Seminar Report

Topic: Decorator and Strategy

Group: 3

Class: APCS2

Table of contents:

I. Decorator

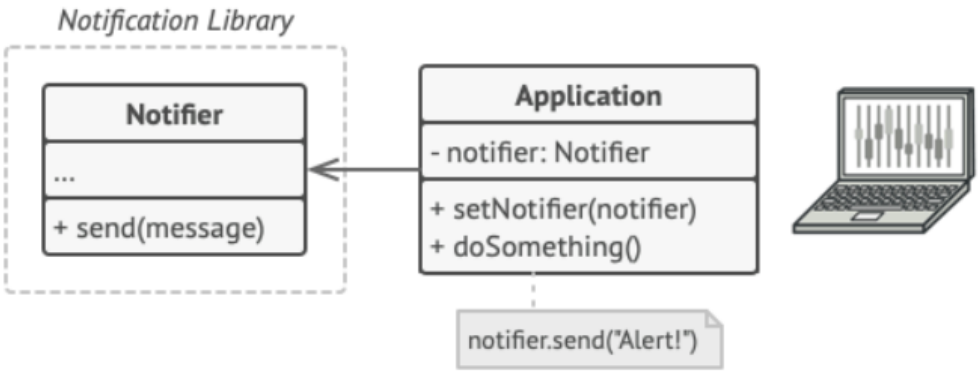
i. Introduction

Decorator is a structural design pattern that allows you to attach additional behaviors to objects by enclosing them in special wrapper objects that contain the behaviors.

ii. Problem

Assume you're working on a notification library that allows other applications to send out alerts to their users when crucial events occur.

The library's initial version was developed from around Notifier class, which had only a few properties, a constructor, and a single send function. The function may take a message parameter from the client and send it to a list of emails specified in the notifier's constructor. A client-side third-party program was expected to generate and setup the notifier object once, then utilize it every time something significant occurred.



A program could use the notifier class to send notifications about important events to a predefined set of emails. You'll eventually understand that library users want more than simply email alerts. Many of them would appreciate receiving an SMS alerting them to crucial situations. Others might like to be alerted on Facebook, while business users, of course, would prefer to get Slack notifications.

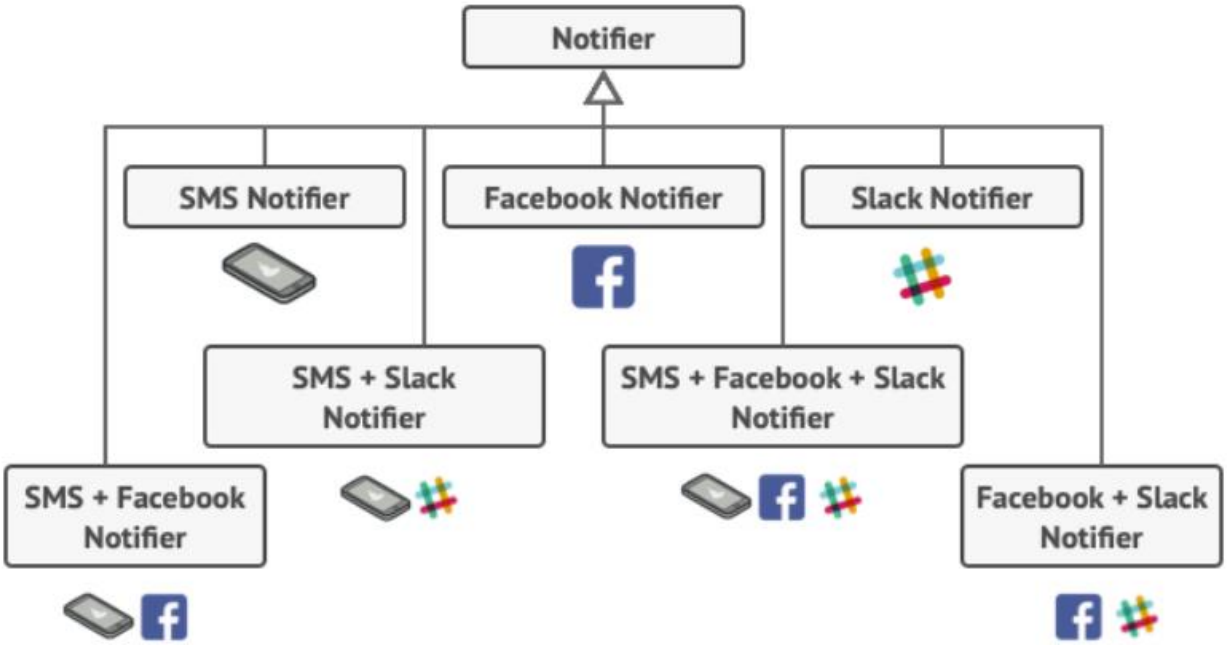
Each notification type is implemented as a notifier's subclass.

