Design patterns CA

Mark O Donoghue (T00196848)

Tralee Institute of Technology

Design patterns

Contents

[Mark O Donoghue(T00196848) 3](#_Toc29158083)

[Banking Application 3](#_Toc29158084)

[1. Introduction 3](#_Toc29158085)

[2.1 Strategy pattern Introduction 3](#_Toc29158086)

[2.1.1 UML Class Diagram before refactoring to pattern 4](#_Toc29158087)

[2.2 Calculating Interest on a Loan before refactoring to pattern 5](#_Toc29158088)

[2.3 The Problems 6](#_Toc29158089)

[2.3.1 DRY Principle 6](#_Toc29158090)

[2.3.2 OCP Principle 7](#_Toc29158091)

[2.4 Refactoring to the Strategy Pattern 8](#_Toc29158092)

[2.4.1 UML Class Diagram refactored to strategy pattern 10](#_Toc29158093)

[2.4.2 Implementation 11](#_Toc29158094)

[2.5 Runtime 15](#_Toc29158095)

[2.6 Unit Testing 16](#_Toc29158096)

[2.7 Conclusion on strategy pattern 18](#_Toc29158097)

[3.1 Observer pattern Introduction 19](#_Toc29158098)

[3.2 Observer UML Class Diagram 20](#_Toc29158099)

[3.3 Observer Implementation 22](#_Toc29158100)

[3.4 Observer Conclusion 35](#_Toc29158101)

[4.1 Decorator pattern Introduction 36](#_Toc29158102)

[4.2 Decorator Use Case 36](#_Toc29158103)

[4.3 Decorator UML Class Diagram 37](#_Toc29158104)

[4.4 Decorator Implementation 38](#_Toc29158105)

[4.5 Decorator Conclusion 45](#_Toc29158106)

[5.1 Singleton Introduction 47](#_Toc29158107)

[5.2 Singleton Use case 47](#_Toc29158108)

[5.3 Singleton UML Class Diagram 48](#_Toc29158109)

[5.4 Singleton Implementation 49](#_Toc29158110)

[5.5 Singleton Conclusion 52](#_Toc29158111)

[6.1 State pattern Introduction 53](#_Toc29158112)

[State Pattern Use Case 55](#_Toc29158113)

[6.1.1 New Interest Account State 56](#_Toc29158114)

[6.1.2 Low Interest Account State 58](#_Toc29158115)

[6.1.3 High Interest Account State 60](#_Toc29158116)

[6.1.4 Overdrawn Account State 62](#_Toc29158117)

[6.2 State Pattern Context Class 64](#_Toc29158118)

[6.3 Demoing the State pattern 66](#_Toc29158119)

[6.4 Benefits state pattern brought to the application 70](#_Toc29158120)

[6.5 State Conclusion 70](#_Toc29158121)

[7.1 Factory Method Pattern Introduction 71](#_Toc29158122)

[7.2 UML Class diagram Factory method pattern 72](#_Toc29158123)

[7.3 Implementation of the factory method pattern 73](#_Toc29158124)

[7.4 Conclusion Factory Pattern Method 74](#_Toc29158125)

[8.1 Template Method Pattern Introduction 76](#_Toc29158126)

[8.2 Template Method Pattern UML Class Diagram 77](#_Toc29158127)

[8.3 Template Method Pattern Implementation 79](#_Toc29158128)

[8.4 Template Method Pattern Runtime 82](#_Toc29158129)

[8.5 Benefits of the template pattern method to Banking App 82](#_Toc29158130)

[8.6 Conclusion Template Method Pattern 83](#_Toc29158131)

[9 GitLogs 84](#_Toc29158132)

# Mark O Donoghue(T00196848)

# Banking Application

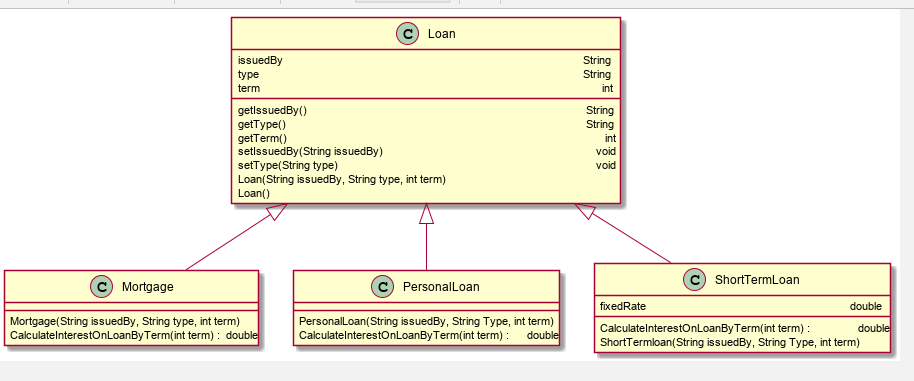
# Introduction

For the purposes of demonstrating design patterns, I decided to create a new banking application that would demonstrate each pattern in a real-world scenario. It is vital to create an application with some real-world value, in order to properly discuss design patterns. The reason it had to be a new application is because previously created applications in college did not fit the criteria. The banking application is a java Restful web service that incorporates spring and ORM technology’s similar to what one might find in a real-world bank. The application that I created incorporates 7 design patterns.

# Strategy pattern Introduction

The intent of the strategy pattern is to define a family of algorithms and encapsulate each one, which allows them to be interchangeable. Strategy allows the algorithm to vary independently from the clients that use it. This encapsulation allows for a developer to add features/functionality etc without altering classes which would violate the OCP principle of which will be discussed later. The advantages of the strategy pattern to the developer are numerous. Since behaviours are encapsulated, new behaviours don’t change the pre-existing behaviour of the application. Unit tests aren’t broken by new behaviours. The Strategy pattern makes code more maintainable by housing all the code that frequently changes in one area. We shall look at this in greater depth with real life examples when discussing the application.

# UML Class Diagram before refactoring to pattern



## Calculating Interest on a Loan before refactoring to pattern

Currently the bank offers 3 distinct types of loans. These beingz short-term loan, personal loan or mortgage. The base requirements for the loan functionality were to calculate interest on a loan, based on length of loan. However, the implementation of this calculation would change based on specific criteria. These criteria being that long-term loans such as Mortgage and Personal Loans would be subject to lower interest rates and short-term loans would be subject to higher interest rates. The Long-term loans were to be calculated as such, the interest rate would be 9% under 15 yrs and anything longer would be 15%. However, the short-term loan behaves differently based off the bank’s requirements. It still calculates loan by term but if the term is less than equal to 5yrs the interest rate is 15% and any term longer is 25%.

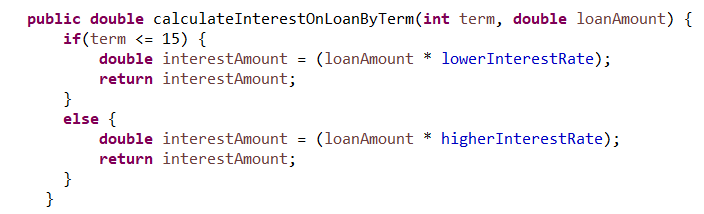


Figure 1 Long Term Loan

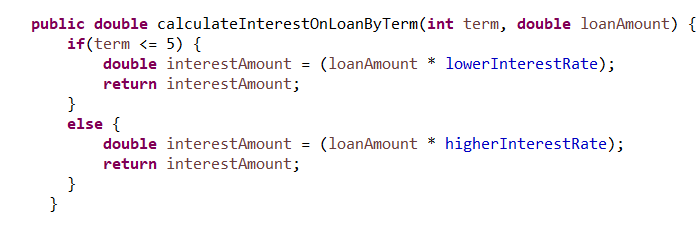


Figure 2 Short Term Loan

Upon reviewing, designing the UML diagram and getting another requirement from the bank, it became apparent that there were some pretty serious flaws in how the application was designed.

## The Problems

### 2.3.1 DRY Principle

The DRY principle (Don’t Repeat Yourself) is a basic principle of software development that aims to reduce the duplication of information in a system. As can be seen from the UML Class diagram in Figure 1 the CalculateInterestOnLoanByTerm() method was being repeated in 3 of the classes. Upon closer inspection, I realised two of the classes contained the exact same implementation (Personal Loan & Mortgage).

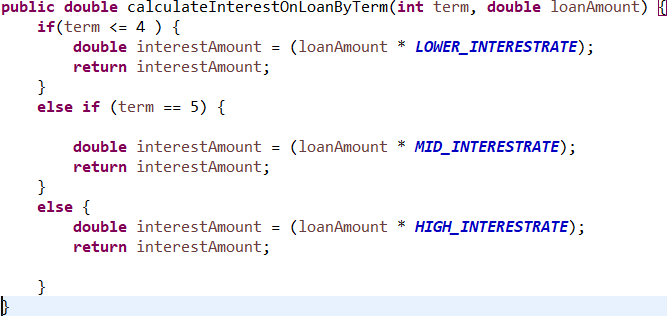
The reason that violating the DRY principle is of such serious concern is if that I was given a new requirement to calculate interest based on different requirements, I would have to change multiple implementations of the method as opposed to just the 1 implementation. This would make the process of altering the code more time consuming as well opening your application to the possibility that you may not alter all implementations meaning that your application may behave incorrectly, producing results that you weren’t expecting.

### 2.3.2 OCP Principle

I then received a new requirement from the bank for calculating short term loans in a different way. The interest amount would now be calculated off the following metrics

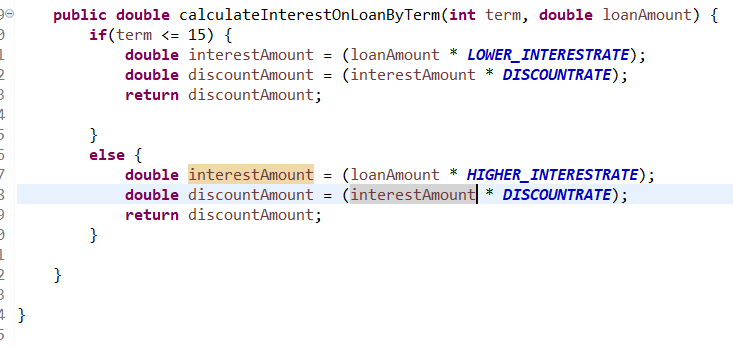
* Term 4yrs or shorter 15% interest
* Terms exactly 5yrs 20% interest
* Term 6yrs or greater 25% interest

To accomplish this change in requirements I updated the CalculateInterestOnLoanByTerm() method in the ShortTermLoan class to reflect this change in requirements.

Figure 3 show the updated ShortTermLoan

By doing this I was breaking another OOP principle called The OCP (open-closed principle). The principle states that a class can be open for extension but closed for modification. What is meant by this is that developers should write code that doesn’t require modification of existing code every time the requirements of the customer/business change. The reasoning behind this principal is that it makes the code more maintainable and upon modifying this method, I was shown exactly what was meant by this. The Unit tests that I had previously written were now beginning to fail and I would have to retest this method in order to ensure that the method was behaving as intended. The cost of me changing the method was time and resources but imagine if that method was used more frequently throughout the application or I modified multiply classes. The cost of the modification would be a lot more apparent.

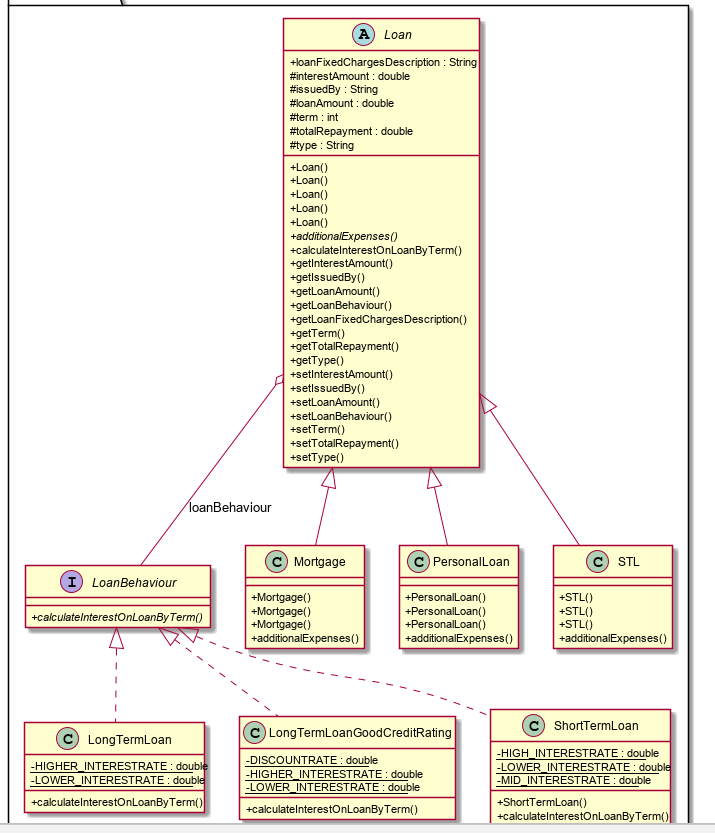
It was at this stage of the development process that I realised it was time for a change to prevent these problems from occurring in the future. Especially since I realised that the bank would continue to alter and add different ways to calculate interest on loans. They wanted to allow customers with good credit ratings who wanted long-term loans to get the loan at a lower rate of interest as the risk on non-repayment was significantly lower. This makes good business sense as it has the potential to attract new customers without a significant increase in risk. After a customer applied for a loan with a good credit rating, they would receive a 10% reduction to the overall cost of the loan.

Figure 4 Long Term Loan good credit rating

## Refactoring to the Strategy Pattern

In order to implement the strategy pattern, I had to identify the behaviour that was changing and would most likely change in the future. It became apparent very quickly that the behaviour was how the interest was being calculated for loans. I identified 3 separate implementations of the behaviour. This is how ShortTermLoans(ShortTermLoan) were being calculated, how LongTermLoans(Mortgage & Personal) were being calculated and how LongTermLoans for customers with excellent credit ratings were being calculated. I needed these behaviours to be abstracted from the main application so that I could use them as needed and avoid further violations of core OOP principles.

### UML Class Diagram refactored to strategy pattern



### 2.4.2 Implementation

I set about accomplishing this task by making my Loan class abstract. Encapsulating its attributes by setting their access modifiers to private and using public getters/setters in order to alter to data. To achieve abstraction, I needed to create an interface which I called LoanBehaviour, to house the method that was causing me so many problems earlier (CalculateInterestOnLoanByTerm(**int** term, double loanAmount). I could have achieved abstraction by using an abstract class, but I didn’t have any concrete methods to add, as the method that LoanBehaviour was housing was abstract and each concrete class would implement it differently. This would have been a waste of an extension slot as each class can only extend 1 class but implement many interfaces.

I created three new classes PersonalLoan, ShortTermLoan , LongTermLoan and that would contain the three separate implementations of this method(sidenote I had to refactor a previously created class to STL to accommodate this, so I can’t stress the importance of a strong naming conventions when it comes to classes, variables, methods etc). Below, is the new LoanBehaviour Interface as well as the three new classes I created with the implementation of LoanBehaviour.

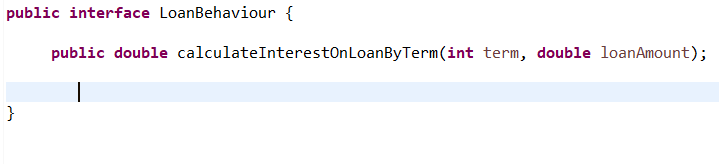


Figure 5 interface LoanBehaviour

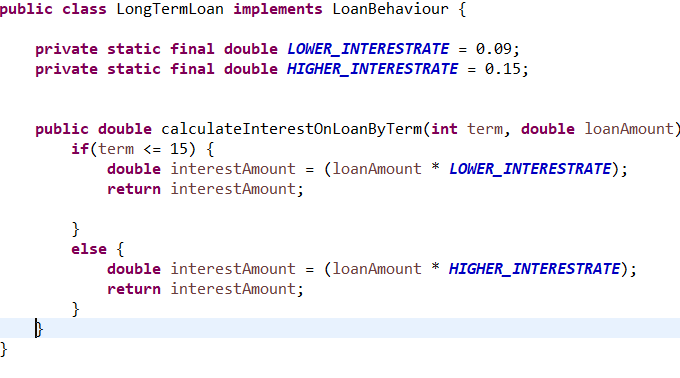


Figure 6 LongTermLoan

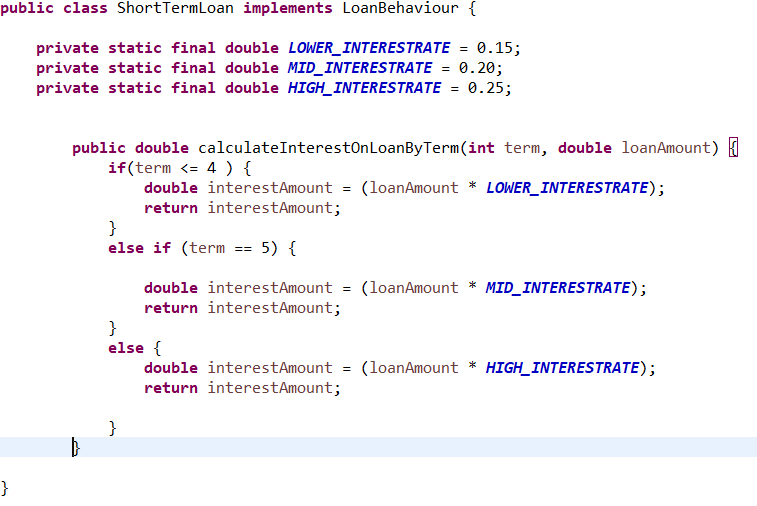
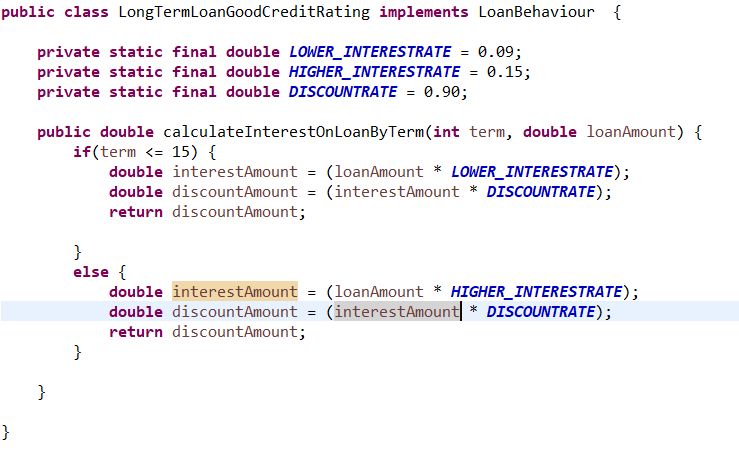


Figure 7 ShortTermLoan

Figure 8 LongTermLoanGoodCreditRating

Following this, I set about removing these method implementations from the Mortgage,SLT and Personal Loan classes. I still however,needed to be able to access them on instantiation of a new loan. I accomplished this by providing a reference of the Loan Behaviour interface and accompanying getter/setter methods in the abstract Loan class.

