Strategy

There is a special pattern named **Strategy**. It's one of the most used patterns in object-oriented design. It defines a family of algorithms, encapsulates each in a separate class and makes them interchangeable within that family.

Introducing the strategy pattern

In **Strategy** pattern, algorithms executed in different branches are moved into their own classes called strategies, all of which implement a common interface. Strategies represent behavior, not domain entities.

The pattern has several advantages:

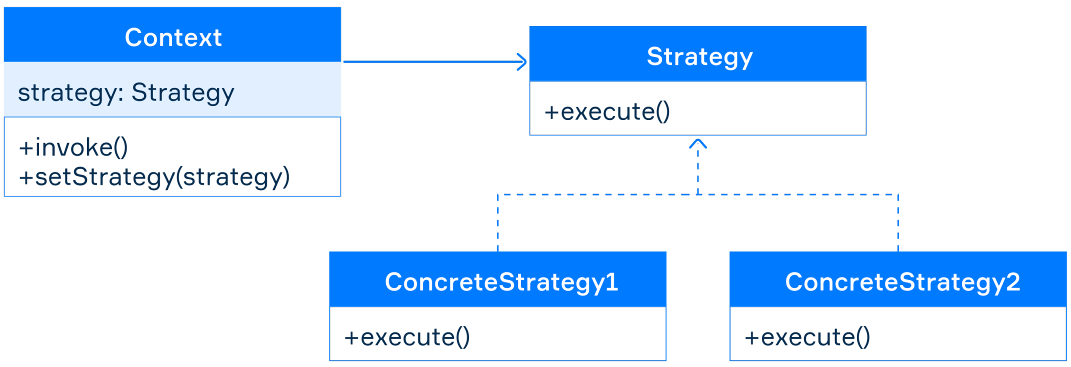
- it allows you to choose an algorithm (behavior) at runtime;
- it isolates the code of algorithms from the other classes, thus simplifying the addition of new algorithms;
- it lets the algorithm vary independently from clients that use it.

To use the pattern, you should take the following steps:

1. Determine the common Strategy interface for a family of algorithms. It should contain one or more abstract methods.
2. Extract all algorithms into their own classes (concrete strategies). They should follow the Strategy interface.
3. Declare a special class called Context for storing a reference to a strategy. The context delegates execution to an instance of a concrete strategy through its interface, instead of implementing the behavior itself.

Note that the interface here is not necessarily the interface in java. It can be a simple or an abstract class. The main thing is that it represents an abstract strategy that is inherited by concrete strategies.

The following diagram illustrates all elements of the pattern and their relations:



An example: sending messages
Suppose there's an application that needs to send messages to customers. There are different ways to reach out to them: via SMS or e-mail. In addition, new sending methods will be added in future (for instance, push notifications). It would be nice to change the existing code as little as possible when adding new strategies.
According to the strategy pattern, we need to define a family of algorithms (sending methods). Each algorithm must encapsulate logic to send a message using a concrete transport (SMS/email).
Here is our hierarchy of sending methods:

```
interface SendingMethod {

   void send(String from, String to, String msg);
}

class SmsSendingMethod implements SendingMethod {

   @Override
   public void send(String from, String to, String msg) {
      System.out.println(String.format("send SMS from '%s' to '%s'", from, to));
   }
}

class EmailSendingMethod implements SendingMethod {

    @Override
    public void send(String from, String to, String msg) {
        System.out.println(String.format("Email from '%s' to '%s'", from, to));
    }
}
```

Our Context is MessageSender which references a sending method and allows us to change the currently used method:

```
class MessageSender {

    private SendingMethod method;

    // it may contain additional fields as well

    public void setMethod(SendingMethod method) {
        this.method = method;
    }

    public void send(String from, String to, String msg) {
        this.method.send(from, to, msg);
    }
}
```

In the client code, we should create an instance of MessageSender, set a sending method and invoke the method *send* with three string arguments. Also, we can change the sending method to another one.

```
MessageSender sender = new MessageSender(); // create a message sender

sender.setMethod(new EmailSendingMethod()); // set a concrete sending method

sender.send("alice@gmail.com", "bob@gmail.com", "Hello!");

sender.setMethod(new SmsSendingMethod()); // set another sending method

sender.send("1-541-444-3333", "1-541-555-2222", "Hello!");
```

After starting, it outputs:

```
Email from 'alice@gmail.com' to 'bob@gmail.com'
send SMS from '1-541-444-3333' to '1-541-555-2222'
```

Complicating the strategy pattern

For different reasons, you will probably meet more complex implementations of

the strategy pattern.
1) Field and constructors. A concrete strategy may need its own settings stored in fields. Here is a class PushSendingMethod with a setting field which we fill in the constructor:

```
class PushSendingMethod implements SendingMethod {

   private final boolean magicFlag;

   public PushSendingMethod(boolean magicFlag) {
      this.magicFlag = magicFlag;
   }

   @Override
   public void send(String from, String to, String msg) {
      System.out.println(String.format("Send push from '%s' to '%s'", from, to));
   }
}
```

We can set the value when creating an instance of the method:

```
sender.setMethod(new PushSendingMethod(true));
```

2) Different and unused arguments. Different strategies may need different arguments or use not all arguments defined in the overridden method. In addition, new strategies may need other arguments which will require a change in the entire hierarchy.
The simplest way to solve the problem is to pass an aggregate type containing the necessary parameters. Each strategy will use only the required fields of the aggregate.
Here is a rewritten interfaceSendingMethod. The method *send* takes an instance of Message that contains different fields. Fields are public just for short.

```
interface SendingMethod {

   void send(Message message);
}

class Message {

   public String from;
   public String to;
   public String title;
   public String text;
   public byte[] attachment; // attachment data just like an image or something
else
}
```

3) Multiple methods. A strategy may have multiple methods. For instance, PaymentStrategy can contain methods to pay for an order and return the money. It's possible to separate the strategy into two independent ones. But sometimes, it's better to have one strategy with multiple methods.

Conclusion

The **Strategy** pattern is useful when we have multiple algorithms for performing a task and we want to choose any of the algorithms at runtime. This pattern is especially good when the number of algorithms increases because it isolates algorithms from the client code. However, this design pattern complicates the system by adding multiple additional classes.