that can be dynamically added to the components. Decorators may execute their behavior either before or after calling the same method in wrapped object.

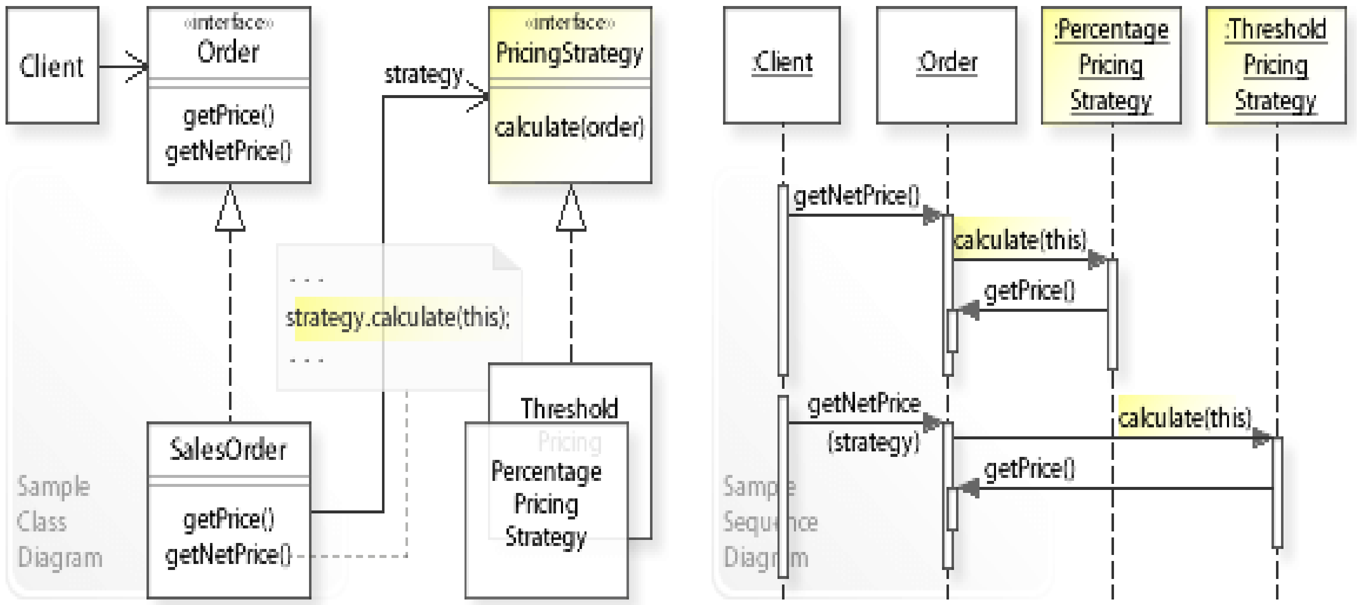
Structural Pattern – Decorator : Code examples

- Checkout java code examples
- Checkout nodejs code examples

Behavioral Pattern – Strategy

- Intent - Strategy is a behavioral design pattern that lets you define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Eg – Design a pricing engine which calculated price based on different pricing strategies decided by usertype. Lets say for user who are premium you want to give them flat x% discount and for normal user based on a threshold price of order.

Behavioral Pattern – Strategy : UML eg



Behavioral Pattern – Strategy : Structure

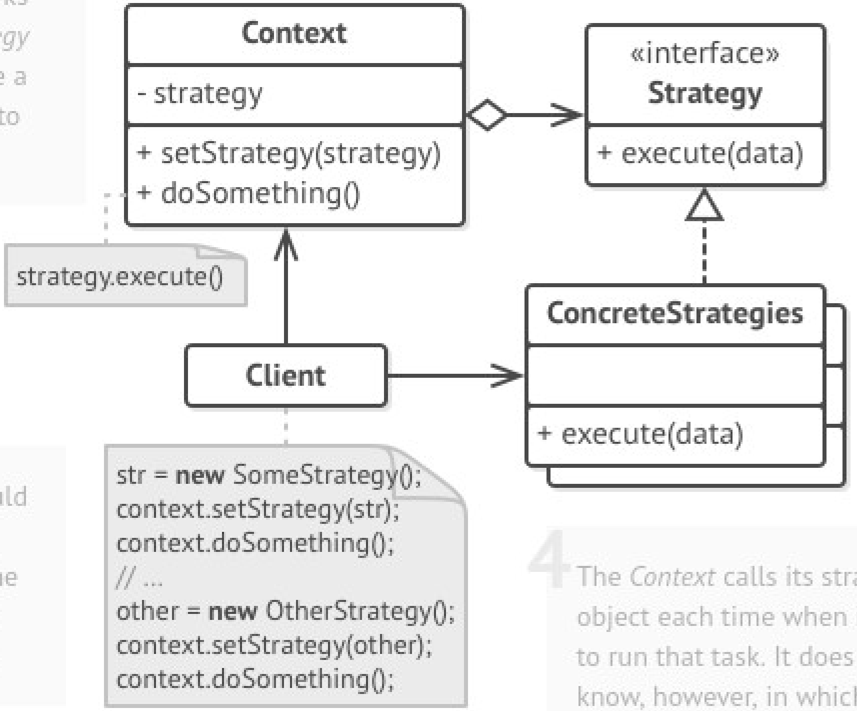
1 **Context** stores a reference to a *Concrete Strategy* object, but works with it through a common *Strategy* interface. Context should expose a setter that allows other objects to replace linked strategy object.

2 **Strategy** declares the common interface for all strategies. This interface makes concrete strategies interchangeable in *Context*.

3 **Concrete Strategies** implement different algorithms, which aim to accomplish the same task in various ways.

4 The *Context* calls its strategy object each time when it needs to run that task. It does not know, however, in which way the task will be executed.

5 **Clients** know what strategy should be picked depending on the situations. They can configure the *Context* with a different strategy whenever they want at runtime, using the setter.



```
str = new SomeStrategy();
context.setStrategy(str);
context.doSomething();
// ...
other = new OtherStrategy();
context.setStrategy(other);
context.doSomething();
```

Structural Pattern – Strategy : Code examples

- Checkout java code examples
- Checkout nodejs code examples

References

- Head First Design Patterns by Eric Freeman, Elisabeth Freeman with Kathy Sierra & Bert Bates.
- https://refactoring.guru/
- http://w3sdesign.com/index0100.php
- http://www.dofactory.com/javascript/design-patterns
- https://github.com/fbeline/Design-Patterns-JS

Thank you!

Sujeet Jaiswal
+91 99530 53952
sujeet.jaiswal@paytmmall.com