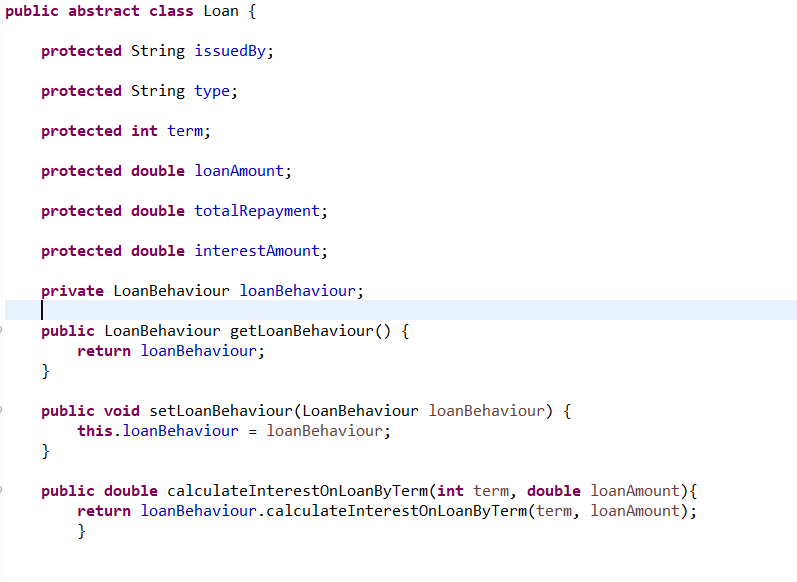


Figure 9 abstract Loan class

At this point, the Loan class knows nothing about the implementation of its own behaviours. This would allow me to create as many additional behaviours as the bank requires without worrying about affecting existing code. As well as not having to worry about breaking/altering existing code by adding additional behaviours, I can also be confident that my unit tests won’t have to be altered in the future as any new change would get its own implementation of the LoanBehaviour interface. Instead of loan or its sub classes inheriting its loan behaviour, it can now be said that a loan is composed of a suitable behaviour. This is with keeping with the design principle of favouring composition over inheritance. The idea behind this principle is that it creates more reusable and flexible code. Since the decision about which algorithm to use is deferred till runtime, it allows the calling code to be more flexible. To give an example from the code, I implemented a customer’s credit score might not be known until runtime and the resulting interest amount that the customer has to pay will be drastically different.

## Runtime

In the below example, two separate loan interest amounts are being calculated a long-term loan and a short-term Loan. They both produce the expected results.



Figure 10 runtime

Both methods return a double value that is equivalent to how much interest is required. The console log shows that these are in fact the expected values. Let’s demonstrate the strategy patterns flexibility. We will say that based on given user input, we determined that Jennifer was a customer of good credit rating. This will allow us to alter the algorithms that are applied to her loan resulting in a different value being returned.

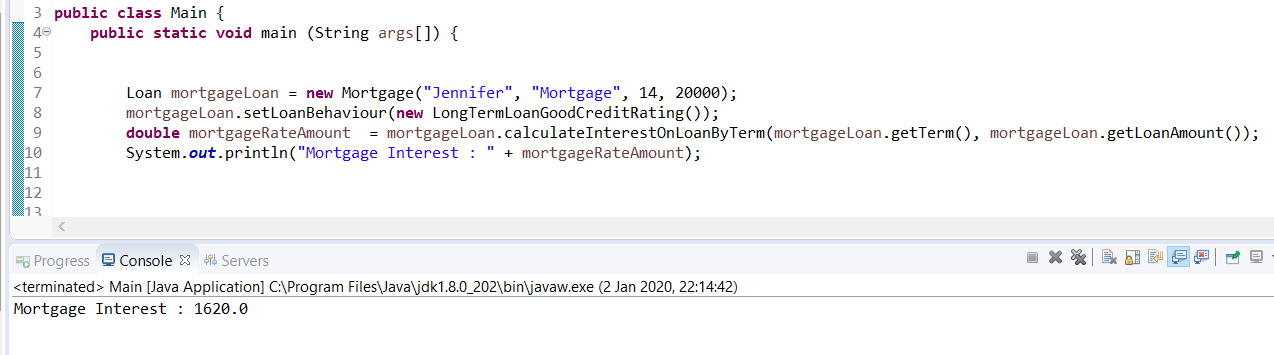


Figure 11 new algorithm acting on jennifers mortgage

As well as this flexibility, a new algorithm could be created for applying interest on loans and could be implemented rather easily without having to violate the OCP principle or having to worry about effecting multiple areas of your application, as the new algorithm would be contained within its own concrete class that implements LoanBehaviour.

## Unit Testing

I used Mockito in order to test my loan behaviours. I mocked out my loan behaviour interface using the @Mock annotation and Tested each behaviour. In each test, i instantiated the mock object with the concrete behaviour that I was looking to test. Below show the unit tests that were carried out on the application.



Figure 12 Strategy Tests part 1

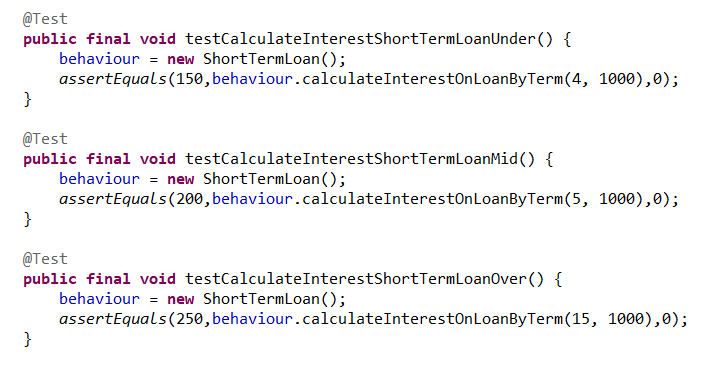


Figure 13 Strategy Test part 2

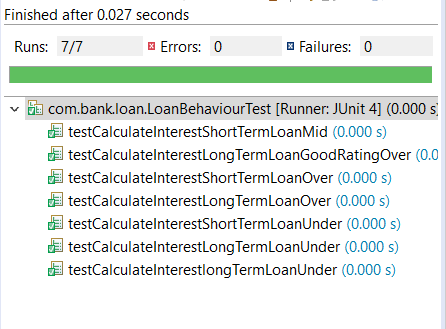


Figure 14 Strategy Tests passing

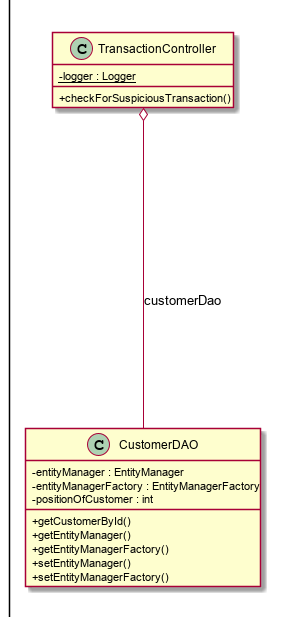
## Conclusion on strategy pattern

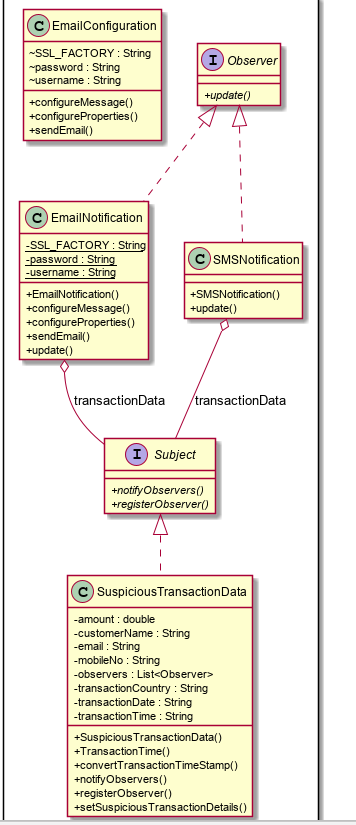
The strategy pattern definitely made the application more maintainable. Behaviours are confined to one place, so if a behaviour isn’t working as expected, a developer would know where to look. It also allowed the application to comply with OOP principles that the application simply wasn’t adhering to before. It made the code more flexible as algorithms weren’t applied to it until runtime, so different algorithms could be applied following different user input. In my view, there is great benefits to be had when implementing the strategy pattern. However, I do imagine there is a big cost to implementing this on a large-scale application that could contain 10s of thousands of lines of code. You would need developers skilled enough to identify all the varying behaviours that the application contains. They could identify quite a large amount of these behaviours and would then have to implement them all as well as remove previous references .This would take a lot of time as well as developers to implement on that scale and a company may not be willing to invest that amount of resources on such an endeavour, especially, if there is nothing functionally wrong with their application.

# Observer pattern Introduction

The intent of the observer pattern is as follows. The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. The observer has a model very similar to a publish/subscribe model were many subscribers, subscribe to a publisher. However, observer typically uses subject and observers. The relationship that exists between them is a one to many relationships. This means that one subject could have many observers. The observers have a dependency on the subject in that when the observers state changes the observers are notified.

## Observer UML Class Diagram





Sidenote had to remove some getters and setters to fit uml.

## Observer Implementation

A requirement was received by the bank that they wanted to notify customers when a possible fraudulent transaction has occurred. This is a common occurrence in the modern age of internet banking and the bank needed a way to inform the customer of the transaction that had occurred and why it was deemed suspicious. Obviously, it is imperative that a customer receives this notification as soon as is possible and in modern times there are many ways and devices that a customer can be reached on. It became glaringly obvious that the observer pattern may be the best way of fulfilling this requirement. The reasoning was that when a suspicious transaction has occurred the customer could be reached on a multitude of devices. These observers could be added and removed as needed which fits perfectly with what the bank wanted to achieve. As technology evolves, so does the technology that we use to communicate with each other. From post to phone to text messages to email. The logic is that as a communication mechanism becomes outdated, it could be removed as a subscriber and as other communication mechanisms become more prominent, they could be added as subscribers.

To implement the observer, I first created a Subject interface this interface would contains abstract method that would be need by any concrete subject class.

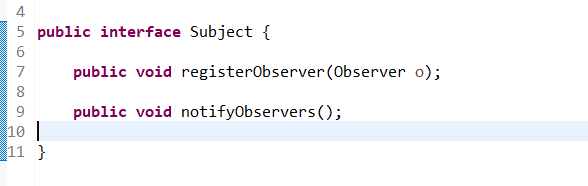


Figure 15 Subject Interface

The subject interface that I created defines the way in which observers can be registered by the subject and a notify method that will notify the observers when the concrete subject has changed state.

Following this, I created the observer interface. This interface contains only one method and that’s the update method. A class that implements the observer interface declares that it will listen to state changes made by the subject.

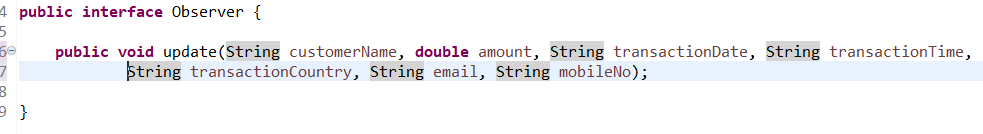
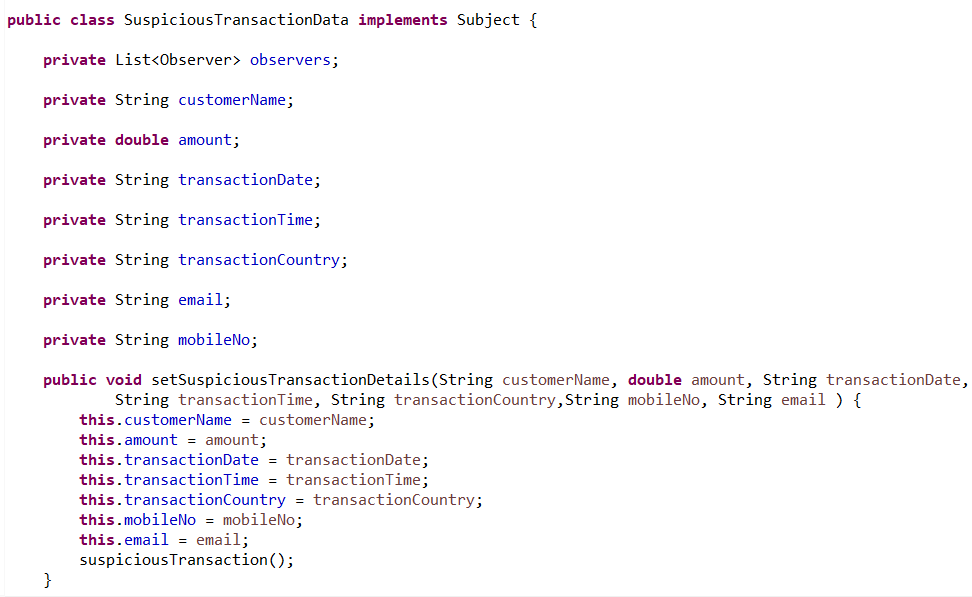


Figure 16 observer interface

Since all observers will have to implement the observer interface, it’s safe to say that each observer will have its own implementation of the observer interface. The details parameters that are being passed into the update method are the same parameters that will be needed to notify the customer of the suspicious transactions.

After creating the observer interface, I created the suspicious transaction concrete subject which implements the subject interface was created.

Figure 17 SuspiciousTransactionData Class

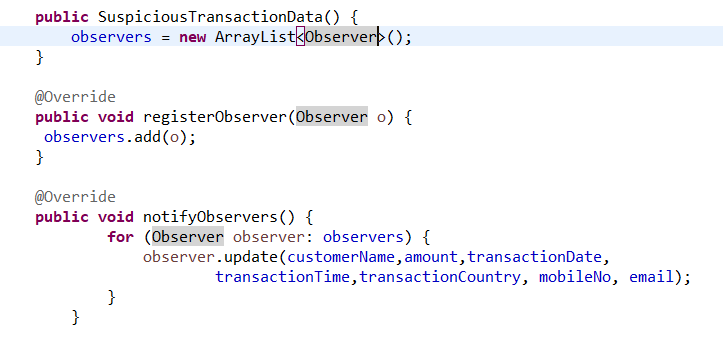


Figure 18 shows more of the SuspiciousTransactionData Class

The class also contains standard getters and setters for each of the attributes that are contained in the SuspiciousTransactionData Class. To give a brief overview of what is occurring in this class, above I have created a List attribute of type Observer to store all the observers that will be notified by this class. On instantiation of the class, the no arg constructor will set the observers list to type ArrayList. When the setSuspiciousTransaction method is called, it will populate the attributes of the SuspiciousTransactionData Class. Aa well as this, it will call the notifyObservers method that will notify all the observers of the SuspiciousTransactionData class state change. Also, I have provided the implantation of the registerObserver method that will be used to “subscribe” classes to the concrete subject class.

Currently two classes implement the observer interface. The first of these being the SMSNotification class. In order to be able to send a text message to a customer, I had to interact with the Nexmo API. This would allow the bank to send actual text messages to customers. To perform this, I had to add the Nexmo jar to my Project Object Model file or more commonly known as pom.xml. This was accomplished using Maven which is a project management and dependency tool. I added the following Maven dependency which gave me access to the libraries that I was going to need in order to send a message.

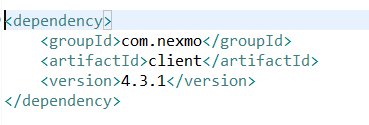


Figure 19 nexmo dependency

After adding this dependency, I created the SMSNotification class.

