**Strategy** is a behavioural design pattern that lets you define a family of algorithms, places each into a separate class, and interchange their objects.

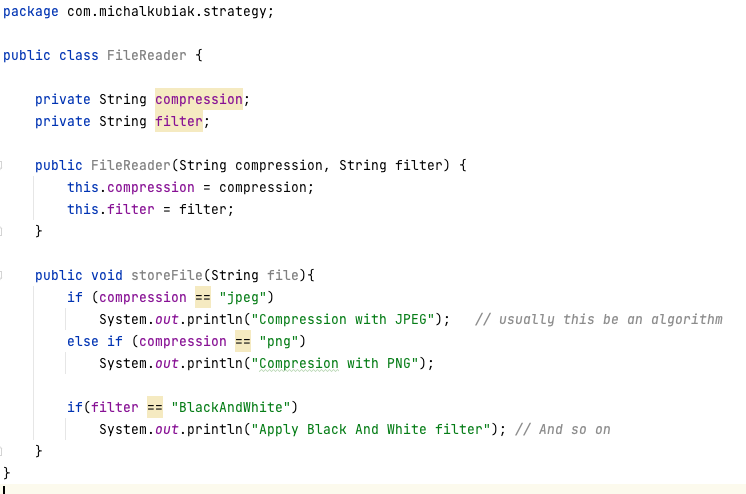
**The Pros** of Strategy implementation details are isolated from the code that uses them, inheritance can be replaced with composition, and the **Open Close Principle** can introduce new strategies without changing the context. **The Cons** are that your clients need to be aware of the differences between strategies to select a proper one. If you have a couple of algorithms that rarely change, then there is no reason to overcomplicate the code with new class interfaces that come along with the Strategy Pattern.

Let’s say in our application, we want a user to store files or upload pictures. We created a new package under a new branch which we pushed to GitHub. The package contains a new class called FileReader.

The first thing we would usually do is compress that file. There are different types of image compression algorithms, such a JPEG, for example. Then, we can apply a filter, so all images have the same look, for example, a black and white filter.

We need a few private fields which determine the compression type and the filter we will use. The first field will be a String called compression, it would be better to use an Enum, but for simplicity, I went with String to demonstrate this Design Pattern. Another field was created called filter also of type String.

These fields can be initialized using a constructor, after which we can check the compression and filter types in the storeFile method. Next, we can apply a filter and system out print a corresponding message. Instead of the messages, in an actual application, we would use algorithms. This hypothetical class demonstrates the initial problem and applies a **Strategy Design Pattern** to make the class more flexible.

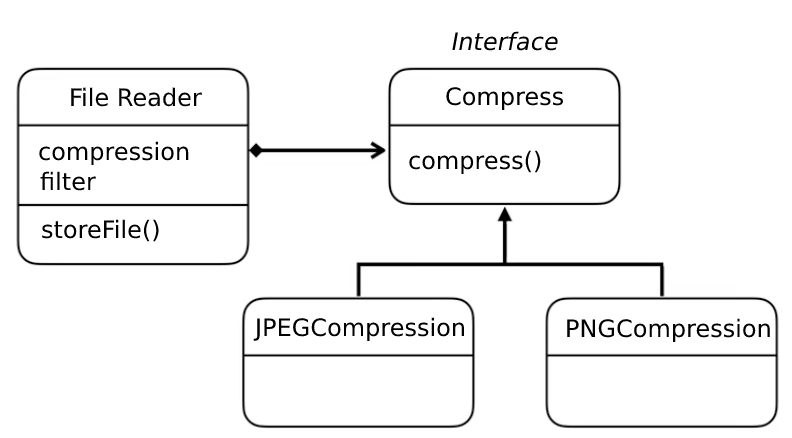


There are several problems with this code or this example of a class in an application. This class is violating the **Single Responsibility Principle** because this implementation is responsible for storing the file. It is also responsible for checking the compression type and checking the filter or applying one. Instead of the SOUT, we would have an algorithm there to compress a file, which would be a lot of lines of code. As this class gets bigger, it becomes unmaintainable. Which means we are violating the SRP principle.