What's the worst that might happen? You expanded the Notifier class and created new subclasses to house the extra notification methods. The client was now expected to

create an instance of the appropriate notification class and utilize it for all future notifications.

But then someone questioned you, "Why can't you utilize many notification kinds at the same time?" If your house is on fire, you'd undoubtedly want to know about it via every available medium."

You attempted to solve the problem by designing special subclasses that included many notification methods into a single class. However, it became clear soon that this method would greatly bloat the code, not just the library code but also the client code.



Combinatorial explosion of subclasses.

You must find some other way to structure notifications classes so that their number won't accidentally break some Guinness record.

iii. Solution

When you need to alter an object's behavior, the first thing that springs to mind is to extend a class. However, there are a few important restrictions to be aware of when it comes to inheritance.

- Inheritance is a static. At runtime, you can't change the behavior of an existing object.
- Subclasses can only have one parent class, therefore you can only replace the entire object with another one built from a different subclass. In most

languages, inheritance prevents a class from inheriting several classes' behaviors at the same time.

Using Aggregation or Composition (Aggregation: object A includes object B; B may survive without A) is one technique to get around these limitations. Composition: object A is made up of objects B; A is in charge of B's life cycle; B cannot exist without A.) replacement of the word "inheritance" Both methods function in a similar way: one object has a reference to another and delegated work to it, but inheritance allows the object to do the work itself by inheriting the behavior from its superclass.

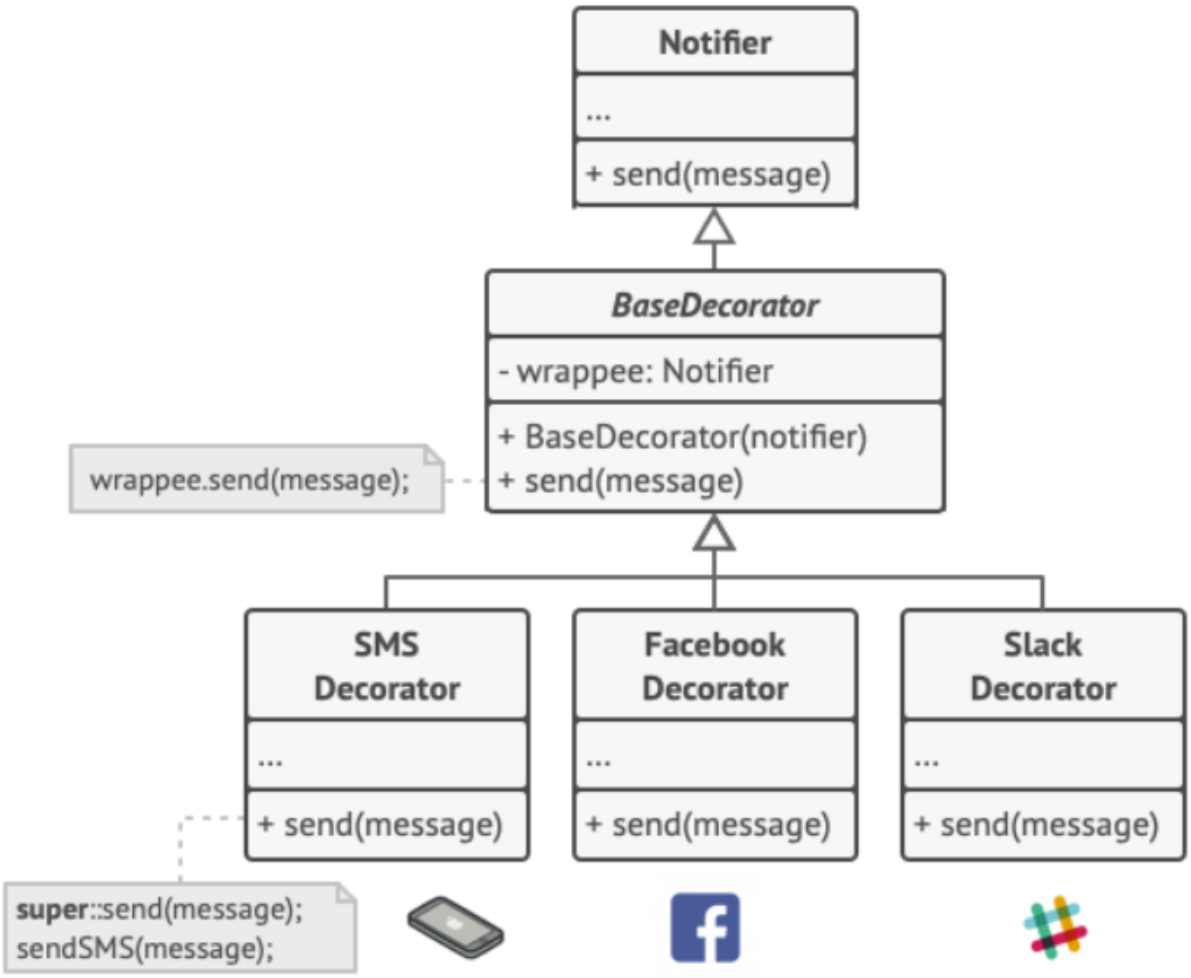
You may simply replace the linked "helper" object with another using this new technique, modifying the container's behavior at runtime. The behavior of several classes may be used by an object, which can have references to numerous objects and delegate tasks to them. Many design patterns, like Decorator, are based on the notion of aggregation/composition. Now, let's get back to the pattern discussion.

