Strategy Design Pattern

Strategy design pattern is behavioral design pattern where we choose a specific implementation of algorithm or task in run time – out of multiple other implementations for same task.

The important point is that these implementations are interchangeable – based on task a implementation may be picked without disturbing the application workflow.

Introduction

Strategy pattern involves removing an algorithm from its host class and putting it in separate class so that in the same programming context there might be different algorithms (i.e. strategies), which can be selected in runtime.

Strategy pattern enables a client code to choose from a family of related but different algorithms and gives it a simple way to choose any of the algorithm in runtime depending on the client context.

Driven by Open/closed Principle

This pattern is based on **Open/closed principle**. We don't need to modify the context [closed for modification], but can choose and add any implementation [open for extension].

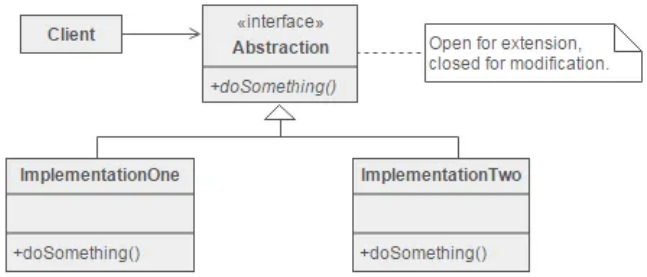
For example, in `Collections.sort()` – we don't need to change the sort method to achieve different sorting results. We can just supply different comparators in runtime.

> Read More: Comparator Example

Design Participants

In Strategy pattern, we first create an abstraction of algorithm. This is an interface having the abstract operation. Then we create implementations of this abstraction and these are called strategies.

A client will always call the abstraction, and will pass a context object. This context object will decide which strategy to use.



Strategy Pattern Participants

Problem Statement

Let's solve a design problem to understand strategy pattern in more detail.

I want to design a social media application which allows me to connect to my friends on all four social platforms i.e. Facebook, Google Plus, Twitter and Orkut (for example sake). Now I want that client should be able to tell the name of friend and desired platform – then my application should connect to him transparently.

More importantly, if I want to add more social platforms into application then application code should accommodate it without breaking the design.

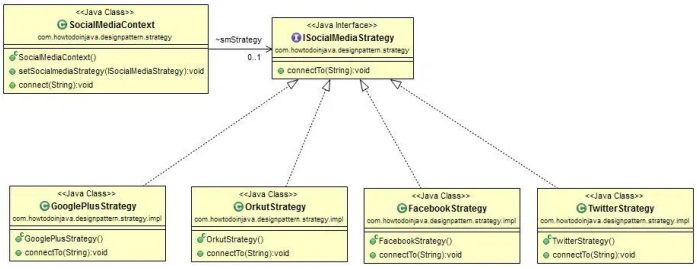
Solution with strategy design pattern

In above problem, we have an operation which can be done in multiple ways (connect to friend) and user can choose desired way on runtime. So it's good candidate for strategy design pattern.

To implement the solution, let's design one participant one at a time.

- **ISocialMediaStrategy** – The interface which abstract the operation.
- **SocialMediaContext** – The context which determines the implementation.
- **Implementations** – Various implementations of `ISocialMediaStrategy`. E.g. `FacebookStrategy`, `GooglePlusStrategy`, `TwitterStrategy` and `OrkutStrategy`.

Class Diagram



Strategy Pattern Class Diagram

Code Implementation

Now let's code above design participants.

ISocialMediaStrategy.java

```
package com.howtodoinjava.designpattern.strategy;

public interface ISocialMediaStrategy
{
    public void connectTo(String friendName);
}
```

SocialMediaContext.java

```
package com.howtodoinjava.designpattern.strategy;

public class SocialMediaContext
{
    ISocialMediaStrategy smStrategy;

    public void setSocialmediaStrategy(ISocialMediaStrategy s
    {
        this.smStrategy = smStrategy;
    }
```

```
    public void connect(String name)
    {
        smStrategy.connectTo(name);
    }
}
```

FacebookStrategy.java

```
package com.howtodoinjava.designpattern.strategy.impl;

import com.howtodoinjava.designpattern.strategy.ISocialMediaS

public class FacebookStrategy implements ISocialMediaStrategy

    public void connectTo(String friendName)
    {
        System.out.println("Connecting with " + friendName +
    }
}
```

GooglePlusStrategy.java

```
package com.howtodoinjava.designpattern.strategy.impl;

import com.howtodoinjava.designpattern.strategy.ISocialMediaS

public class GooglePlusStrategy implements ISocialMediaStrate

    public void connectTo(String friendName)
    {
        System.out.println("Connecting with " + friendName +
    }
}
```

TwitterStrategy.java

```
package com.howtodoinjava.designpattern.strategy.impl;

import com.howtodoinjava.designpattern.strategy.ISocialMediaS

public class TwitterStrategy implements ISocialMediaStrategy

    public void connectTo(String friendName)
    {
        System.out.println("Connecting with " + friendName +
    }
}
```

OrkutStrategy.java

```
package com.howtodoinjava.designpattern.strategy.impl;

import com.howtodoinjava.designpattern.strategy.ISocialMediaS

public class OrkutStrategy implements ISocialMediaStrategy {

    public void connectTo(String friendName)
    {
        System.out.println("Connecting with " + friendName +
    }
}
```