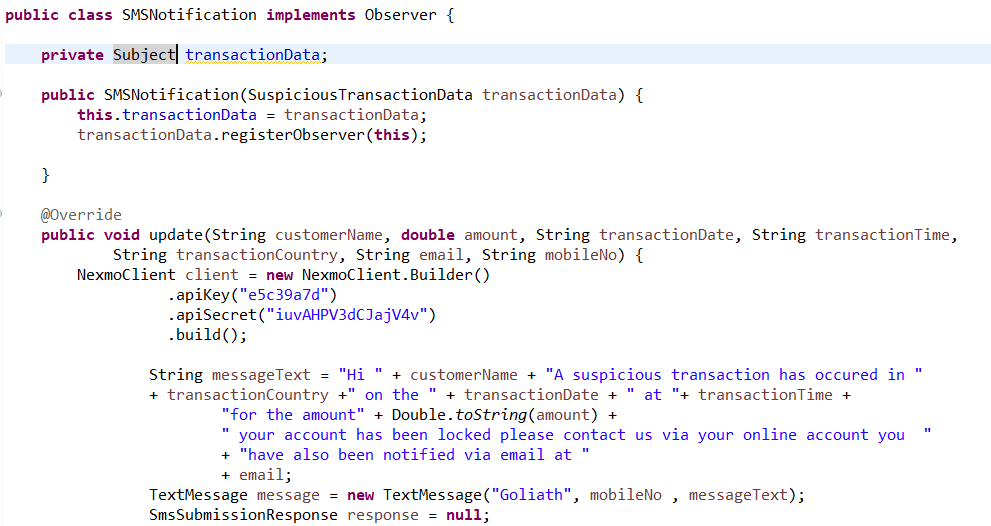


Figure 20 SMS Notification class part 1

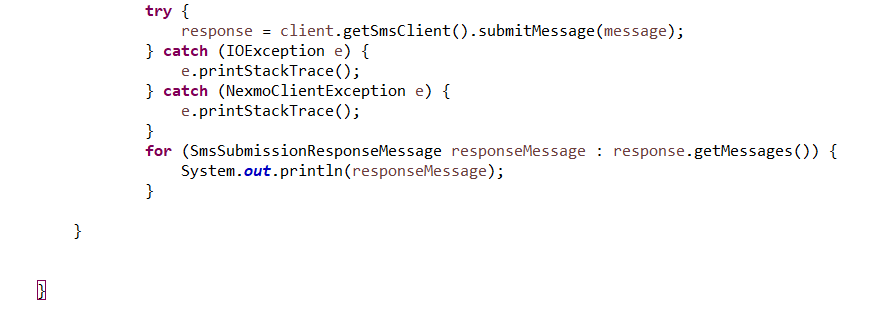


Figure 21 SMS Notification class part 2

To briefly explain what is happening in the above, the code is doing the SMSNotification class. The SMSNotification class constructor uses transaction data which is of type subject to register the observer. When the update method gets called, a Text message is built using the arguments of the update method. The Nexmo client is built and the message is passed in as a param and a text message with the relevant details is sent to the customer.

The other class that I created, that implemented the observer interface, was the email notification class. This class also need access to an external library, so I added the dependency to my pom.xml file.



Figure 22 Email Notification class part 1

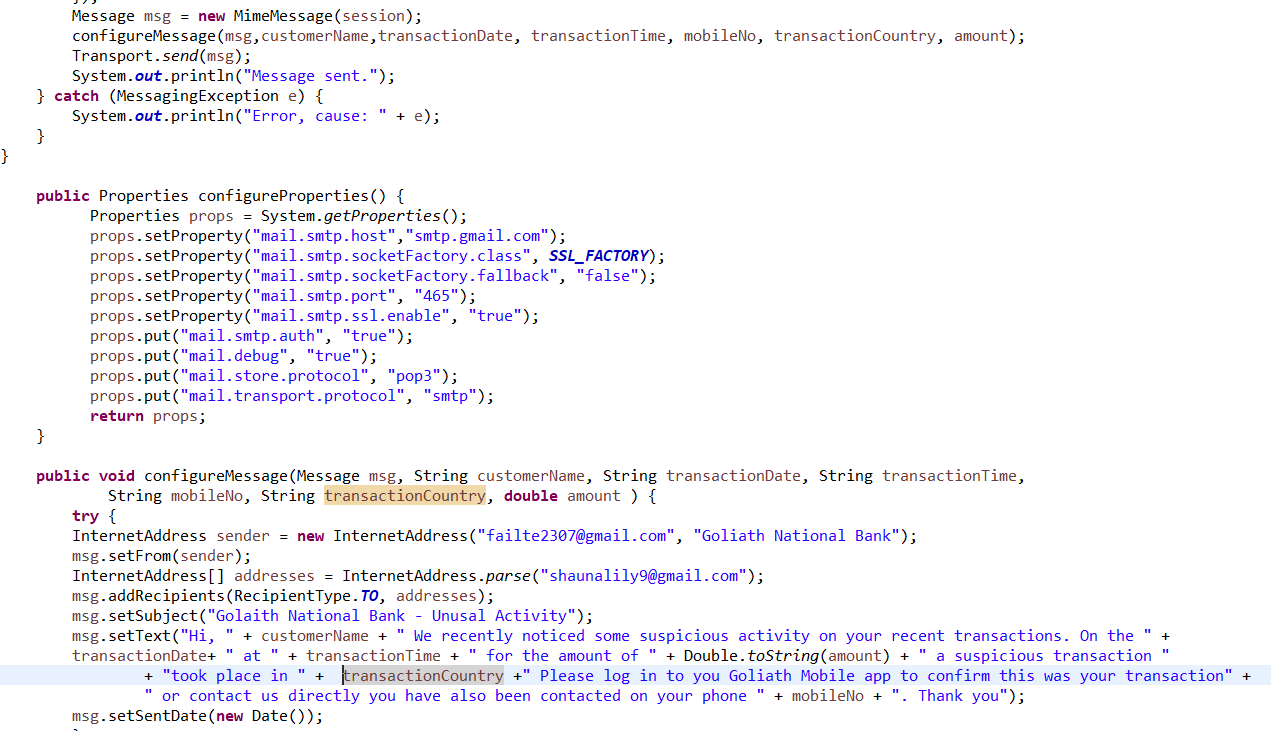


Figure 23 Email Notification class part 2

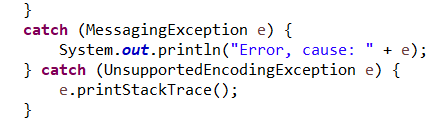


Figure 24 Email Notification class part 3

The above class again uses a constructor that takes a param of type subject called transactionData in order to register the class as an observer. The rest of the code deals with creating an email and includes helper methods that configure the email properties and configure the email message itself. Following this, I created a Customer entity class that would persist the data from my hosted mySQL database onto the Plain Old Object Class (POJO).

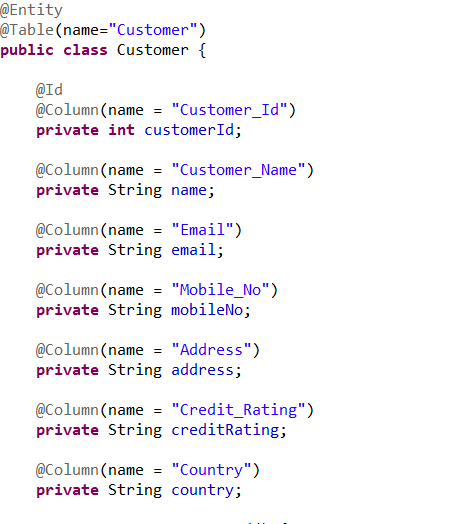


Figure 25 Customer Entity class

The annotations declare the class as an entity, reference the table and columns of the mySQL database. I then created a CustomerDTO object that would allow the customer data transverse the layers of my project, so that the data would be able to come through the controller and into the DAO layer. To show the DAO layer, we can see that it is configured using spring.



Figure 26 Customer DAO class

In the DAO class, I created a simple method that would find a customer from my database by id. I then pass the returned Customer Entity into a Customer DTO object using its constructor and return the DTO object. I use an EntityManager to manage the applications entities. In this way, I can delegate responsibility of managing database connections to the entity manager. I use Spring annotations such as @Repository to allow the application to know that this is a repository class. The @Repository annotation also saves the need of me having to create a CustomerDAO bean that would be used for injecting it into the TransactionController. Finally, the @PersistenceUnit annotation allows me to manage the configuration of my entity manager. It does this by mapping a persistence.xml to do the entity manager. This persistence.xml file allows me to specify what entities I want it to manage as well as what dialect and drivers and to use. The below screenshot shows the contents of the persistence.xml file.



Figure 27 persistence.xml contents

Finally, I created a TranactionController class the contents of which can be seen in the below screenshot.



Figure 28 Contents of Transaction Controller Class

As can be seen above, there is an annotation called @RestController to let the class know that it is a RESTful Controller. This allowed me to easily create an endpoint meaning I can receive a HTTP Request. I Autowired in a dependency of my CustomerDAO class using the @Autowired annotation. This will allow me to access to the getCustomerById method in my CustomerDAO class.

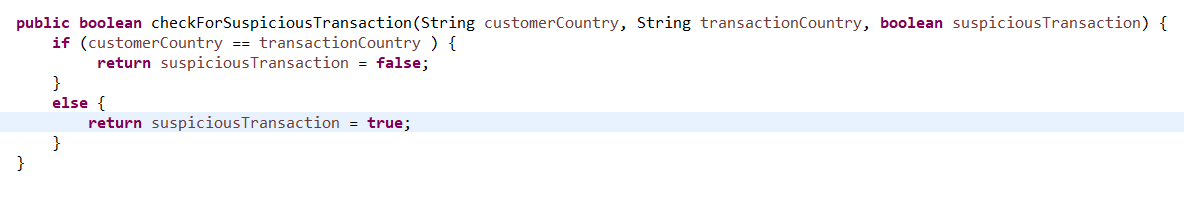
Finally, I created checkForSuspiciousTransaction method which takes in a transaction object at the endpoint /transactioncheck. I then create a CustomerDTO object and used the transactions customerId to populate it. This population occurs when finding the customer from the database that matches that customerId. I then use the check for suspicious transaction method that sets the field isSuspeciousTransaction Boolean to reflect whether the transaction is suspicious or not. It sets it to true or false, depending on whether the Customers country and the country the transaction happens in match or not

Figure 29 Suspicious transaction method

If they don’t match, I deem the transaction to be suspicious and instantiate my concrete subject class. I then instantiate my 2 concrete observer class passing in references to my concrete subject class. This registers them as observers. I then set the values of my concrete subject class formatting the date and time at the same time. This runs the update method on each concrete observer class notifying the customer by email and SMS. Finally, I return the customer object to the client as a JSON value.

Below shows screen of the endpoint being hit through postman and returning the customer object in JSON form.

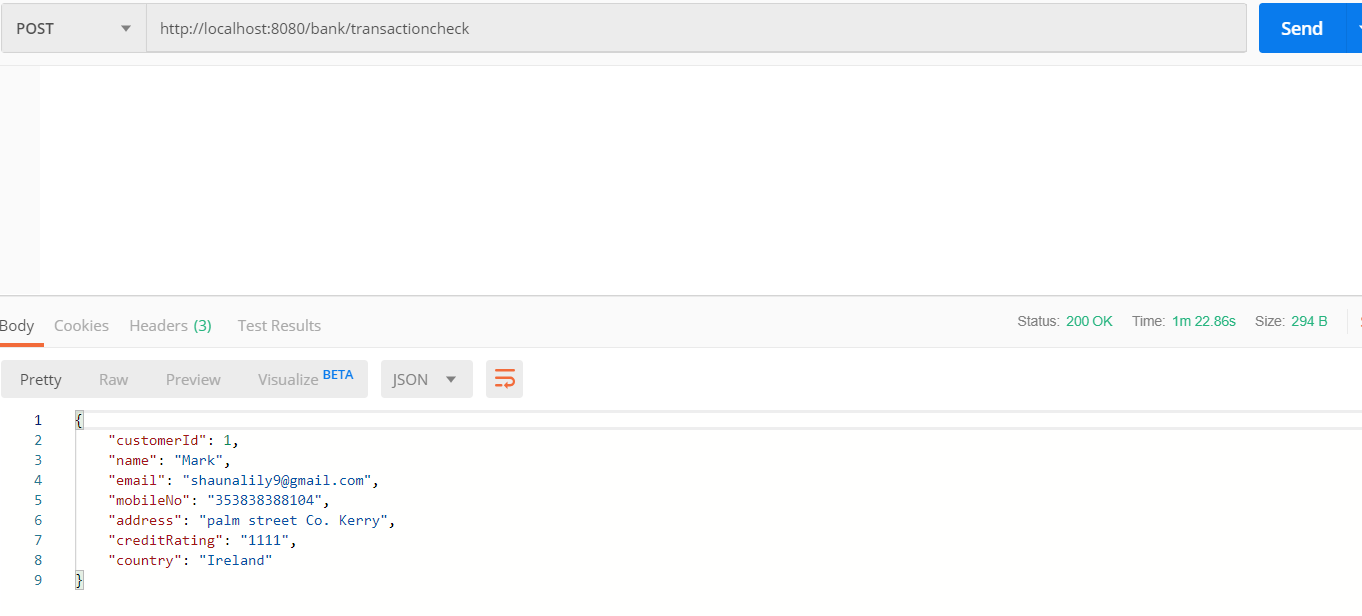


Figure 30 endpoint being hit and Customer object being returned

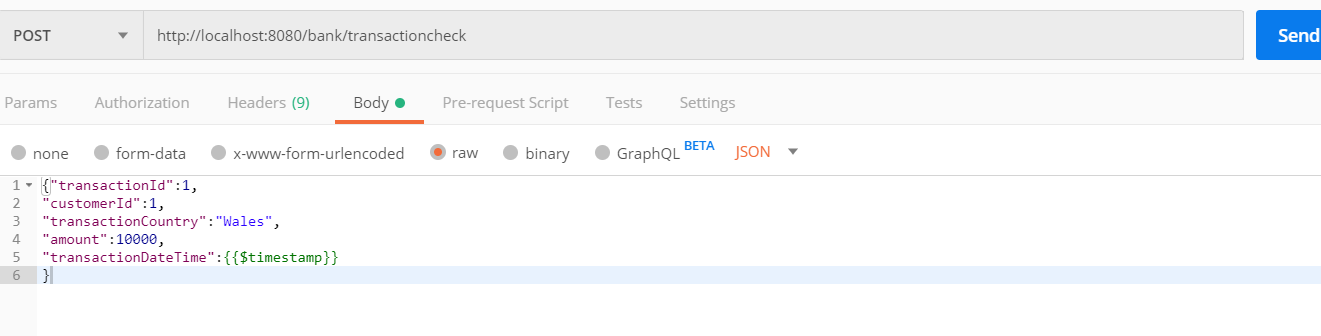


Figure 31 The JSON transaction object that was consumed by the client

The final two screens show the email and the being sent to the correct devices. A message being delivered via email and a text message being sent to the phone.

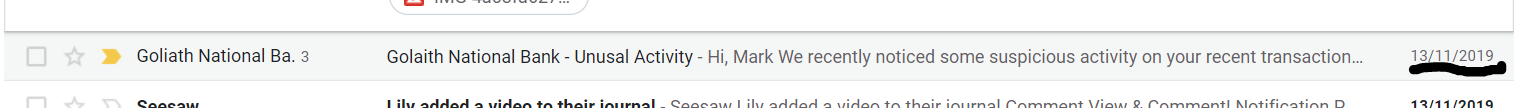


Figure 32 Email in correct inbox

Below are the contents of the email message

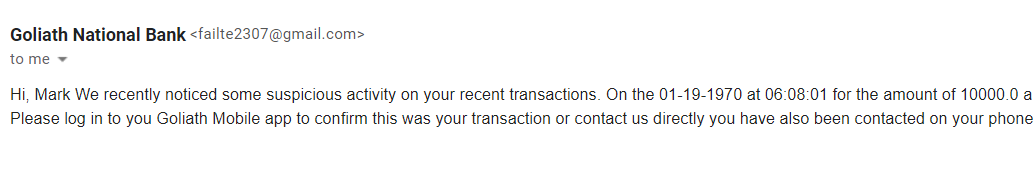


Figure 33 Contents of email message

Below are the contents of the SMS message that was received.

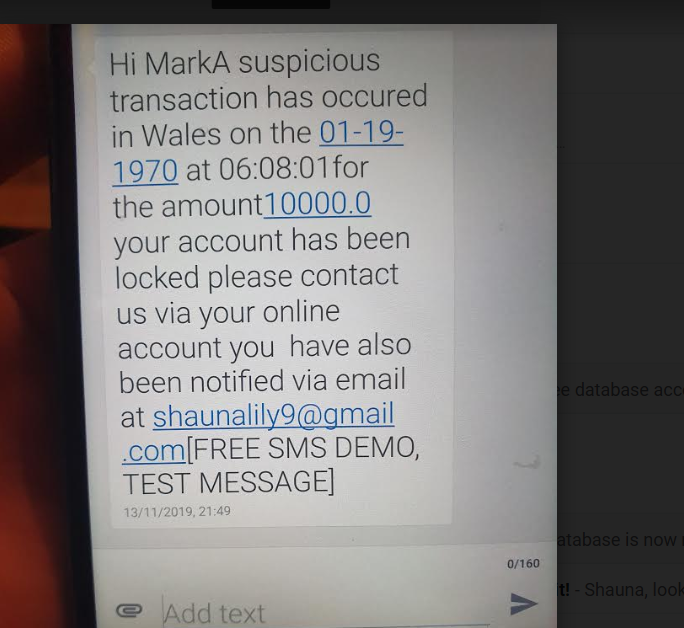


Figure 34 Contents of SMS message

## Observer Conclusion

The observer pattern follows the publish subscribe model, making it a great fit for a banking application. The Observer allows for decoupling, which means multiple concrete observer classes can be configured at run time. This pattern is a good fit for the use case I described as it is important that a customer is notified about a suspicious transaction. As well as this, the way in which they are informed can be easily manipulated to allow for either a preferred method of communication or to incorporate other communication mechanisms such as snapchat in the future.