Inheritance vs. Aggregation

The alternate nickname for the Decorator pattern is "Wrapper," which clearly describes the pattern's fundamental principle. A wrapper is a type of object that may be associated to a specific target. All requests are delegated to the wrapper, which has the same set of methods as the target. The wrapper, on the other hand, can change the outcome by doing anything before or after passing the request to the destination.

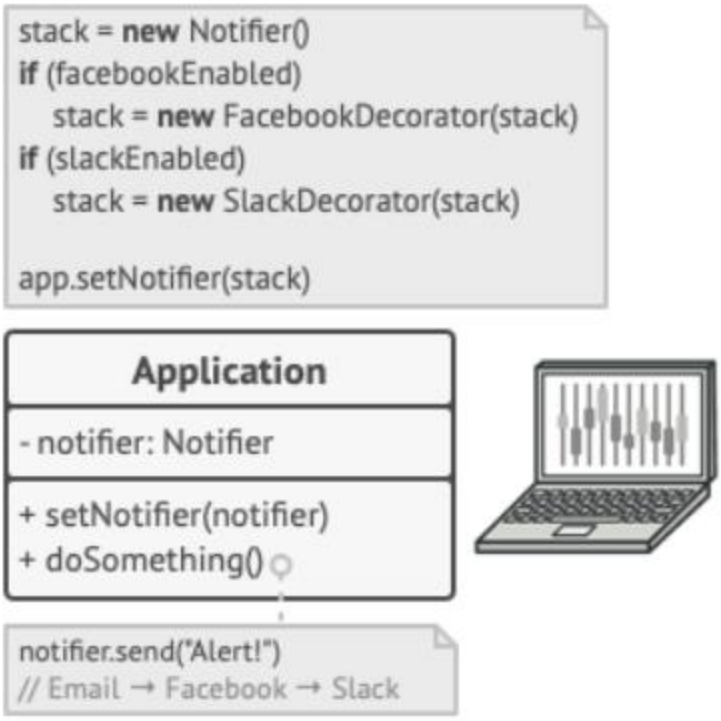
When does a simple wrapper stop being a simple wrapper and start being a true decorator? The wrapper, as previously stated, implements the same interface as the wrapped object. As a result, these objects appear to be identical to the client. Allow any object that follows that interface to be referenced in the wrapper's reference field. This will let you cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.

In our notifications example, let's leave the simple email notification behavior inside the base **Notifier** class, but turn all other notification methods into decorators.



Various notification methods become decorators.

The client code would need to wrap a basic notifier object into a set of decorators that match the client's preferences. The resulting objects will be structured as a stack.



Apps might configure complex stacks of notification decorators.