Demo

Now see how these strategies can be used in runtime.

```
package com.howtodoinjava.designpattern.strategy.demo;

import com.howtodoinjava.designpattern.strategy.SocialMediaCo
import com.howtodoinjava.designpattern.strategy.impl.Facebook
import com.howtodoinjava.designpattern.strategy.impl.GooglePl
import com.howtodoinjava.designpattern.strategy.impl.OrkutStr
import com.howtodoinjava.designpattern.strategy.impl.TwitterS

public class Demo {
    public static void main(String[] args) {

        // Creating social Media Connect Object for connectin
        // any social media strategy.
        SocialMediaContext context = new SocialMediaContext()

        // Setting Facebook strategy.
        context.setSocialmediaStrategy(new FacebookStrategy()
        context.connect("Lokesh");

        System.out.println("====================");

        // Setting Twitter strategy.
        context.setSocialmediaStrategy(new TwitterStrategy())
        context.connect("Lokesh");

        System.out.println("====================");

        // Setting GooglePlus strategy.
        context.setSocialmediaStrategy(new GooglePlusStrategy
        context.connect("Lokesh");

        System.out.println("====================");

        // Setting Orkut strategy.
        context.setSocialmediaStrategy(new OrkutStrategy());
        context.connect("Lokesh");
    }
}
```

Output:

```
Connecting with Lokesh through Facebook
====================
Connecting with Lokesh through Twitter
====================
Connecting with Lokesh through GooglePlus
====================
Connecting with Lokesh through Orkut [not possible though :)]
```

Popular Implementations

1. Java `Collections.sort(list, comparator)` method where client actually passes suitable comparator based on the requirement in runtime to the method and the method is generic to accept any comparator type. Based on the comparator being passed, same collection can be sorted differently.
2. Appenders, Layouts and Filters in Log4j.
3. Layout Managers in UI toolkits.

Summary

- This pattern defines a set of related algorithm and encapsulate them in separated classes, and allows client to choose any algorithm at run time.
- It allows to add new algorithm without modifying existing algorithms or context class, which uses algorithm or strategies
- Strategy is a behavioral pattern in Gang of Four Design pattern list.
- Strategy pattern is based upon Open Closed design principle of SOLID principals.
- Combination of Collections.sort() and Comparator interface is an solid example of Strategy pattern.

That's all about strategy design pattern. Drop me your questions in comments section.

Download Source Code

Happy Learning !!

HCFarver.dk

Share this:

Twitter Facebook LinkedIn Reddit

Feedback, Discussion and Comments

Abhijit

November 20, 2019

Thanks for the explanation. I built a similar solution for 2 implementation strategies of connecting to HTTP or S FTP . Do you think, this is a slightly better way of implementing Strategy Design pattern? The code that I wrote would not require the user to know about the name of the Implementation class. In the future if another implementation class gets added, the user would need not know the name of the class.

Here goes my code:

1. Strategy Interface

```
public interface IConnectionStrategy {

    /**Get Implementation type*/
    public boolean getTypeOfConnectionStrategy(String

    /** concrete method to test connection */
    public boolean testConnection();
}
```

2. Implementation 1 : HTTP

```
public class HTTPConnectionStrategy implements IConne
    @Override
    public boolean testConnection() {
        // establish connection code goes here
        return true;
    }
    @Override
    public boolean getTypeOfConnectionStrategy(String
        return (type.equals("HTTP"));
    }
}
```

2. Implementation 2: SFTP

```
public class SFTPConnectionStrategy implements IConne
    @Override
    public boolean testConnection() {
        // establish connection code goes here
        return true;
    }
    @Override
    public boolean getTypeOfConnectionStrategy(String
        return (type.equals("SFTP"));
    }
}
```

3. Strategy Context class

```
/**
 *Strategy Context to determine strategy at run time
public class StrategyContext {
    //externally initialized
    List<IConnectionStrategy> connexonStrategies;
    /**Test the connection based on type*/
    public boolean testConnection(String argType) {
        boolean isTestSuccessful=false;
        for (IConnectionStrategy iConnectionStrategy
            if(iConnectionStrategy.getTypeOfConnectio
                isTestSuccessful =  iConnectionStrate
            }
        }
        return isTestSuccessful;
    }
}

class TestClass{
    // Testing Connection
    public static void main(String[] args) {
        StrategyContext strategyContext = new Strateg
        strategyContext.testConnection("SFTP");
    }
}
```

blockbuster

May 13, 2020

it is not a better implementing because it break the “open-close” principle.once you add a new strategy you have to modify the existing code, for example, you have to add the “argType”.

Java Dev

December 22, 2018

Nice article on Strategy Pattern. In my blog post on the same pattern, I have also mentioned about how the client code would be without the application of this pattern – it will have a lot of conditional statements and we would be tying the individual implementations to the client code.

https://javadevcentral.com/strategy-design-pattern

Comments are closed on this article!

Meta Links

About Me
Contact Us
Privacy policy
Advertise
Guest and Sponsored Posts

Recommended Reading

10 Life Lessons
Secure Hash Algorithms
How Web Servers work?
How Java I/O Works Internally?
Best Way to Learn Java
Java Best Practices Guide
Microservices Tutorial
REST API Tutorial
How to Start New Blog