# Decorator pattern Introduction

The decorator patterns intent is that The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. The decorator attempts to counter the typical overuse of inheritance and allows you to decorate classes at runtime using a form of object composition. In this way, objects are able to take on new responsibilities without having to make code changes to the underlying classes.

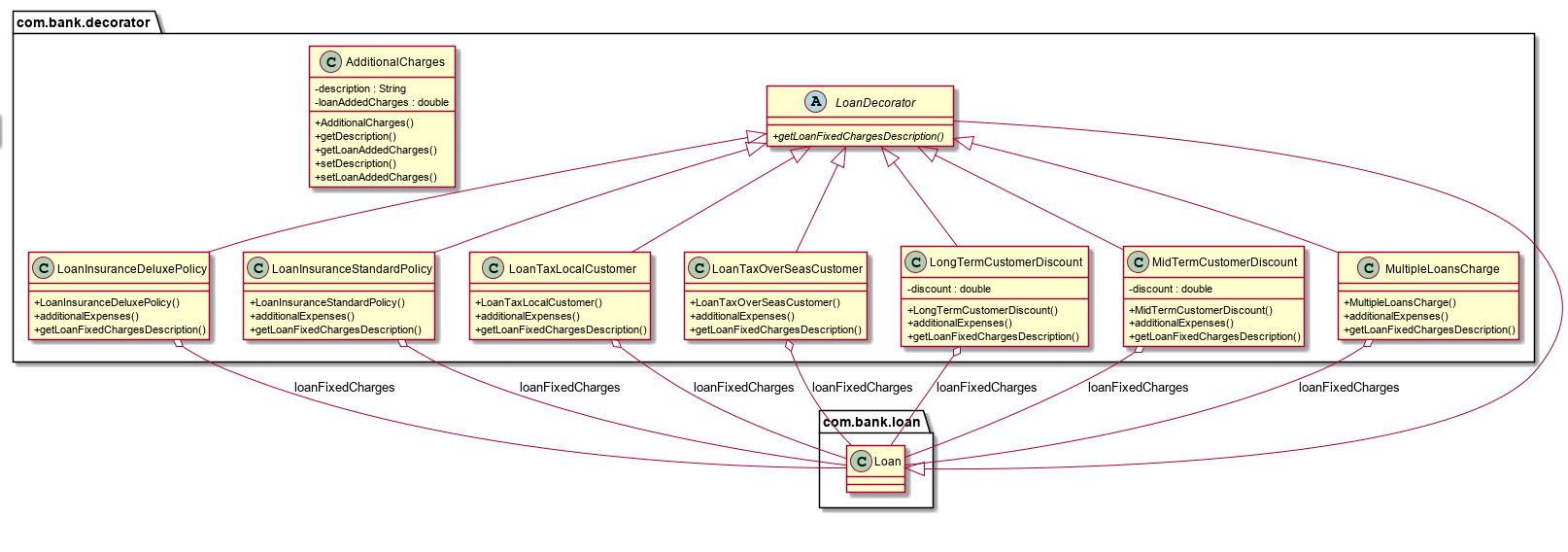
## Decorator Use Case

The bank wanted to provide additional functionality to the loan classes that were implemented in the strategy pattern. The exact requirements were that all loans would have a fixed charge associated with them. The loan types that were defined in the strategy are mortgage, personal loan and STL. Each of these classes would have a different fixed charge associated with them as per the bank’s requirements. This requirement by itself wouldn’t require any structural changes to the code because as we know from the strategy pattern each type of loan (mortgage etc) already extend the abstract class loan. So, all the would be required is adding an additional Expenses field in loan and leaving each subclass return the correct fixed charge. However, the bank wanted to add extra additional charges and quite a few of them ranging from Tax on overseas customers, multiple loan charges and customer discounts. This would create a massive problem with the structure of our code.

I’d argue that having each class extend from the abstract loan class would bring serious flaws into the system. It would cause class explosion as the loan class would be responsible for calculating all the different combinations of additional charges. Since the bank intended on having a lot of fixed charges, this problem would have become evident really quickly.

As well as this, whenever a fixed charge price changed, the code would also have to be modified. Since, realistically these changes could be frequent, this is a huge violation of the OCP principle. So, the only solution was to provide a flexible change to the structure that would be in compliance of the OCP principle. The change that I decided was introducing the decorator pattern.

## Decorator UML Class Diagram



Side not couldn’t fit Loan and Mortgage etc in UML Class Diagram but can be seen in strategy UML.

## Decorator Implementation

Decorators implement the same interface or abstract class as the component since the abstract Loan that I intended on calculating fixed charges on was already in existence from the strategy pattern I added the following.

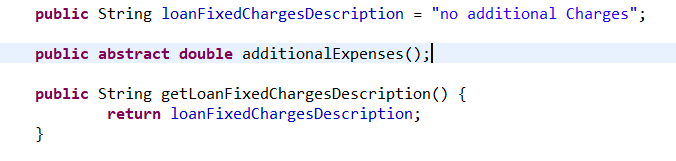


Figure 35 Added Abstract Loan Class

I then created the abstract LoanDecorator class that extends the abstract Loan class. All classes that extend these classes would be my decorators.

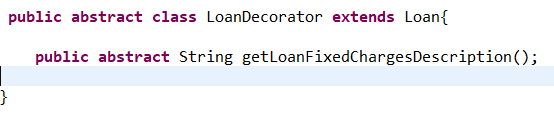


Figure 36 Contents of LoanDecorator

I then set about creating all my decorator classes. I added a constructor so that when these classes instantiate, they have a reference to Loan. Taking the first decorator class as the only example, so that I am not repeating myself. The additional expenses method is first delegated to the object we are decorating so that the cost can be calculated then the cost of LoanTaxLocalCustomer is added to it. The method getLoanFixedChargesDescription behaves in a similar manner, first delegating to the object being decorated and appending on its String. For the sake of time, I will provide screens of all the other decorator classes they all perform similar functionality bar the discount classes which take away from the overall cost.

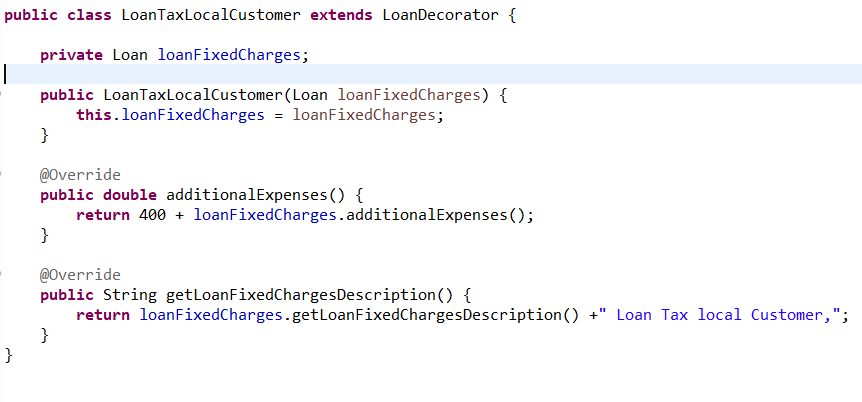


Figure 37 LoanTaxLocalCustomer

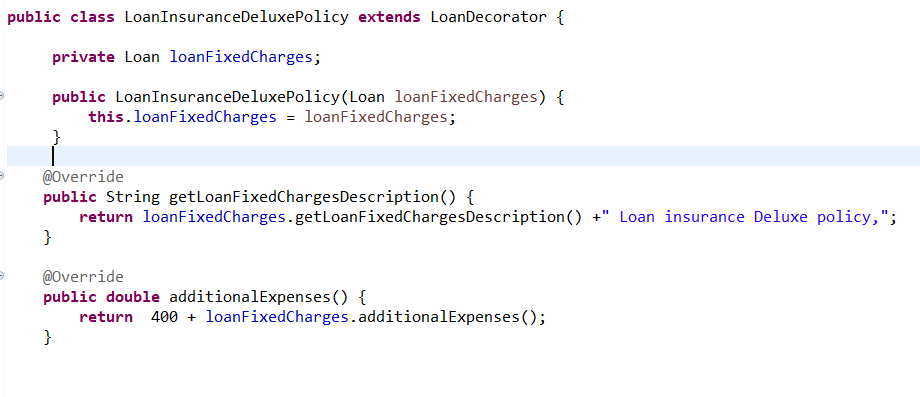


Figure 38 LoanInsuranceDeluxePolicy

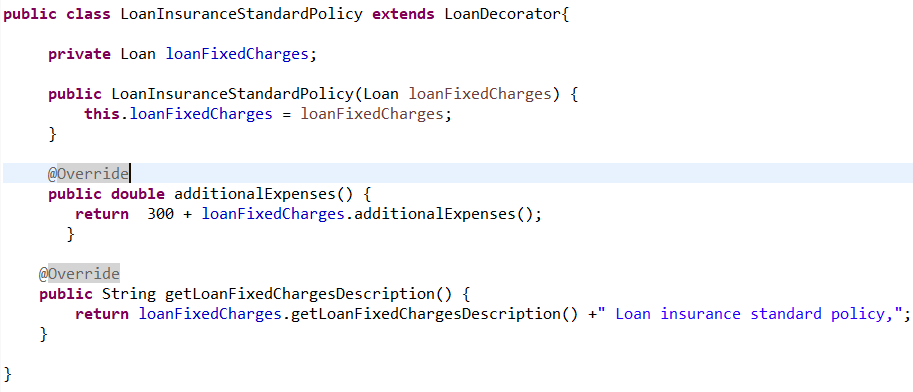


Figure 39 LoanInsuranceStandardPolicy

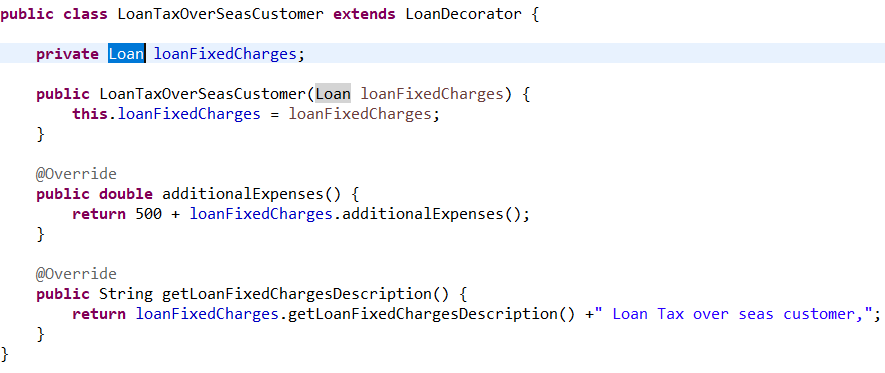


Figure 40 LoanTaxOverSeasCustomer

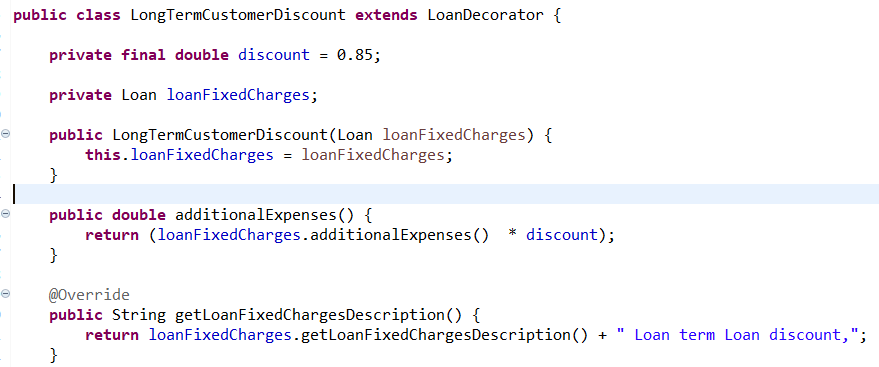


Figure 41 LongTermLoanDiscount

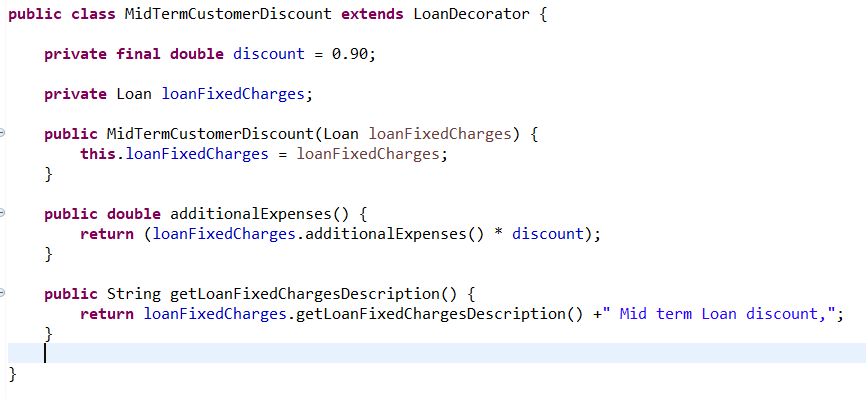


Figure 42 MidTermDiscountLoan

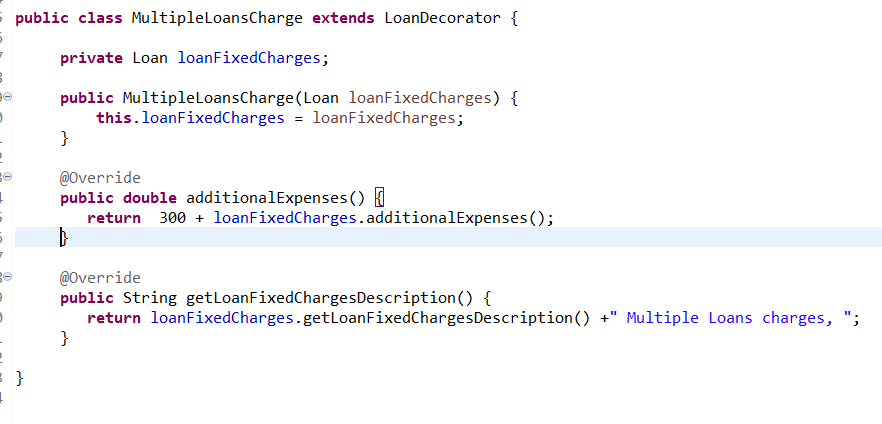


Figure 43 MultipleLoanCharges

Since the Mortgage STL and Personal Loan classes already existed from the strategy pattern, I added the following methods to them.

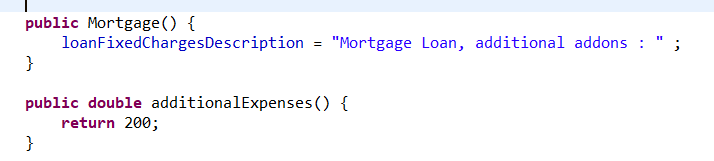


Figure 44 Constructor and methods that were added to each class

I then created a class called additional charges, that would be responsible for returning the description and the total cost of all the fixed charges that were being calculated on loans.

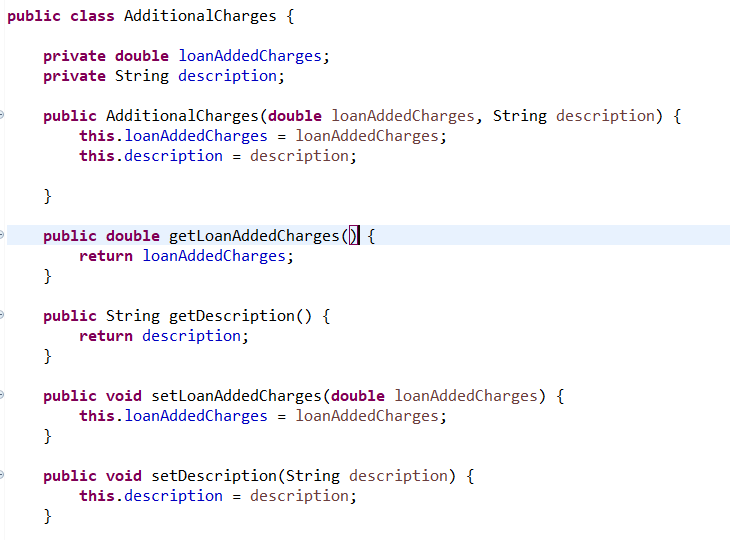


Figure 45 Additional Charges

Finally, I created a controller that would query the most popular types of loans and would return their description and the total cost of all the fixed charges. The controller classed looked as follows.