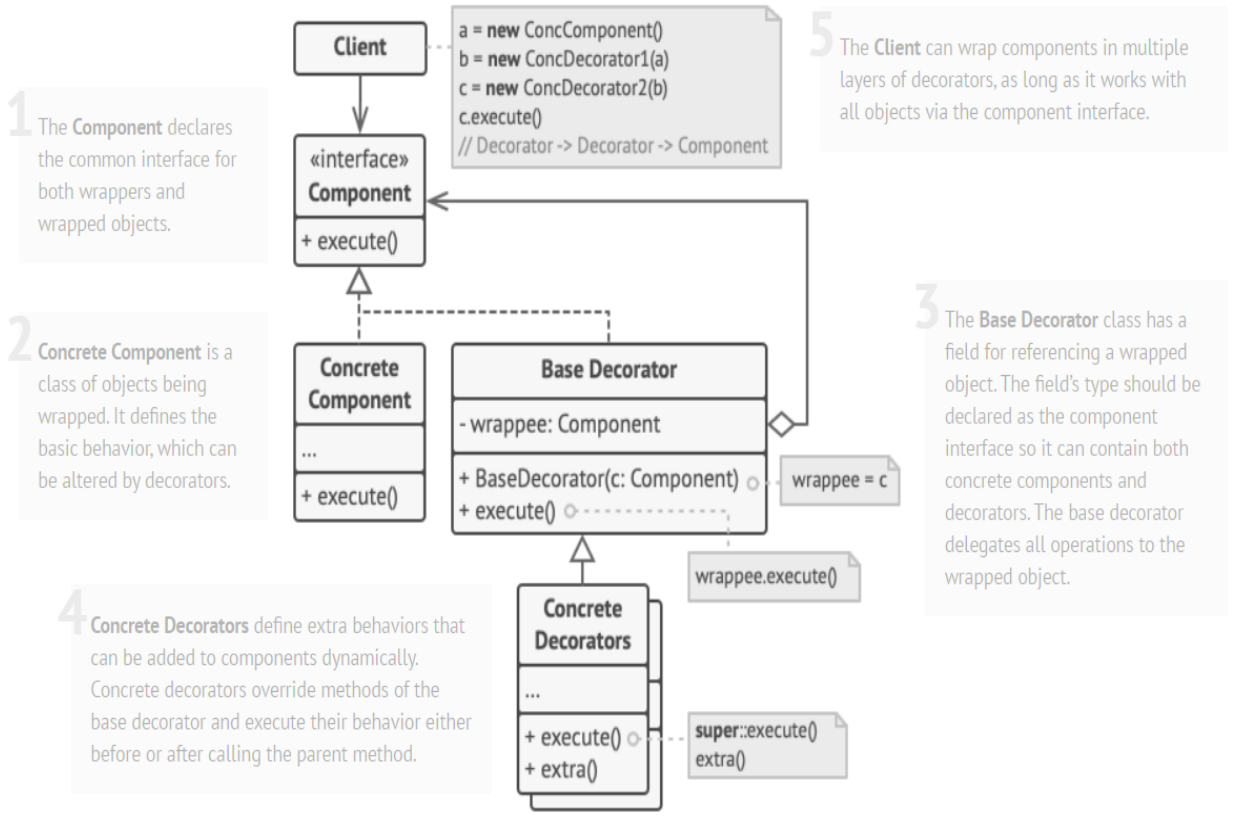
The client's actual working object would be the final decorator in the stack. Because all decorators implement the same interface as the underlying notifier, the remainder of the client code doesn't care if it's working with a "pure" or "decorated" notifier object.

Other activities, such as formatting messages or constructing the recipient list, might be approached in the same way. The client can use any custom decorators to decorate the object as long as they use the same interface as the others.

iv. Real-World analogy

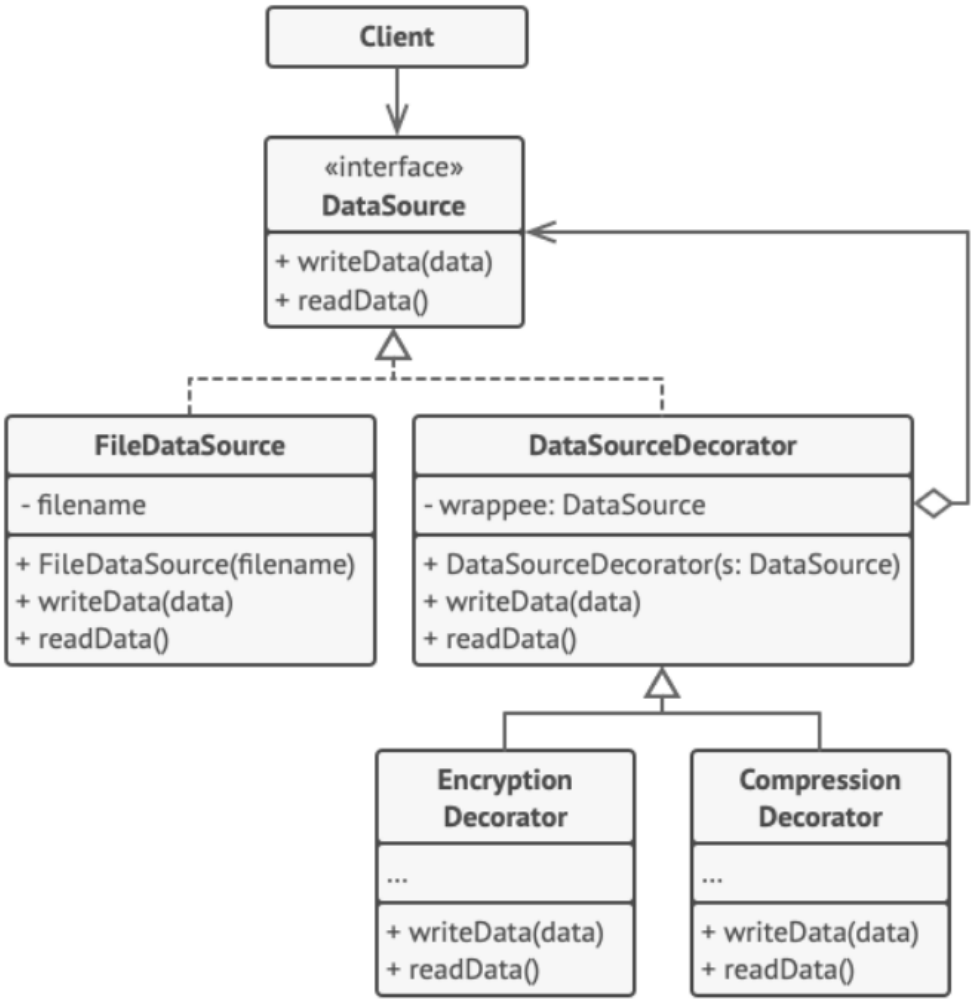
Using decorations in the form of clothing is an example. When it's cold outside, you put on a sweater. If you're still cold after wearing a sweater, layer a jacket on top. You can put on a raincoat if it's raining. All of these garments "extend" your fundamental behavior but are not a part of you, and you can readily remove any item of clothing when not in use.