The second issue is that it would be hard to support new compression types or filters. We would have to add more if statements.

To solve these problems, we need to apply Polymorphism. We want our file reader to behave differently depending on the compression or filter type we use. We can introduce a Compression interface with a method compress. Then we create classes that implement this interface, such as JPEG and PNG compression classes. With this structure, we are following the **Open Close Principle.** If we want to add a new compression type, we just make a new class that implements the Compression interface. In our FileReader class, we would change the compression type to Compressor to maintain a reference to the compressor object.

The same problem would be used for the filtering, the BlackAndWhite would implement the Filter interface, which would then be referenced as a type in the FileReader class. When we want the FileReader to maintain a reference to a Compress object, we give it a concrete Compress object such as JPEGCompression or PNGCompression.



In the other case, the filtering would be the same. The Compress interface would be called Filter. It would have a method called applyFilter(), which would be implemented by BlackAndWhite or any other class which dealt with filtering.

This is what we call the Strategy Pattern. The Gang of Four book has a structure of the Strategy Pattern while is the same as the one above. Only the FileReader is ‘Context’, the Interface is the ‘Strategy’, and the JPEGCompression would be a ‘ConcreteStrategy’.

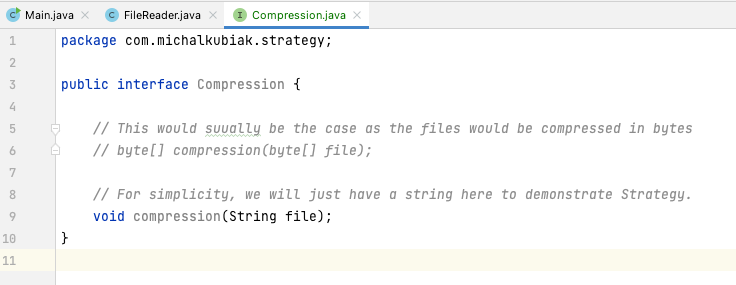
The context class must maintain a reference to a Strategy object. The strategy is an interface *or* an abstract class representing code or algorithm, and different classes implement it. It is similar to the State Pattern, both are structurally similar, and both are used to change the behaviour of an object.

The difference between the State and Strategy Pattern is that the FileReader object can have a single state in the State Pattern. All the behaviours are represented by a subclass of the interface or abstract class it references. In the Strategy Pattern, we do not have a single state. Different behaviours are defined using different strategy objects. State would be considered an extension of Strategy, where both are based on composition but Strategy makes the helper objects independent. State does not restrict dependencies.

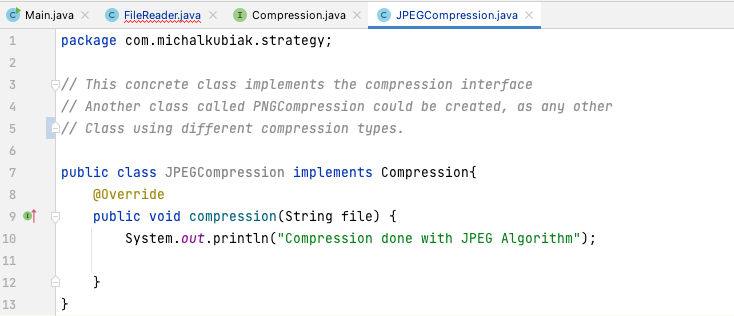
Strategy is used to isolate the business logic of a class from the implementation details. It is used when a class has a lot of conditions or operations

Implementation:

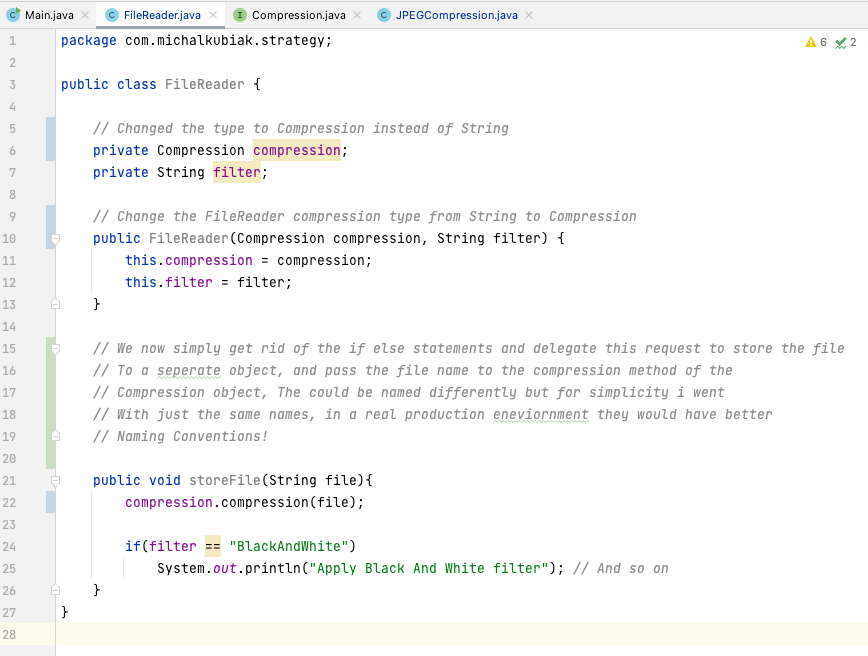
Compression Interface:



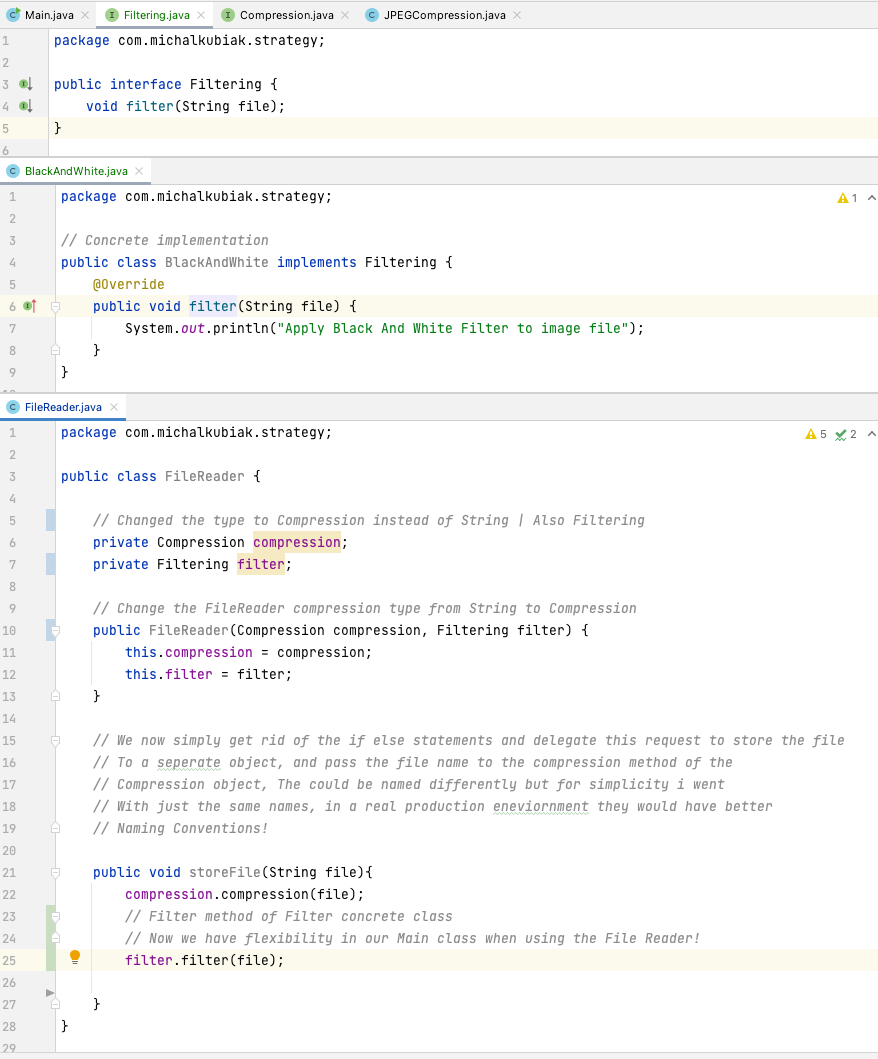
Concrete Compression Objects:



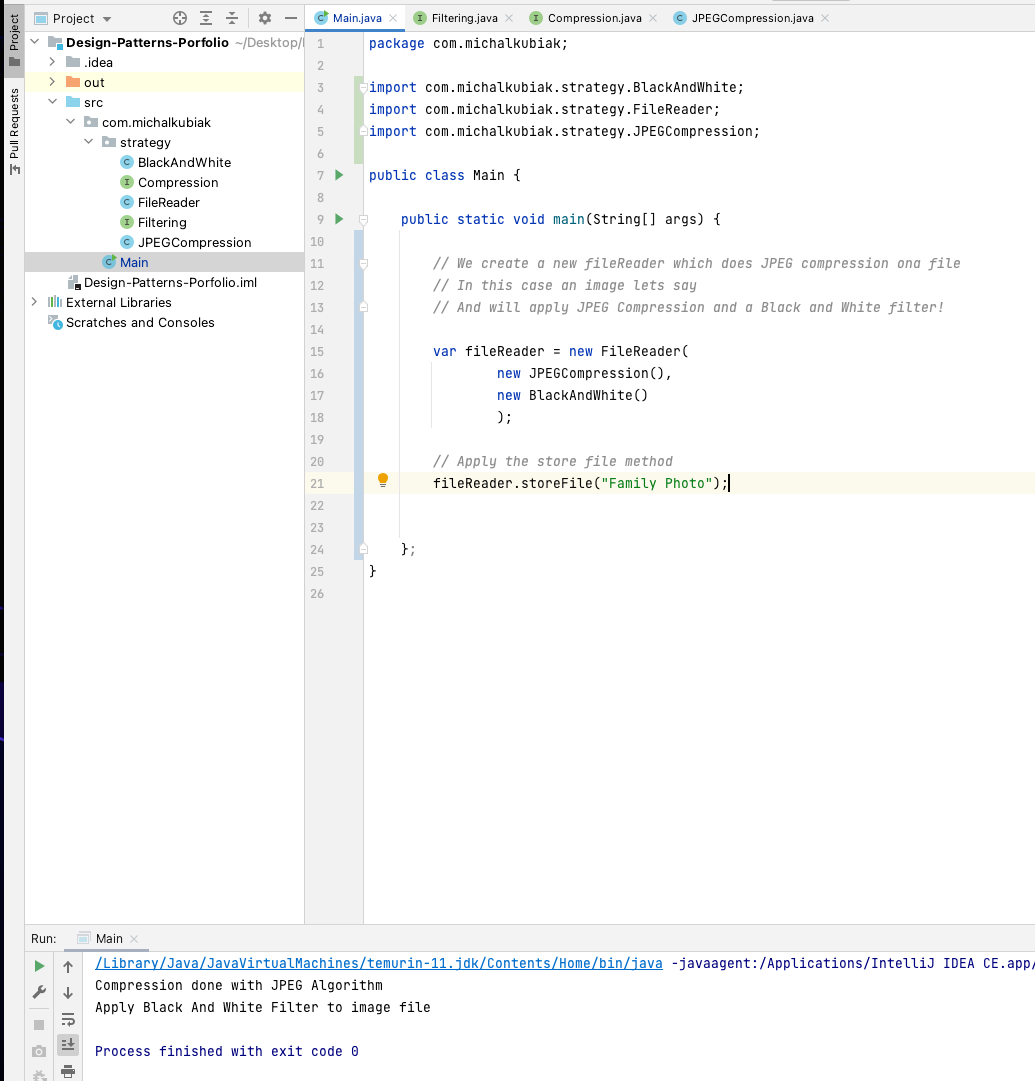
File Reader Type Changes for Compression Type:



Filtering Interface and subsequent filter class implementation and File Reader type changes:



Main Class Showcase and Use Cases:



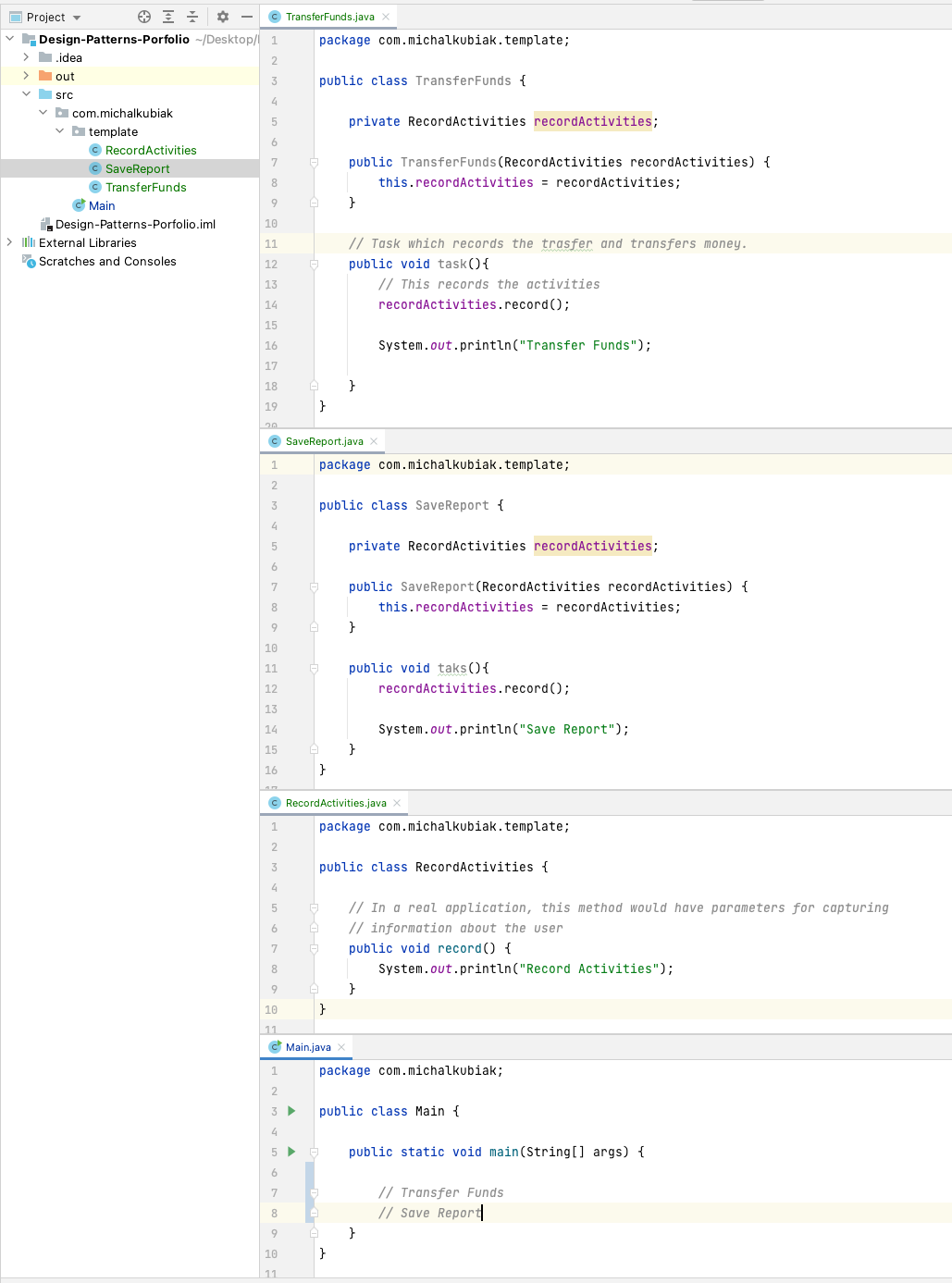
When the app is compiled, we can see that the string ‘’Family Photo” has had a JPEG compression applied and was filtered Black And White.