Figure 46 Loan Controller first method

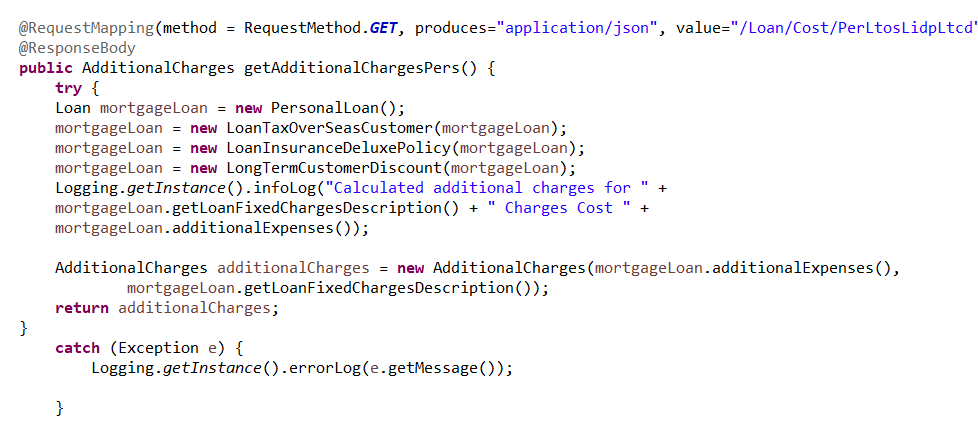


Figure 47 Loan Controller second method

The most popular types of loans were a mortgage loan, for local customers who purchased a standard insurance policy and had been with the bank no greater than 15yrs. The 2nd most popular loan was for a personal loan, for an overseas customer who wanted the deluxe insurance policy and had been with the bank for 15yrs plus.

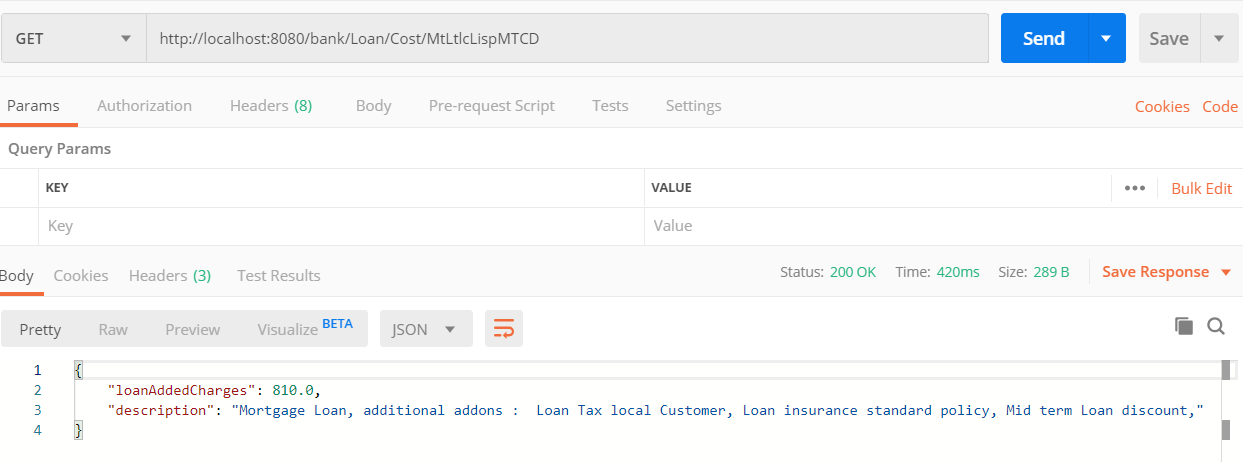


Figure 48 most popular endpoint being hit JSON values being produced

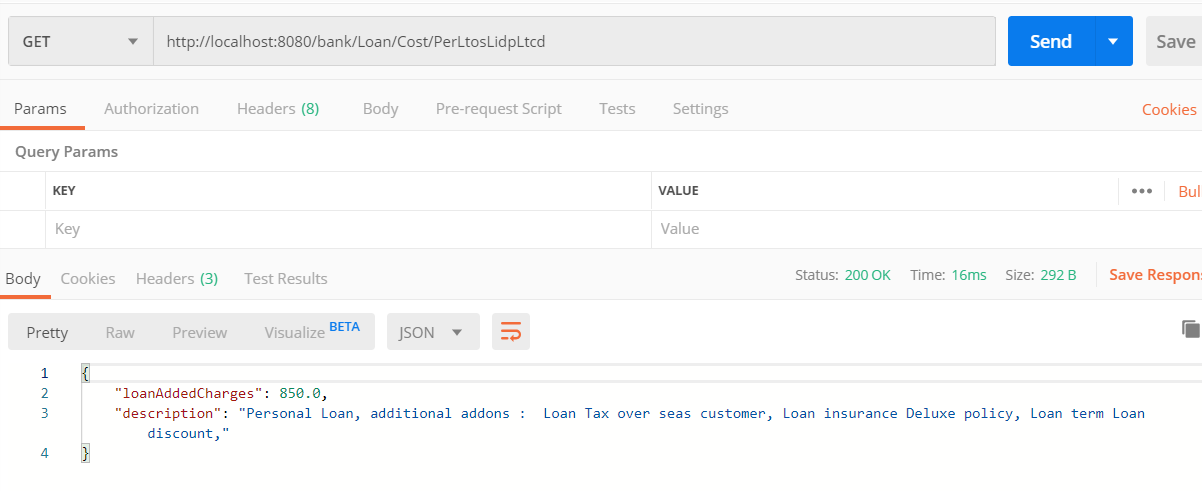


Figure 49 2nd most popular endpoint being hit JSON values returned

## Decorator Conclusion

The main advantages of the decorator is that it is an alternative to sub classing. It allows you to extend an objects behaviour without having to create a new subclass. The main issues that it solved in the context of the bank application, was the OCP and class explosion. However, the decorator does have its drawbacks. Imagine it was the mortgage loan class that had a discount associated with it that calculation wouldn’t have been implemented correctly. This would have made me rethink even using the decorator. Something of interesting note is that the decorator is typically created by other patterns such as the factory and builder.

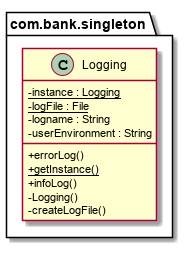
# Singleton Introduction

The singleton patterns intent is that “The Singleton Pattern ensures a class has only one instance and provides a global point of access to it.” This means that every time you wish to use this class that you are using the same instance. So how is this accomplished? Well, we can take a class and let it be responsible for managing an instance of itself. Then, in order to get an instance of that class you have to go through the class itself.

## Singleton Use case

At this stage, the banking application was starting to become quite large, since a high priority in banking would be focused on security and traceability of all errors that may occur. The bank would also like to keep track of what operations were being performed and what information was being received. With this in mind, logging functionality was proposed, that would facilitate a log for historical information as well as for information about system errors. The goal was to create a logging class that would handle the applications logging needs. However, we need to instantiate this class every time that we write a log and it is something that would occur throughout the system and as I mentioned earlier, this could cause problems with a systems memory usage. This problem is compounded by the fact the system has a high number of concurrent users meaning it really could cause a strain on the application. The solution we proposed is having only 1 log instance and share it around the entire application. At this point, we began implementation of the singleton.

## Singleton UML Class Diagram



## Singleton Implementation

The singleton is a class that is responsible for controlling instantiation of itself. It also provides a point of global access. I created the Logging class as my singleton the contents of which are as follows.



Figure 50 contents of Logging Class part 1

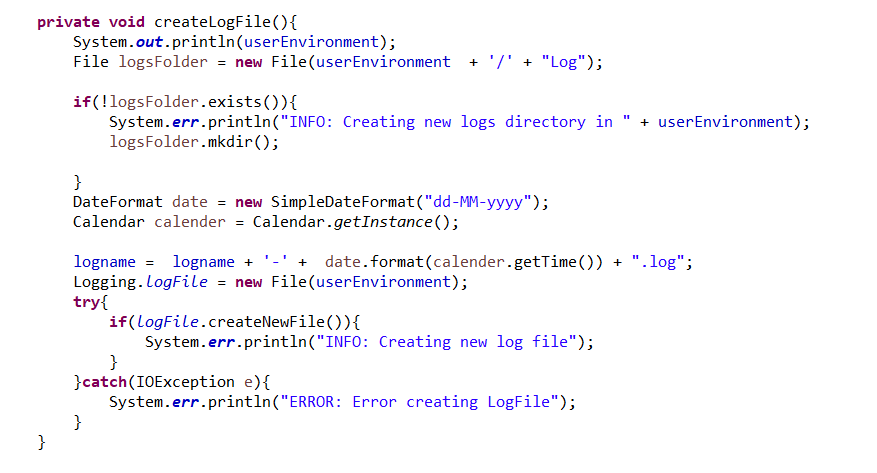


Figure 51 contents of Logging Class part 2



Figure 52 Contents of Logging Class part 3

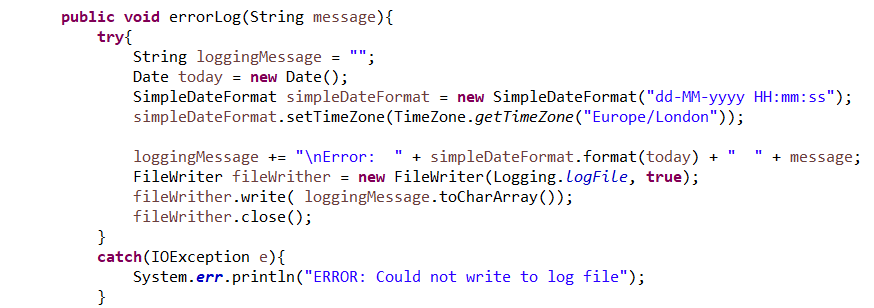


Figure 53 Contents of Logging class part 4

The Logging class was coded in a way to stop multithreaded concurrent access and deadlocking. How this was achieved was through use of the key word synchronised. Instead of using it in on the method, it is only used if the instance is equal to null. To do this every time, and not just on the first instantiation, it would have added additional overhead. The implementation that I used is referred to as double checked locking. The private constructor will stop any class from creating an instance of Logging rather the only way to access it will be by going through the class itself. Following this, I created a method for creating Log files and logging information about the system, as well as errors that could occur in the system.

Following creating the Logging singleton class and all the methods that it would use, I added Logging functionality to the rest of my application. Below, is a screenshot of Logging being performed in the LoanController. This is the same controller that was used to show off the decorator pattern. Here, the logging is being used to display information about the processes the system is performing as well as error information in the case that something went wrong. The information that should be getting logged, is the total cost of the fixed charges plus a description of all the fixed charges that have been accumulated. It should log that total fixed charge is 810.0 and description is Mortgage Loan, additional addons : Loan Tax local Customer, Loan insurance standard policy, Mid term Loan discount,. Below, is the endpoint being hit and a JSON value being returned.

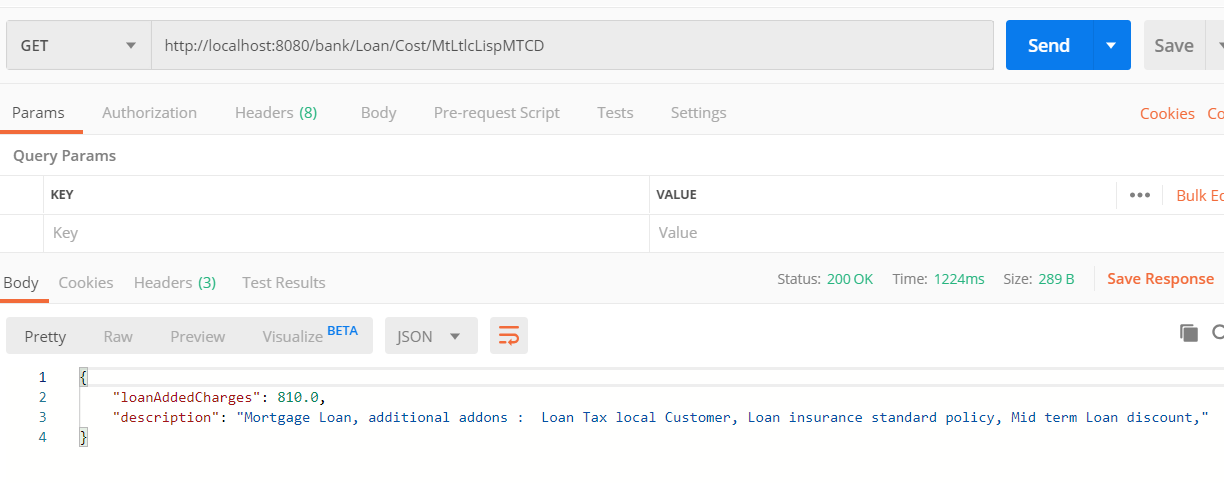


Figure 54 Loan Controller endpoint being hit

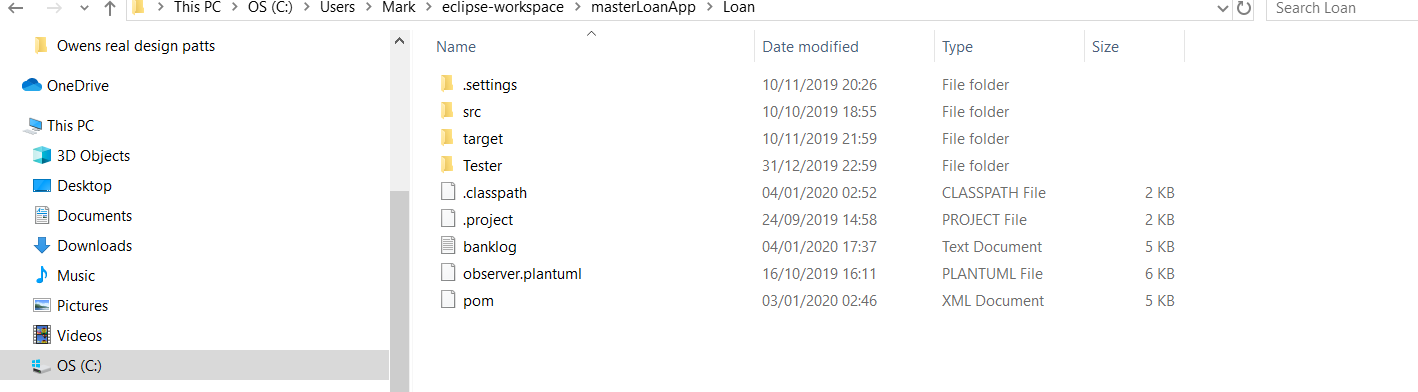


Figure 55 bankLog

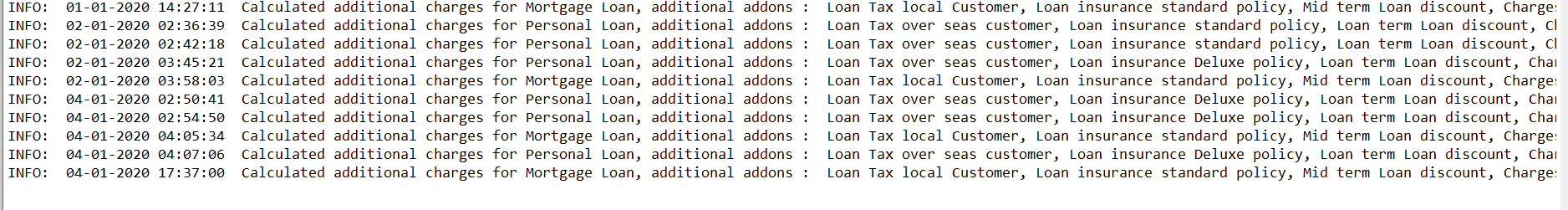


Figure 56 bank log contents bottom most 1, is the 1 that was used in the example above

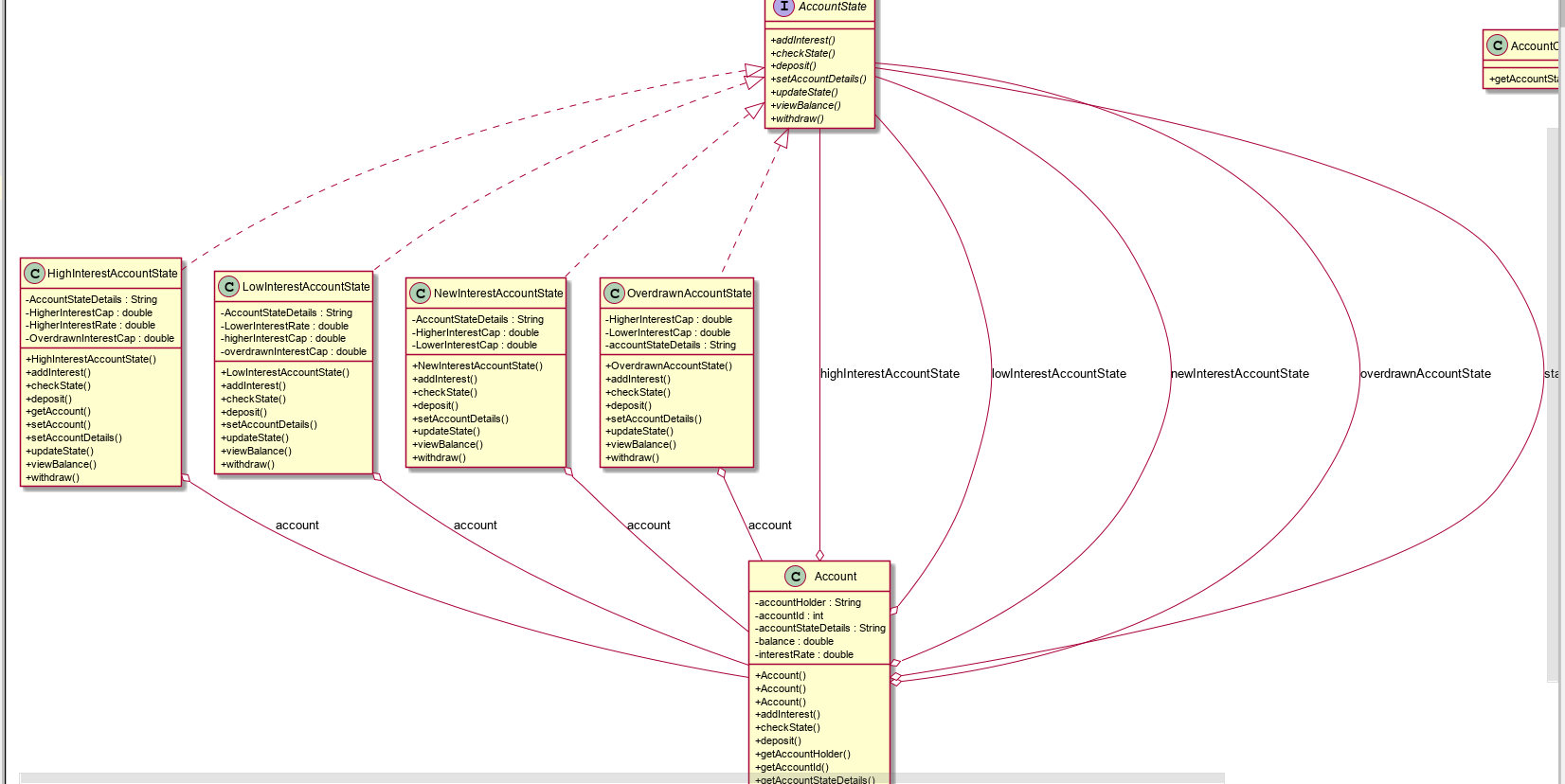
Here it can be seen that my singleton is behaving as intended.