v. Structure



vi. Pseudocode

In this example, the **Decorator** pattern lets you compress and encrypt sensitive data independently from the code that actually uses this data.



The encryption and compression decorators example.

The application wraps the data source object with a pair of decorators. Both wrappers change the way the data is written to and read from the disk:

- Just before the data is **written to disk**, the decorators encrypt and compress it. The original class writes the encrypted and protected data to the file without knowing about the change.
- Right after the data is **read from disk**, it goes through the same decorators, which decompress and decode it.

The decorators and the data source class implement the same interface, which makes them all interchangeable in the client code.

```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }
    method writeData(data) is
    method readData():data is

class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source
    method writeData(data) is
        wrappee.writeData(data)
    method readData():data is
        return wrappee.readData()

class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
    method readData():data is

class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
```

```
    method readData():data is

class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)
        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)
        source = new EncryptionDecorator(source)
        source.writeData(salaryRecords)



class SalaryManager is
    field source: DataSource

    constructor SalaryManager(source: DataSource) { ... }

    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)

class ApplicationConfigurator is
    method configurationExample() is
```

```
    source = new FileDataSource("salary.dat")
    if (enabledEncryption)
        source = new EncryptionDecorator(source)
    if (enabledCompression)
        source = new CompressionDecorator(source)


    logger = new SalaryManager(source)
    salary = logger.load()
// ...
```

vii. Applicability

Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.

The Decorator allows you to layer your business logic, define a decorator for each layer, and construct objects at runtime using various combinations of this logic. Because all of these objects have the same interface, the client programs could treat them all the same.

Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

The final keyword in several programming languages can be used to keep a class from being extended further. The only method to reuse existing functionality in a final class would be to wrap it in your own wrapper using the Decorator approach.

viii. Implementation

- Ensure that your business's domain may be represented as a major component with several optional layers layered on top of it.

- Determine which methods are shared by the principal component and optional layers. Build a component interface and add those methods to it.
- Create a concrete component class that contains the fundamental behavior.
- Build a decorator base class. It should provide a field for storing a wrapped object reference. To support linking to concrete components as well as decorators, the field should be specified with the component interface type. All work must be delegated to the wrapped object via the base decorator.
- Double-check that every class implements the component interface.
- Create concrete decorators by extending them from the base decorator. A concrete decorator must execute its behavior before or after the call to the parent method (which always delegates to the wrapped object).
- The client code must be responsible for creating decorators and composing them in the way the client needs.

ix. Pros and Cons

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

- It's hard to remove a specific wrapper from the wrappers stack.
- It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- The initial configuration code of layers might look pretty ugly.

x. Relations with other patterns

- **Adapter** changes the interface of an existing object, while **Decorator** enhances an object without changing its interface. In addition, *Decorator* supports recursive composition, which isn't possible when you use *Adapter*.
- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.
- **Chain of Responsibility** and **Decorator** have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.