The application can now be extended by creating more filters or compression methods classes that implement their interfaces, making the app more maintainable and more flexible in the future!

**Template Method** is a behavioural design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Let’s say we are making a Banking Application, and we are to record the user’s activities in a trail and record who did what and when. Because we have different *types* of tasks, we do not want to implement these methods of a single class because every time we want to change the Task, we would have to come back and modify this Banking App Class.

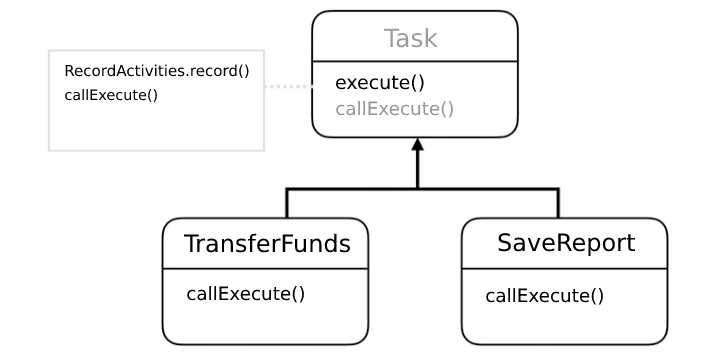
We want our system to be **Closed for Modification & Open for Extensions**. Each Task should be a separate class. We created a class called RecordActivities which would, in a real application, record a user’s Task of transferring funds. Now we add a second and third class which saves a report and transfers funds. There is a couple of problems with the implementation of these classes in the figure below.



The first problem is the **duplication of code.** Every new Task needs to record the activities after adding the RecordActivities class’ method record and initializing it with a constructor. It is too repetitive. There is also no structure to follow. A new task does *not* have to reference the record() function. There is no safety net for a new task to have this method in any new task.

There are two ways to solve this problem, using polymorphism and the Strategy Pattern or using inheritance. The Template Pattern uses inheritance to solve this problem. TransferFunds and SaveReport would typically call the recordActivities.record() method, code duplication. We can have them extend a common class called Task, where we move that common behaviour. The concrete classes will **extend** the Task abstract class, call the callExecute() abstract method, and override it. When the class is extended, the behaviour will be determined. We introduce a callExecute abstract method and call it in the execute method. Task, being an abstract class, can only be extended.

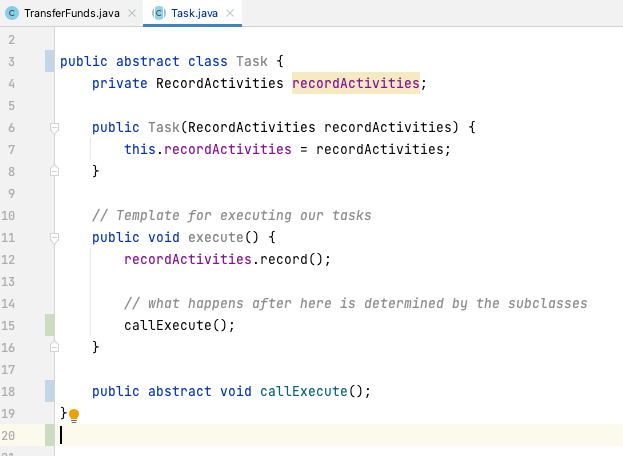
**The pros** of this pattern are that it lets clients override various sides or parts of a more extensive codebase or algorithm, so they are less affected by changes to other features. Duplicated code is decreased by putting it into the superclass. **The cons** are that the client will be limited to only the provided template. The **Liskov Substitution Principle** could be violated by not having a default implementation step in the subclass. Generally, the Template method is more brutal to maintain the bigger the algorithms or steps along the way get.