## Singleton Conclusion

The singleton pattern is a simple pattern in terms of a UML class diagram. It only consists of 1 class. However, don’t let this fool you, the singleton can be tricky to implement correctly and should be used whenever you need control over the number instances of class you are creating. The singleton is considered an Anti-pattern as it often brings more disadvantages the advantages. It also doesn’t satisfy any of the OOP principles that the other patterns. A singleton can still be accessed through the use of reflection, but it does make it harder to get at. Also, a singleton can’t be subclassed as it has a provide constructor, in order to allow it to be subclassed, the constructor would have to have an access modifier of public or protected. However, this would mean it’s not a singleton any more as other classes could use it.

# State pattern Introduction

The intent of The State Pattern is it allows an object to alter its behaviour when its internal state changes. The object will appear to change its class. How the state pattern achieves this intent, is by encapsulating state and delegating behaviour to the object that represents the current state. “The object will appear to change its class” what is meant by this is that if an object completely changes its behaviour it gives the impression that the object is instantiated from a separate class. However, this is not the case. The state pattern uses composition to make it appear like a class change when in reality we are referencing different state objects.



Sidenote had to delete some getters and setters to fit diagram

## State Pattern Use Case

The bank received a new set of requirements. These requirements were in relation to creating new accounts. They wanted to store standard information on accounts such as, account holder etc and also allow an account to accumulate interest and a customer be able to withdraw and deposit money. Additionally, they had a list of requirements about the behaviour of an account based on its state. These are a long list of requirements and I will discuss them before showing the concrete state classes as I can provide pictures as context at that stage. First, I needed to define what method were common to all states and abstract them. I decided on an AccountState interface. This interface held all the functionality of the accounts.

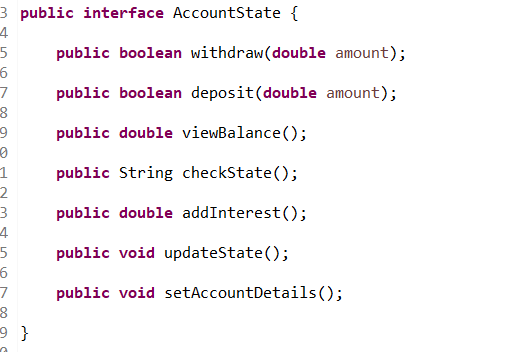


Figure 54 Account State interface

After creating the account state interface, I was ready to create my concrete states. There were 4 states that an account can through in total these are showed below. Just so I don’t have to repeat myself all concrete states implement this interface. They all have a reference the Account context class which I will speak of later and they are passed a reference of the account class through their constructor.

### New Interest Account State

This is the first state that an account can enter. When an account is created, it enters this state. However, if the account is created with money already inside it as in, they made the account in the bank and deposited at the same time they can enter 1 of 2 states dependant of the amount of money. These being low interest account and high interest account. The only way for a new interest account to change state, is through the method withdraw. Below are what each method does while in this state.

**Withdraw:** returns false as you cannot withdraw while in this state

**Deposit:** sets the balance of the account. Calls the updateState() method and returns true

**Viewbalance:** returns 0 as a new account can’t have an amount in it

**CheckState:** returns current state of the account

**addInterest:** returns 0 as a new account can’t gather interest

**updateState:** is a method for checking if the account can change state. If money is deposited it can. The two states that it could possibly change to are high and low interests accounts

**setAccountDetails:** sets the accountdetails field to reflect the current state of the account.

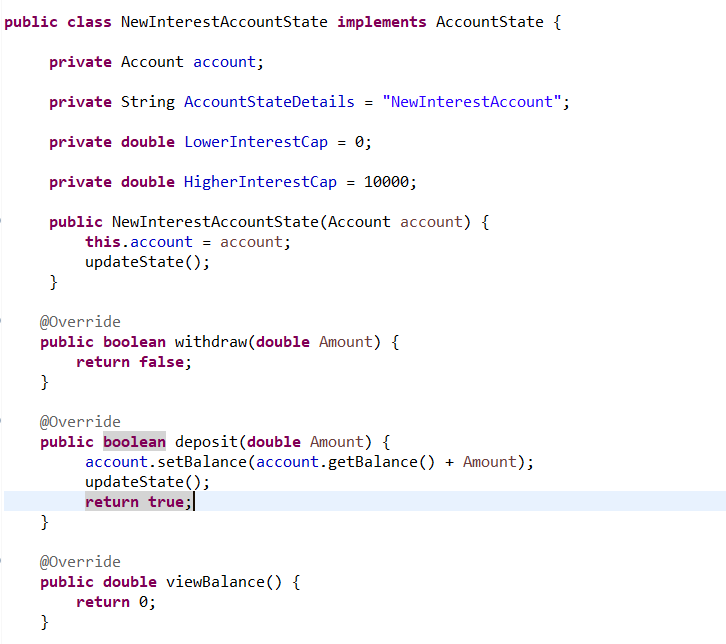


Figure 57 New interest account state part 1

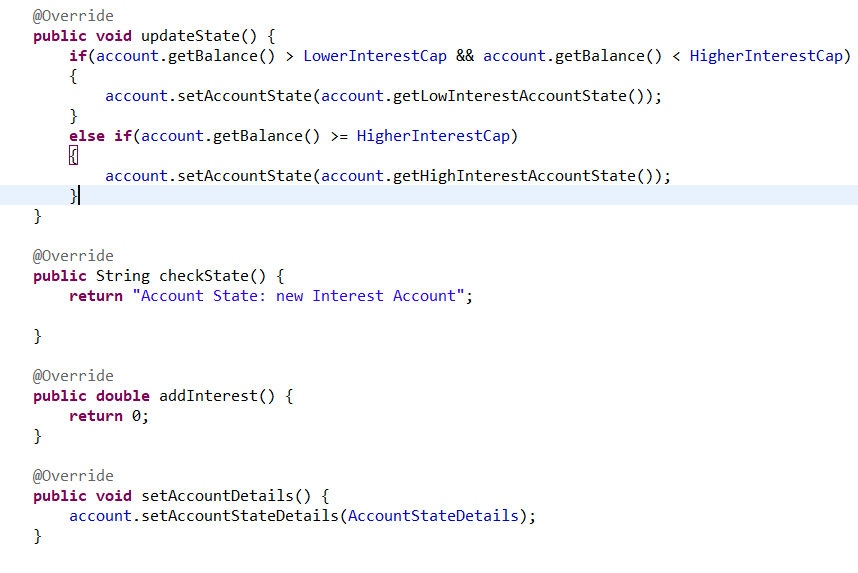


Figure 58 New Interest state Part 2

### Low Interest Account State

A low interest account state is an account that can gather low interest. It can change state by either depositing or withdrawing. The 2 states that it can enter are either high interest account state or an overdrawn state.

**Withdraw:** sets the balance, calls the update state method to see if a state change is required and returns true.

**Deposit:** sets the balance, calls the update state method to see if a state change is required and returns true.

**Viewbalance:** returns the accounts current balance

**CheckState:** returns current state of the account

**addInterest:** adds 4 percent to the account balance as this is the interest rate for the account.

**updateState:** is a method for checking if the account can change state. If money is deposited or withdrawn, the account may be eligible for a state change depending on the conditions. The two states that it could possibly change to are high interest account and overdrawn account

**setAccountDetails:** sets the accountdetails field to reflect the current state of the account. Also sets the current interest of the account based on its state.



Figure 59 LowInterestAccountState part 1

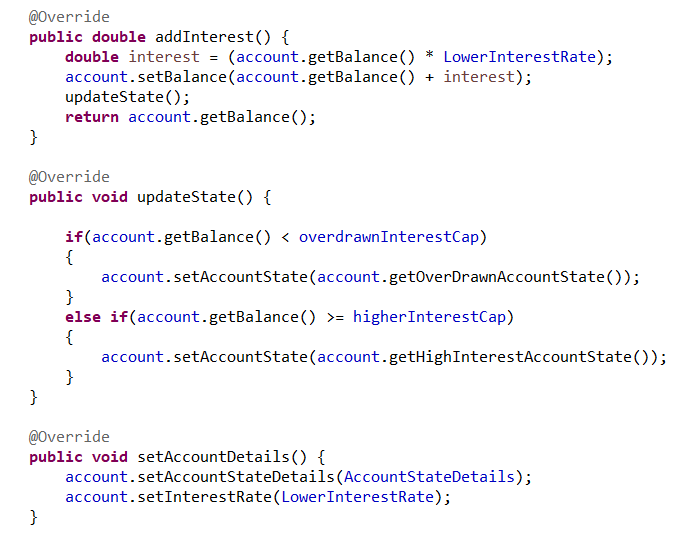


Figure 60 LowInterestAccountState part 2

### High Interest Account State

A high interest account state is an account that can gather a high rate of interest. It can only change state from withdrawing money. It can only enter two states these being low interest account and high interest account.

**Withdraw:** sets the balance of the account, calls the update state method to see if a state change is required and returns true.

**Deposit:** sets the balance and returns true.

**Viewbalance:** returns the accounts current balance

**CheckState:** returns current state of the account

**addInterest:** adds 5 percent to the account balance as this is the interest rate for the account.

**updateState:** is a method for checking if the account can change state. If money is withdrawn, the account may be eligible for a state change depending on the conditions. The two states that it could possibly change to are low interest account and overdrawn account

**setAccountDetails:** sets the accountdetails field to reflect the current state of the account. Also sets the current interest of the account based on its state.

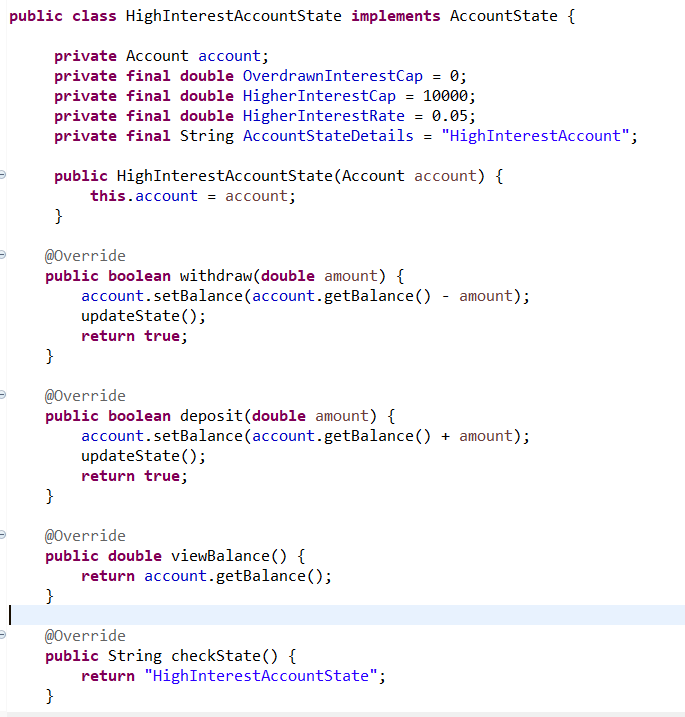


Figure 61 High Interest account part 1

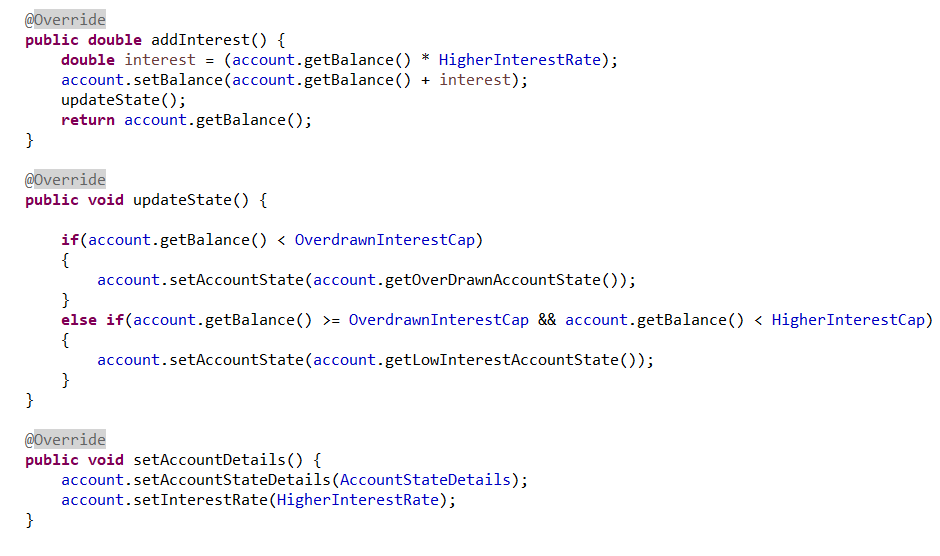


Figure 62 High interest account state part 2

### Overdrawn Account State

This is the state that an account enters when the balance falls below zero. Only a deposit that brings the account above that threshold will bring it out of this state. An overdrawn account can enter 2 states low interest state and high interest state.

**Withdraw:** can’t withdraw while in this state returns false

**Deposit:** sets the balance, calls the update state method to see if a state change is required and returns true.

**Viewbalance:** returns the accounts current balance

**CheckState:** returns current state of the account

**addInterest:** returns 0

**updateState:** is a method for checking if the account can change state conditions. The two states that it could possibly change to are low interest account and The account can only change state by depositing money. The 2 states that it can enter are the low interest account and the high interest account.

**setAccountDetails:** sets the accountdetails field to reflect the current state of the account.

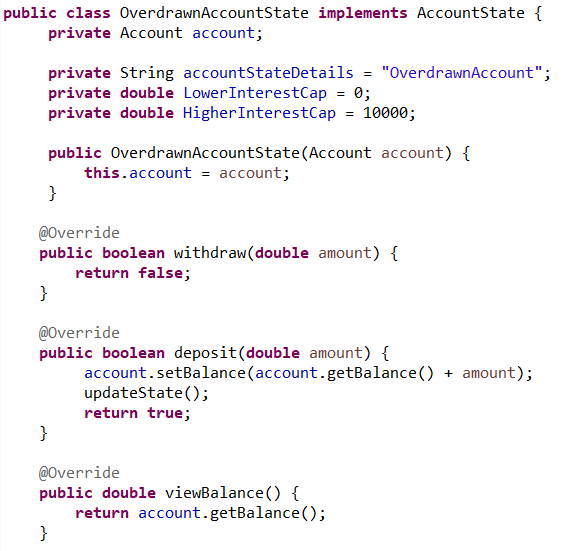


Figure 63 Overdrawn account state part 1



Figure 64 Overdrawn account state part 2

## State Pattern Context Class

Following this, I updated the context class account.



Figure 65 Account context class part 1

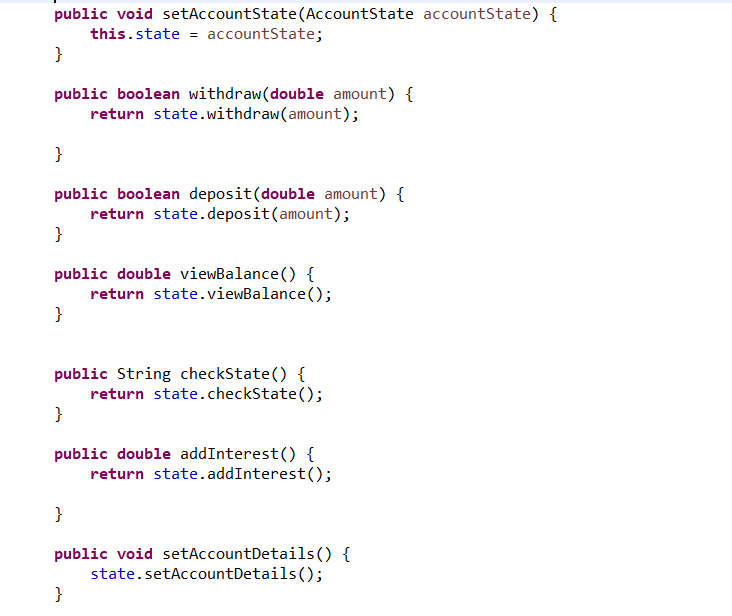


Figure 66 account context class part 2

The context class also included all relevant getters and setters. As can be seen above, the context class holds references to the various states and the state instance variable which begins in a state of new interest account. The constructor also sets the state instances of each state. The methods simply delegate all the actions that can be performed on an account to the current state.