The *CoR* handlers can execute arbitrary operations independently of each other. They can also stop passing the request further at any point. On the other hand, various *Decorators* can extend the object's behavior while keeping it consistent with the base interface. In addition, decorators aren't allowed to break the flow of the request.

- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.

  A *Decorator* is like a *Composite* but only has one child component. There's another significant difference: *Decorator* adds additional responsibilities to the wrapped object, while *Composite* just "sums up" its children's results.

  However, the patterns can also cooperate: you can use *Decorator* to extend the behavior of a specific object in the *Composite* tree.

- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
- **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts.

  Decorator and **Proxy** have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a *Proxy* usually manages the life cycle of its service object on its own, whereas the composition of *Decorators* is always controlled by the client.

xi. Reference

https://refactoring.guru/design-patterns/decorator

II. Strategy

i. Introduction

An example of a software design pattern is one that may be used to solve a problem that happens often in a certain setting in software design, which is defined as follows: Initial classifications of design patterns were divided into three sub-categories depending on the kind of issue they were intended to answer. Objects

may be created in a controlled manner using creational patterns that are based on a set of criteria that have been specified. Organization of distinct classes and objects into bigger structures and the addition of additional functionality are the goals of architectural patterns. To sum up, behavioral patterns are concerned with establishing common communication patterns between items and then putting these patterns into action.

Multiple design patterns are used to create the behavioral pattern, such as the Blackboard design pattern, the Command pattern, and so on... Alternatively, our theme – Strategy Pattern.

Consequently, the strategy pattern (also known as the policy pattern) is a behavioral software design pattern that allows for the selection of an algorithm at runtime in software applications. Instead of explicitly executing a single algorithm, code gets run-time instructions as to which algorithm in a family of algorithms to utilize, which is then implemented. The short version is that you may build a family of algorithms, place each of them in a distinct class, and make their objects interchangeable by using this pattern.

ii. Problem

Suppose that we set up an online shopping platform. That place allows every customer to choose the product they want and proceed to checkout after selecting all they need.

However, different customers used this system, and they had different options to pay the money: one wanted to pay by Credit card; one wanted to pay by Debit card; another one wanted to pay in cash after receiving the goods. It's easy to imagine that a third or fourth one will also have different needs.

As a result, if the programmer continues to add solutions to the main class, the class's size will double for each circumstance. After a certain point, the beast got too difficult to keep up. Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the delivery logic, affected the whole class, increasing the chance of creating an error in already-working code.

As a result, collaboration became less effective. A few of your new hires have complained about the amount of time spent on merge conflicts after the successful

release. New features need changing the same large class, which conflicts with the code written by other developers.

iii. Solution

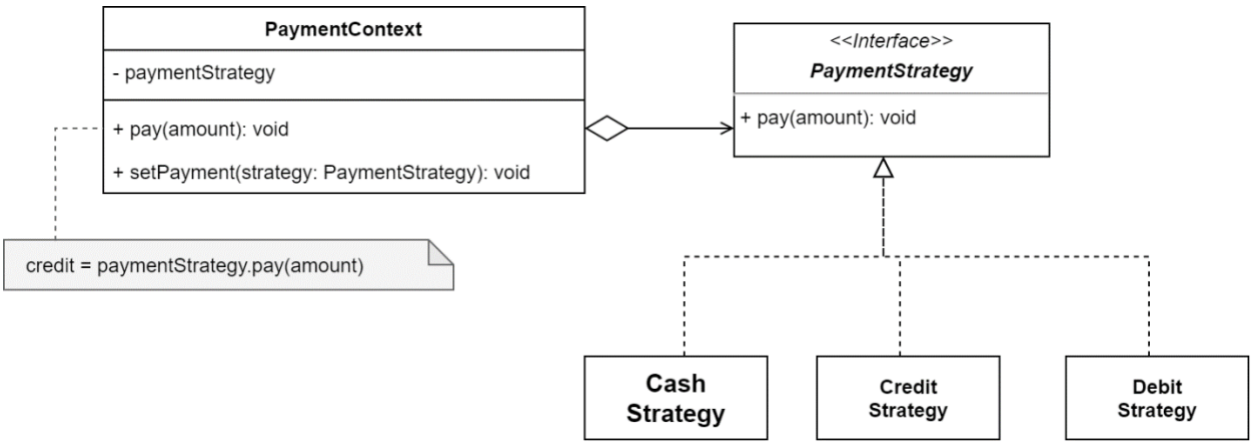
It is suggested that you take a class that does a single task in a variety of different ways and break all of the algorithms into independent classes called strategies, according to the Strategy pattern.

A field for holding a reference to one of the strategies must be included in the original class, which is called context. Instead of doing the job itself, the context delegated it to a related strategy object, which then completed the task.