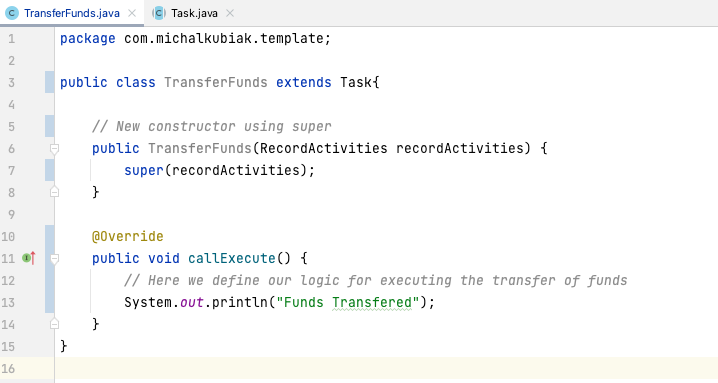
Because our execute method defines a template or skeleton for an operation, the Gang of Four book, the Template method is demonstrated by extending a concrete class to an abstract class with a ‘templateMethod’ and some primitive abstract operations. We can have several abstract methods that are called by the templateMethod. A concrete class will extend the abstract class overriding or implementing those primitive operations, also known as hooks. A note here is that the primitive methods (or, in this example, the callExecute() method) do not have to be abstract, it is up to the subclass to decide whether or not it needs overriding.

Implementation:

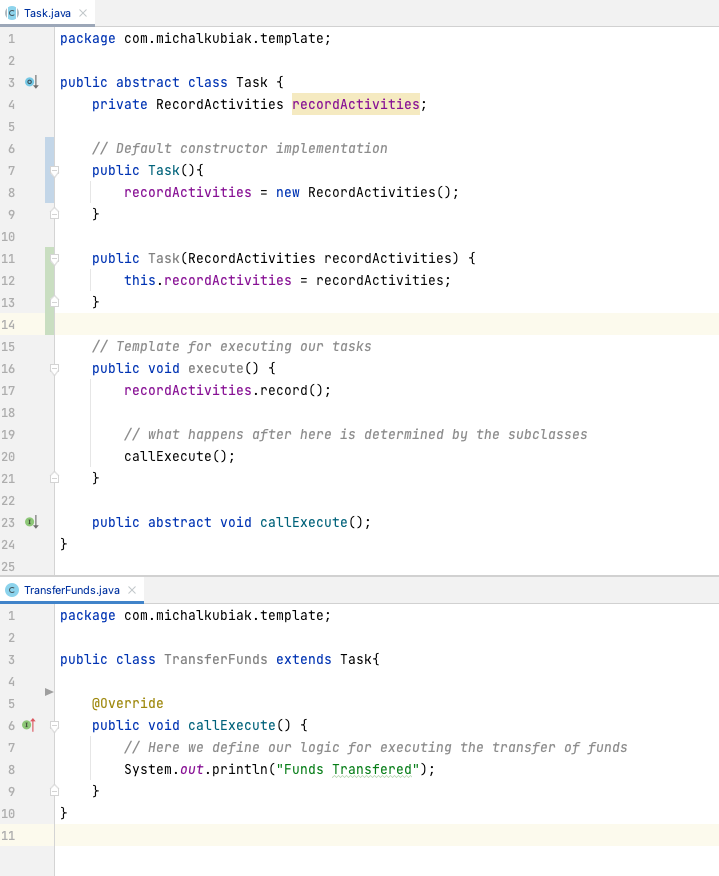
Creating the Task class, making it abstract, and adding the execute method, which calls the callExecute abstract method and the record() method from the RecordActivities class.



When creating the TransferFunds class, we need to convert the constructor to take the recordActivities from its extended class using ‘super’. This is how we implement a concrete Task.



Further duplicate code decrease, super constructor transfer into the Task by adding a default constructor. The TransferFunds now only needs the callExecute() method and whatever implementation or algorithm inside it that it needs.



In the main class, a new Task object has access to both execute and callExecute methods. We can change this by making the abstract methods protected. Otherwise, the callExecute method, if called, will not record the activities in the execute() method, which goes against the purpose of the template method. The application now runs the TransferFunds task and records the activities and transfers funds!