## Demoing the State pattern

First, I will begin with a simple driver that will show the state pattern working as intended.

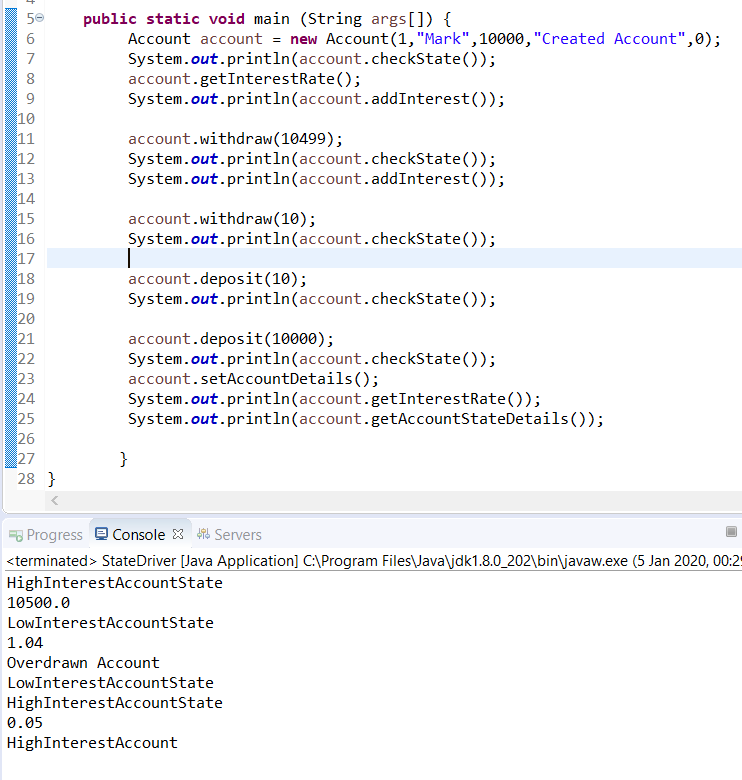


Figure 67 simple driver state pattern

As can be seen, an account is created and 10000 euro deposited. The state of the account is checked, and it correctly shows that it is in high interest account. Interest is added onto the account, that being 5%, and it correctly shows the new balance is 10500. 10499 is withdrawn leaving the balance at 1 euro. The state is now checked and it shows that it is indeed in low interest account. When interest is added it pushed it up to 1.04 euro which is the interest rate for that state. After 10 more euro is withdrawn, the status displays that it is an overdrawn account. After the 10 is put back in, the state changes to low interest account. After 10000 is put in, the state and interest rate is checked showing that the account has changed state correctly with values of high interest account and 5% interest rate being returned.

The show this working in a more real-life scenario.

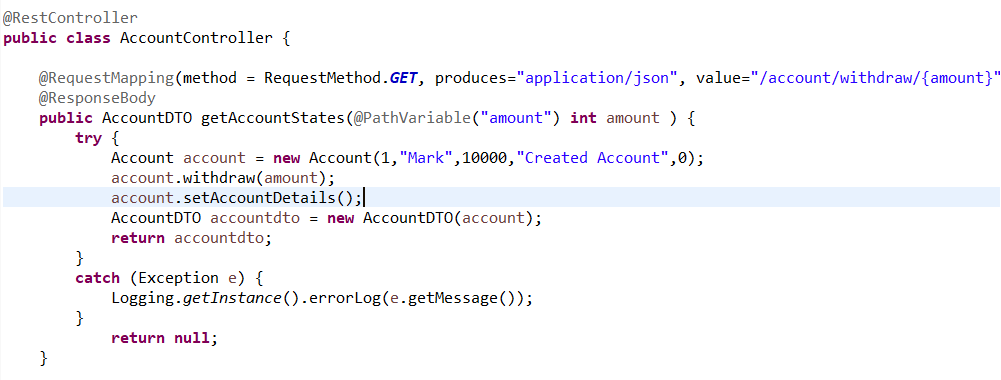


Figure 68 AccountController contents

In the controller above, a method has been set up to take in a withdrawal. Since its account balance is originally 10000, the account will be a high interest account. However, we will request a withdrawal of a certain amount changing the state of the account and see if the JSON value that is being returned shows this.

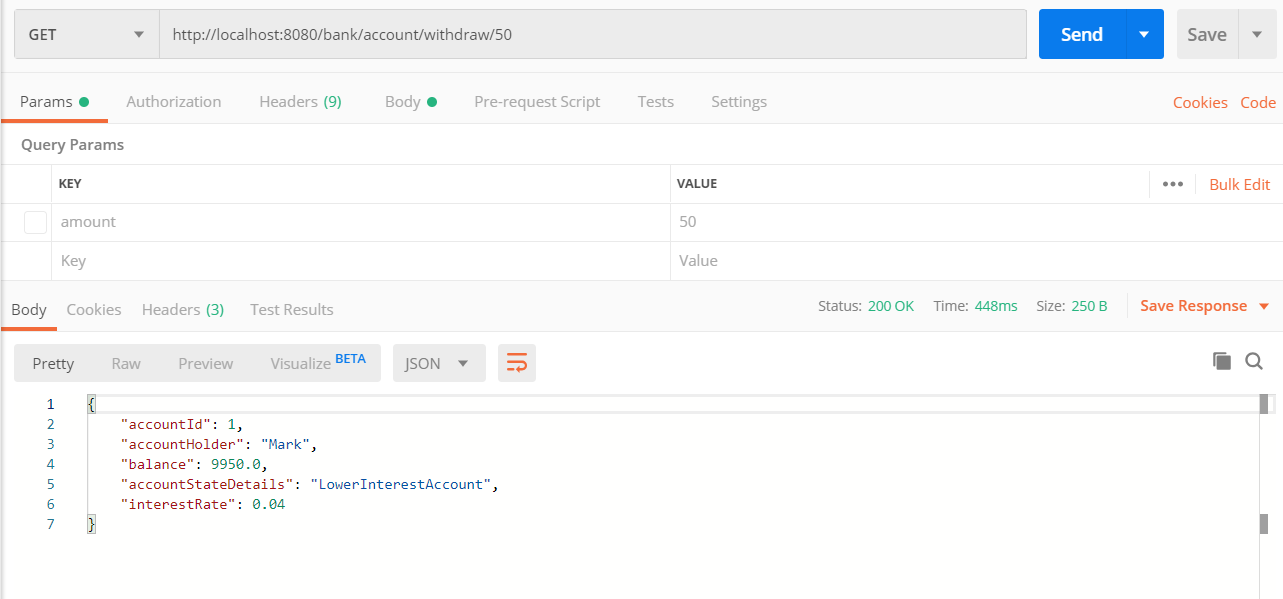


Figure 69 account controller endpoint being hit

Here, we can see that the endpoint has successfully been hit and that 50 euro was passed over to the controller. The response returned a JSON value that shows that state has indeed been changes and the client can display to the customer the new state of their account.

## Benefits state pattern brought to the application

The benefits of implementing the state pattern were numerous to the application. Without the use of the state pattern, the account class would have been full of conditional ifs managing the many states the account can be in. These would have been difficult to maintain and if other states were introduced would have violated OCP in a big way as most of the method such as withdraw etc would have had to been changed.

It also meant that each state could be closed for modification, but the Account context class open for extension. In this way, additional states could be added rather easily without fear of breaking pre-existing code.

In terms of read ability, there has also been a massive improvement. The account class would have grown so large and unreadable that it would become difficult to alter it in the future. The state pattern provides structure to this a new developer can read the code and easily infer meaning.

## State Conclusion

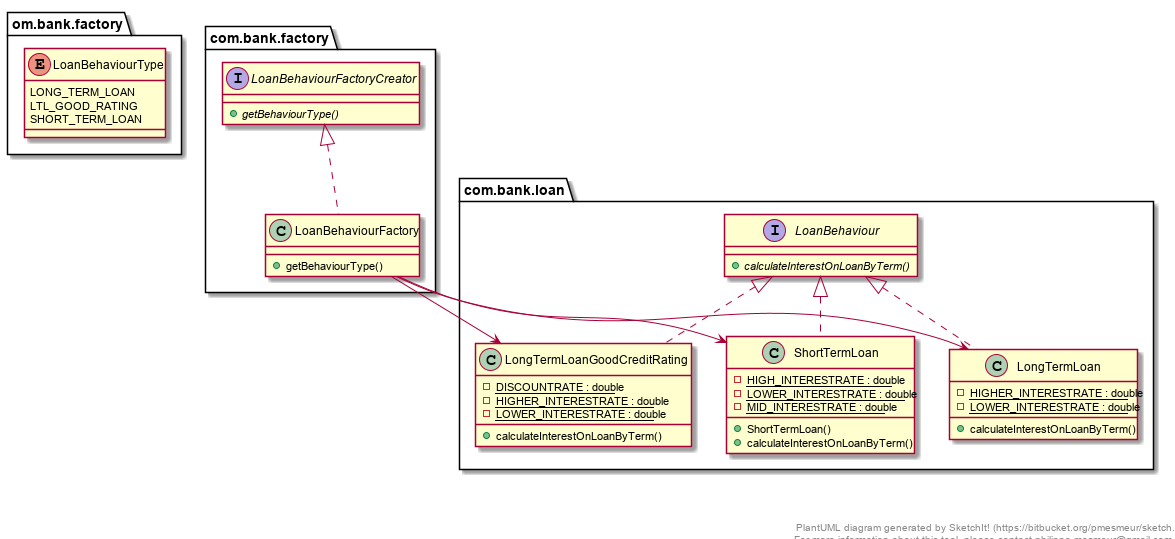
The state brought a lot of benefits to the application so in my view it was worth implementing it. The State and strategy pattern are very similar patterns with their UML class diagrams being almost identical. One could argue that they accomplish similar goals but with key differences. Strategy’s use of algorithms is client chosen which varies drastically from state were the client interacts with the context (Account) but has no control over which state to choose.

Another thing that I noticed, is that the order of transition and when it will happen is well defined in State making it a perfect fit in this case cause the state changes are well defined. Strategy has no such structure like this placed on it, it is free to implement whichever behaviour that it wants.

# Factory Method Pattern Introduction

The intent of the factory method pattern intent is the Factory Method Pattern defines an interface for creating an object, but let’s subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

## UML Class diagram Factory method pattern



## Implementation of the factory method pattern

I wanted to implement the factory method pattern with my strategy pattern. The strategy pattern was built around the idea of loan behaviours. With this in mind, i began by creating an abstract factory creator. In this way, its subclasses would be responsible for its creation.

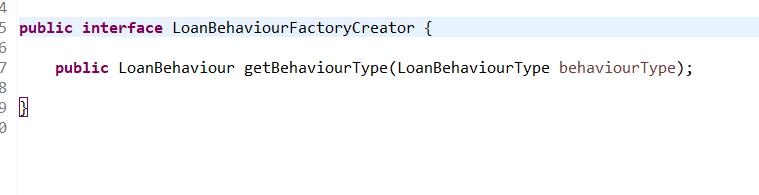


Figure 70 LoanBehaviourFactoryCreator interface contents

Following this, icreated an Enum that would ensure type safety and readability. This enum is then passed into the getBehaviourType() method. Below, are the contents of the enum class.

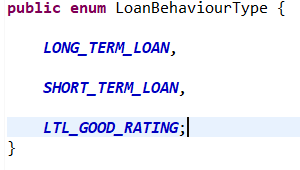


Figure 71 Contents Of Loan Behaviour type

These represent the 3 algorithms that can be calculated on loans. After this I created

Concrete creator class that implements the abstract creator interface. This is the class that will be responsible for applying which algorithm ‘Behaviour’ to use. Below are the contents of concrete creator class.

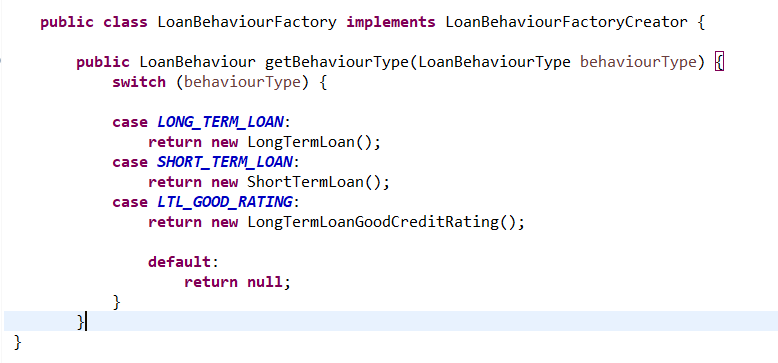


Figure 72 contents of LoanBehaviourFactory

As can be seen, the logic for creating loan behaviours is contained within one class. This satisfies the SRP principle that states that a class should be responsible for 1 and only 1 thing. It satisfies this by encapsulating all the creation of behaviours in 1 class as the logic for this would have spread around the application otherwise. The other classes that regard the strategy can be seen in chapter 2. These consist of the abstract product and concrete product classes (LongTermLoan) etc. Below, we can see the factory method pattern in action.

## Conclusion Factory Pattern Method

The factory pattern method satisfies two very important principles in OOP, these being the OCP and the SRP. It satisfies OCP by allowing you create additional functionality without having to modify the system. It satisfies SRP by encapsulating all the creation logic in 1 class and as I mentioned earlier this creation logic would have spread around the application.

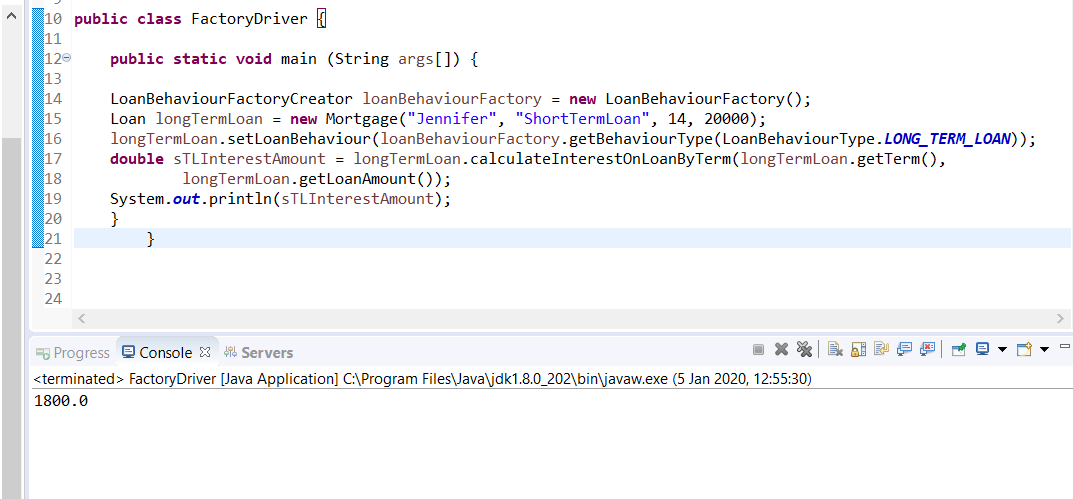


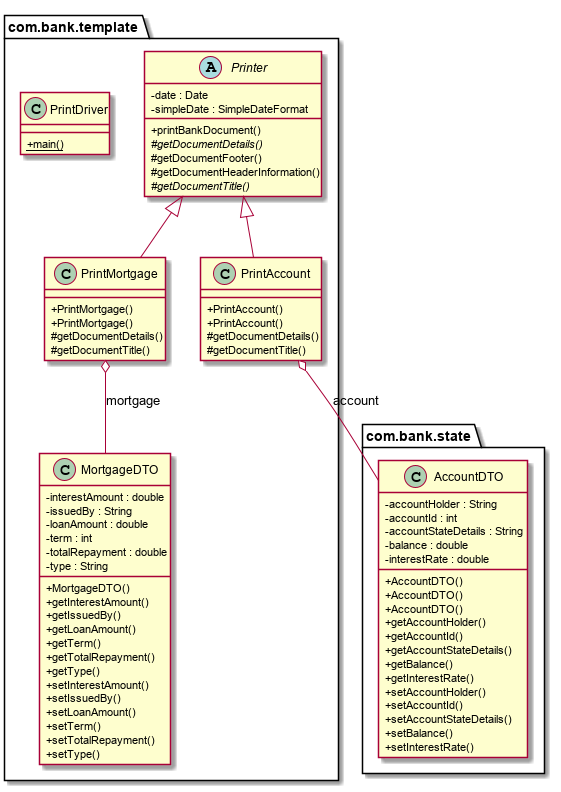
Figure 73 Factory pattern method driver

Here, we can see that the responsibility for creating the new instance of the behaviour is that of the loanBehaviourFactory. Here the factory has got the correct behaviour and applied it to the mortgage loan producing the expected result of 1800.