The context isn't in charge of determining which algorithm is best suited for the task at hand. Instead, the client instructs the context to execute the chosen approach. In reality, the context is clueless when it comes to strategic thinking. Regardless of the strategy, it communicates with it using the same generic interface, which exposes just one single method for activating the algorithm enclosed inside the specified strategy.

As a result, the context becomes independent of particular strategies, allowing you to add new algorithms or alter current ones without having to change the code of the context or any other strategies in the system..

So, for a particular task (problem), there will be multiple solutions and from the solutions, the user has to choose only one solution at runtime. So, in this example paying money is the task and three options (Credit, Debit, and Cash) to pay the money and customer will choose only one option at the time of billing.



In our payment system, each paying algorithm can be extracted to its own class with a single pay method.

Although given the same arguments, each payment class might have its own method, the main PaymentContext class doesn't really care which algorithm is. The class has a method for switching the active payment strategy, so its clients, such as the buttons in the user interface, can replace the currently selected paying behavior with another.

//Source Code:

```
#include <iostream>

using namespace std;

class paymentStrategy
{
public:
   virtual ~paymentStrategy() {
      /* ... */
   }
   virtual void Pay(double amount) = 0;
};

class ConcreteStrategyCredit : public paymentStrategy
{
public:
```

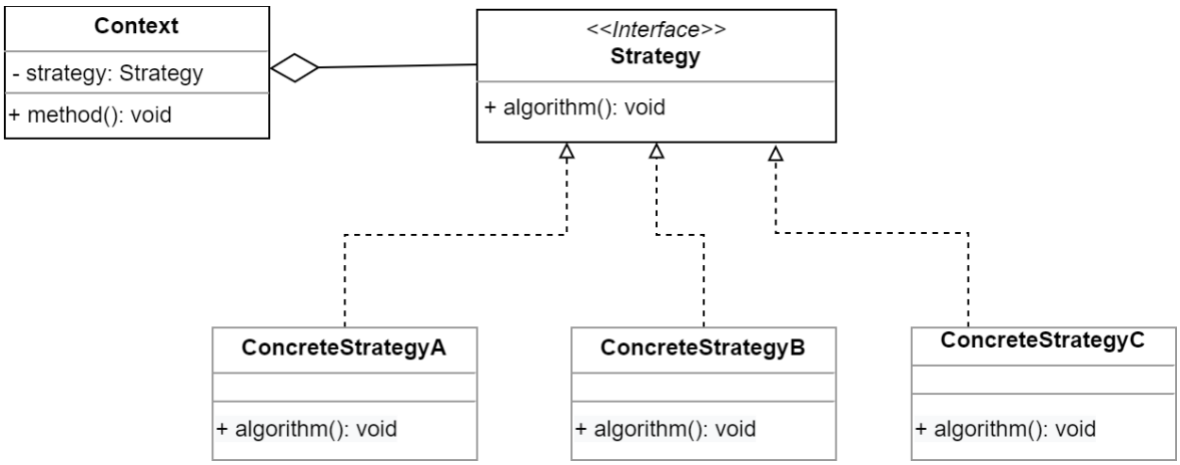
```
    ~ConcreteStrategyCredit() {
        /* ... */
    }

    void Pay(double amount)
    {
        cout << "Customer pays Rs " << amount << " using Credit Card" << endl;
    }
};

class ConcreteStrategyDebit : public paymentStrategy
{
public:
    ~ConcreteStrategyDebit() {
        /* ... */
    }

    void Pay(double amount)
    {
        cout << "Customer pays Rs " << amount << " using Debit Card" << endl;
    }
};

class ConcreteStrategyCash : public paymentStrategy
{
public:
    ~ConcreteStrategyCash() {
        /* ... */
    }

    void Pay(double amount)
    {
        cout << "Customer pays Rs " << amount << " using Cash" << endl;
    }
};

class PaymentContext
```

```
{
public:
   PaymentContext( paymentStrategy* const s ) : strategy( s ) {}

   ~PaymentContext()
   {
      delete strategy;
   }

   void Pay(double amount)
   {
      strategy->Pay(amount);
   }

private:
   paymentStrategy *strategy;
};

int main()
{
   PaymentContext context1( new ConcreteStrategyCredit() );
   context1.Pay(300000);

   PaymentContext context2( new ConcreteStrategyDebit() );
   context2.Pay(200000);

   PaymentContext context3( new ConcreteStrategyCash() );
   context3.Pay(500000);

   return 0;
}
```

OUTPUT:

```
Customer pays Rs 300000 using Credit Card
Customer pays Rs 200000 using Debit Card
Customer pays Rs 500000 using Cash
```

iv. Implementation



Strategy – defines an interface common to all supported algorithms. In order to invoke the algorithm specified by ConcreteStrategy, Context makes use of this interface.

The Context keeps a reference to the concrete strategies and interacts with this object solely via the interface for the concrete strategies. When an operation is necessary, the algorithm is invoked from the strategy object, which is subsequently executed. The Context is completely unaware of the execution of the plan. Additional objects may be added to convey data from the context object to the strategy, if this is required.

Concrete Strategies are distinct forms of an algorithm that is used in the context of the strategy.

Requests from the client are received by the context object, which then passes them on to the strategy object. Client-developed Concrete Strategy that is supplied to the context is most often used. From this point forward, the client's interactions are limited to the context.

//Pseudocode

```
interface Strategy is
    method algorithm()
```

```
class ConcreteStrategyA implements Strategy is
    method algorithm() is
        ...

class ConcreteStrategyB implements Strategy is
    method algorithm() is
        ...

class ConcreteStrategyC implements Strategy is
    method algorithm() is
        ...

class Context is
    private strategy: Strategy

    method setStrategy(Strategy strategy) is
        this.strategy = strategy

    method someMethod() is
        return strategy.algorithm()

class ExampleApplication is
    method main() is
        Create context object.
```

```
Context context( new ConcreteStrategyA() )
context.contextInterface()

Context context1( new ConcreteStrategyB() )
context1.contextInterface()

Context context2( new ConcreteStrategyC() )
context2.contextInterface()

Print result.
```

v. Pros and cons of Strategy pattern

1. Pros

- Prevents the conditional statements (if – else, switch, …)
- A loose connection exists between the algorithms and their corresponding context item. They have the ability to be updated or replaced without affecting the context object.
- At runtime, you may change the algorithms that are employed in an object.
- Code that utilizes an algorithm may be separated from the implementation specifics.
- The composition may be used in lieu of inheritance.
- Open/Closed Principle. You can introduce new strategies without having to change the context.
- Allows you to have more options. Clients will have more choices at their disposal.
- Ensures that no code is rewritten.

2. Cons

- If you just have a number of algorithms that seldom change, there isn't a compelling need to overcomplicate the program by adding additional classes and interfaces that come with a pattern.

- Clients must be aware of the presence of many tactics, and they must be aware of the differences between the various methods.
- There is an increase in the number of objects that are created in the system. In certain edge instances, this might result in an increase in overhead. Numerous functional type support features are built into current programming languages, allowing you to implement several variants of an algorithm inside a single set of anonymous functions. Then you could use these methods in the same way that you would have used the strategy objects, but without having to add any more classes or interfaces to your codebase.
- Some methods specified by the strategy may need arguments to be given to them in certain circumstances. Although not all strategies make use of all parameters, they must be produced and sent via the system.
- Communication overhead between Strategy and Context. To do this, the Strategy base class must offer an interface for all essential behaviors.
- The application needs to create and maintain two objects in the place of one since the Context and Strategy object need to be configured together.

vi. Some real-world problems

Use the Strategy pattern whenever:

- Many closely similar classes are only distinguished by their behavior. Strategies is a technique of configuring a class with one of a variety of different behaviors..
- It is necessary to have many alternative variations of an algorithm. For example, you may create algorithms that represent various trade-offs between space and time. When these versions are implemented as a class hierarchy of algorithms, it is possible to use strategies.
- It is useful when you wish to have a set of behaviors that are not necessarily relevant to the client class.
- A class specifies many behaviors, which appear as a large number of conditional statements in its actions. Place closely similar conditional branches into their own Strategy class, rather than cramming them all into one.

Strategy pattern has been applied to many real-world problems as:

- Games: Currently, we want the player to be able to walk or run, but in the future we may want him to be able to swim, teleport, burrow underground, and so on.
- Sorting: It is necessary for us to sort these numbers, but we aren't sure if we will use BrickSort, BubbleSort, or another sorting algorithm. You could have an application in which the sorting method (QuickSort, HeapSort, and so on) is selected at runtime (for example, a database).
- Validation: We must examine stuff in accordance with "some rule," but it is not yet apparent what that rule will be, and we may come up with new rules in the process.
- Storing information: It is desirable for the program to keep information in the database, but it may also need to be able to save a file or make a web call in the future.
- Outputting: We need to output X as a plain string, but later may be a CSV, XML, JSON, etc.

vii. Reference

1. https://refactoring.guru/design-patterns/strategy
2. https://en.wikipedia.org/wiki/Strategy_pattern
3. https://sourcemaking.com/design_patterns/strategy