# Template Method Pattern Introduction

The strategy patterns intent is The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

## Template Method Pattern UML Class Diagram



## Template Method Pattern Implementation

I implemented the temple method pattern to define algorithms that would be needed to print out official bank documents. 4 distinct algorithms were defined for the creation of any document these being document header information, document title, document details and document footer information.

I began by creating a printer class I made the class abstract as I would need to have subclasses to provide their own implementation of certain method.



Figure 74 Printer abstract class part 1

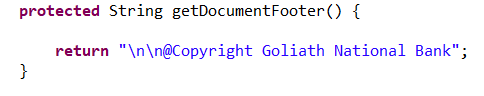


Figure 75 Printer abstact class part 2

As is clearly evident, 2 methods provide their implementation and 2 methods, the abstract ones, will get their implementation from their subclasses. I then proceeded to create 2 subclasses of printer because the bank wanted to be able to print Account documents and Mortgage Loan documents. The print bank document is a final method and will define how all documents are printed providing a template in essence

The 2 subclasses that implement the abstract Printer class are

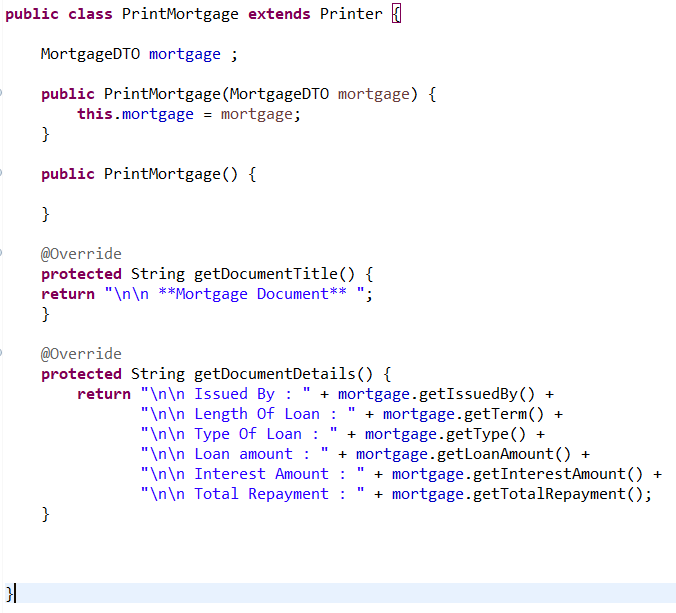


Figure 76 PrintMortgage extends the printer

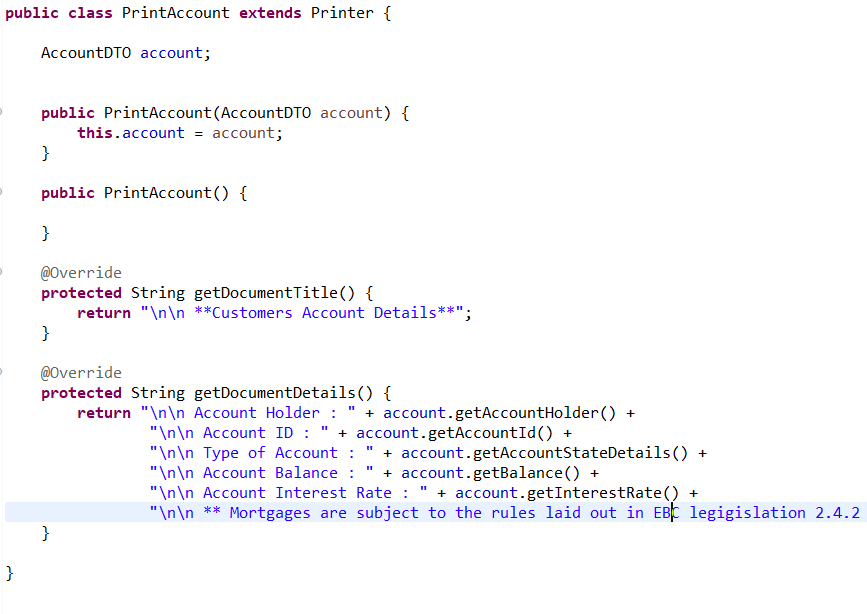


Figure 77 Print account extends abstract class Printer

As can be seen, both provide their own implementation of the abstract method from the printer class.

## Template Method Pattern Runtime

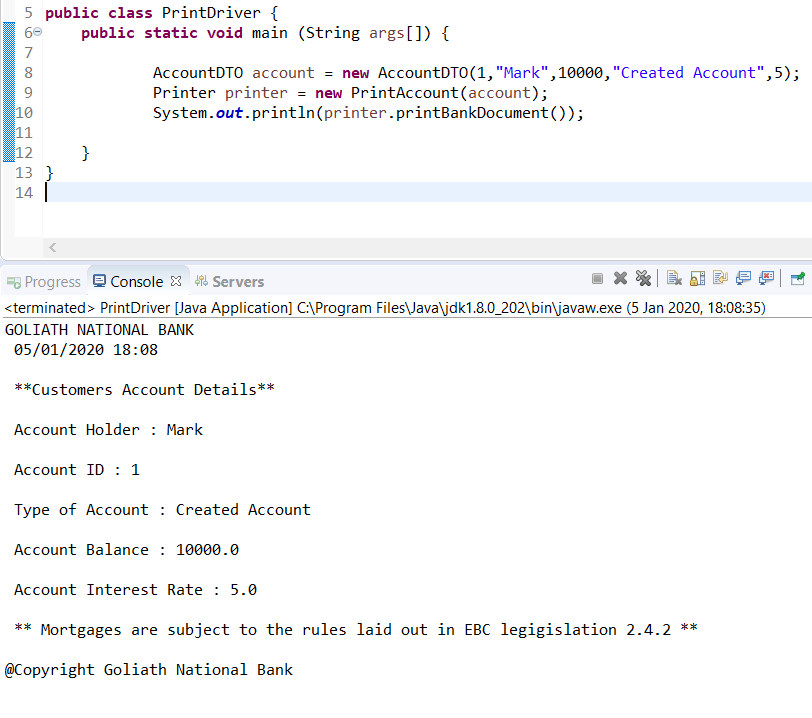


Figure 78 driver results

As can be seen, I use polymorphism for the object to take on the type PrintAccount and call the template method. The print Account class then provides its algorithms, and these are the ones seen below in the console log.

## Benefits of the template pattern method to Banking App

The template provides a structure that is need for this use case. The algorithms are executed in a predefined order and all printed documents must adhere to this order. This is perfect use case for this uniformity as the bank wanted this.

It also uses the Hollywood principle of, don’t call us we will call you. This is because we are telling the subclasses not to call the parent class. The parent class will call the subclass when it needs it

Finally, to template method pattern, makes it really easy to add additional printing functionality for the 2 different abstract method. If a new bank document was required to be printed, its as simple as extending the printer class and providing your own implementation of the two abstract methods.

## Conclusion Template Method Pattern

The template method was ideal for this use case, as printing documents needs structure something that the strategy wouldn’t have been able to provide. In my opinion, the template would only fit into use cases that were very structured and extremely unlikely to change in the future. The template method cut down on the duplication of code also as all code that stayed the same was inherited by the subclasses. The subclasses only provided their own implementations when there was a need for change.

# GitLogs

commit af2f483c6fb73dd9aa537aedb9505ac1a1ae9c1f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 19:09:20 2020 +0000

Final commit to ensure repo is up to date

commit 5c2b73d5ee47f4718fe017d704b5acfce01deab8

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 18:07:05 2020 +0000

changed acces modifer of print document

commit 5a0e1d89fe027ea5ff700921d35f6a7bb79bbf39

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 17:34:16 2020 +0000

Added additional guidelines to mortgage template class

commit a901579bc0c2e1fed69c8930d3c3f09fe8f2fcce

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 17:24:34 2020 +0000

Added classes in relation to template method consists of printer as well as two seperate ways of printing

commit f8e7122a76f9345eb0df24694f5f30ae7e910b94

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 15:19:25 2020 +0000

Added unit testing to the strategy pattern tested all possible outcomes

commit 2379d145d709ce0a2ea3804c8a342c69e352cf47

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 14:05:03 2020 +0000

introduced an abstract interface that the concrete factory implements

commit 43319d872dace4c33ccafaf18da95669b8405af1

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Jan 5 03:13:36 2020 +0000

added Account Controller method

commit ec938034177320f9d792c410b47fbcee9735642e

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sat Jan 4 20:30:11 2020 +0000

created branch to house entire apllication

commit f1ec5a04e55197f79876e647c52eef413caab2ca

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sat Jan 4 20:09:46 2020 +0000

making sure state hold most recent version of the code

commit 68b41312c091521389df182743065731818b4907

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sat Jan 4 19:19:28 2020 +0000

Small refactoring changes

commit 0758e027dc8dcd88897e0928a0f7356690a53603

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sat Jan 4 03:08:17 2020 +0000

altered decorator so that the base class is loan as oppossed to loan fixed charges

commit e945c449f68c80f34c4b74ef3f3964a5ab1eb219

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Jan 3 13:18:57 2020 +0000

Updated persistence xml had to use a different hosted db as the one I was using closed down

commit 27c02a78bee9b2f005df02aac7550843c3a5d591

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Jan 3 05:07:10 2020 +0000

added both observers to contain references of subject interface

commit 81c23e2c4026f8864be1972686650fcc6d9baf99

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Jan 3 02:50:11 2020 +0000

changed observers from type ArrayList to type List

commit 0933f579d59c4f2e481f5c2a675f1ef93d5d5f30

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Jan 2 23:55:51 2020 +0000

Added LongTermLoanGoodCreditRating that provides its own implemnetation of LoanBehaviour

commit 99d74000d14d97e67c71f82477e0d9a19754b3ab

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Jan 2 04:25:40 2020 +0000

Added account controller which will demonstrate singleton and State pattern. Returns an account JSON object that will reflect the interest rate and current state

commit 8f39e41d88fcfd4f58c6c2b1e14b6d6175e4518a

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Jan 2 04:02:58 2020 +0000

Added another endpoint for decorater pattern

commit f9a885952d09840ed33ef738e720411ac62ea9e2

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Jan 2 02:05:06 2020 +0000

added setAccountDetails to all classes to ensure that an account reflected its state and current interestRate

commit 67b562363785c9da4297eb72847178cda58a1f0b

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Jan 2 00:20:35 2020 +0000

fleshed out the behaviours of the remaining accounts state pattern implemented

commit e2a1fbbf78f0e50d0dd5f42f6e291a2db6392f81

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Jan 2 00:03:36 2020 +0000

Fleshed out all methods of lowInterestAccount to allow for correct state changes

commit d521ecafebbc26851b966610f699e96f8256f5a2

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 23:50:10 2020 +0000

Fleshed out all methods Of newAccount state

commit 82c04127b1c2e4301b3da48aa320e4f83c9ea1b9

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 23:26:12 2020 +0000

Added a update state method and addInterest to mimic real life accounts

commit df9a6e53ebc130c1fa78ee79c6b1edbc7cf21f53

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 23:08:57 2020 +0000

added references to the state in all method of context class and configured functionality of newAccount

commit 608eea036261e7a3d1148a8896ced7eb5b88716f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 18:47:35 2020 +0000

has all concrete classes implement AccountState and provided refs to Account context Class

commit dcbacc1c321c805c03200d4679b07b0954db270f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 18:40:37 2020 +0000

had HighInterestAccount take on method of state

commit 0dcbd8048a9019412ae38ce2d9896c3dd9b300fd

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 17:10:24 2020 +0000

added context object account class

commit 74c21dee459d3d71209f7daeeb7fa952f96166f6

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 16:03:40 2020 +0000

Created four state concrete classes

commit f38eb2b46e26cae70012175fd3466633fab0df99

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 15:23:17 2020 +0000

Created Account State interface to encapsulate methods that the various account states will use

commit 01e79cb4d70e83224546d5a39b56d665dca139dd

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 14:53:13 2020 +0000

refeactored Logging class to allow it be used by a controller created additional expesnses class to return a decorated object

commit 088bf91b149adcfeb4f1329e099bc9f5645eea10

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Jan 1 00:42:04 2020 +0000

Added error and info logging. Created Loan Controllor. Created endpoint that incorporates Dec and Singleton

commit eabbc272454eb21e97cc3ab249642033035b7135

Author: Markslaw <Shaunalily9@gmail.com>

Date: Tue Dec 31 22:19:40 2019 +0000

added singleton Logging and driver class created privat constructor and logging methods

commit 321e5db9f8638cc7bfe7e62402b5f1ed7e1f18b4

Author: Markslaw <Shaunalily9@gmail.com>

Date: Tue Dec 31 03:04:27 2019 +0000

added fatory pattern delegated responsibility of instatiating loan behaviour

commit b08a9f188cca3e41c7b2d5e107bb8b78f0a67f51

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Dec 30 21:51:18 2019 +0000

Added Factory method plus Enums for type safety

commit cfa2d8cdbd29a8ba7e7558cafcf086e151942244

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Dec 30 17:20:07 2019 +0000

Added enum class and also created a driver

commit ee8d36c17995b6bfe901c28594b6c9e187c75275

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Dec 30 16:52:45 2019 +0000

Refactored decorator classes to make them more real world

commit c94c3fe4421a4ff4bbe378ffe38fb4f358a37514

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Dec 30 14:33:49 2019 +0000

created Loan factory class

commit 69c035a20831b1a7f0e67aa8b0c06656d10bf820

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Dec 30 13:33:19 2019 +0000

Fixed issue with calculating discounts on customers

commit 46782fca255ae96f7fd58275a862901f6343491a

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Dec 30 13:27:43 2019 +0000

Finished implementing decorator created Loan fixed charges interface plus Loan Decortaor abstract class and decs

commit e3f719a7ec9de10baaaab8087c73d0ee3472a86e

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Nov 14 00:55:27 2019 +0000

added the loan decorator plus the loan insurance class as well as the LoanTax class

commit fc7da48847491e7e13575730933d38cd658a88f2

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Nov 13 22:17:07 2019 +0000

Structuaral changes to application removal of spring boot and completed the observer pattern

commit d2716919a4b89df7d0189426f832dc325f52ebb7

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Oct 11 21:46:52 2019 +0100

Controller finished and Data Transfer Object created

commit 8846c7b764570ad2b981cd5adb7ad4dffdd39d0f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Oct 11 18:15:18 2019 +0100

Git customer dao returning a customer entity

commit 3f00312118a09eb2fb5b978402952aa8d818bd19

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Oct 11 01:35:48 2019 +0100

added orm mapping tool hibernate added pesistence.xml file added persistent unit which references my entitymanger

commit a2dc5c37f9bdb8d30776ac1b917b5ae67bb6809b

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Oct 10 22:18:40 2019 +0100

Added Transaction entity

commit fd35920daf05f7e361b40ee50730b96eafdc8cdf

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Oct 10 21:59:23 2019 +0100

Created Customer Entity

commit 8d4eb3719c155ac20452b457f1f5ceb7f6c92ec7

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Oct 10 21:37:39 2019 +0100

Added transaction Controller

commit f2114bb076af586063f6666f5e7b3b1d43d015eb

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Oct 10 19:16:43 2019 +0100

Converted Project to java web application and added spring boot

commit cf41620d87ef9c87aa81c96b3bd8b1e37333c70b

Author: Markslaw <Shaunalily9@gmail.com>

Date: Tue Oct 8 00:10:20 2019 +0100

refactered main to search multiple transactions

commit 93d47004727babcbac1dfec8ecb141c6493c47e0

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Oct 7 20:03:51 2019 +0100

Fixed errors after refcatoring classes

commit ded9f98dd76a04b1d33b8c5c7bd733b553f9a45b

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Oct 7 18:58:18 2019 +0100

Added extra params to suspicious transaction and updated observer

commit 5ea3c6a38d95ee5cf02481f537a6b70b4c2d350f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Oct 7 18:28:16 2019 +0100

refactored transaction data class to be more approiatly named also added mobile and email attributes

commit f342c5975a25be953096853c7cfc4be50a9ae148

Author: Markslaw <Shaunalily9@gmail.com>

Date: Tue Oct 1 00:11:47 2019 +0100

created driver for observer pattern

commit 86497b205daaf9e3289fc765d590d66b9cfe6c8f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Sep 30 22:15:13 2019 +0100

Added method in transaction class to determine is a transaction is suspicious

commit a8536bc0e621ee1f1a3dd965451a218216eee8d2

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Sep 30 20:40:33 2019 +0100

Added logging dependencys

commit 1da288ff70bc466de6346f3985419e887fe78bb1

Author: Markslaw <Shaunalily9@gmail.com>

Date: Mon Sep 30 20:00:20 2019 +0100

Added SMSNotification Class that implements Observer interface

commit 6f82e32e3deda29d407972d1415da7455f8b6aff

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sat Sep 28 17:51:58 2019 +0100

refactored email configuration class as well as got email Notification working as an observer

commit 2701e99af15f41a683a1b7667ccaedb721f839bc

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sat Sep 28 13:57:55 2019 +0100

Added Transaction class and TransactionData class which implements Subject interface

commit ff5b80510dce2e2554bb7f22bb471ff95655f8ae

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Sep 27 19:32:15 2019 +0100

Added payment class as well as sent message to personal phone

commit adcdea44e0223be2b7a8a66707f51c5b25a961d6

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Sep 26 21:40:49 2019 +0100

Updated email class

commit 87dcce27862a5ffab8b6ac4d2990af69e5991c85

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Sep 26 21:26:30 2019 +0100

Added Email Notification and EmailConfiguration class

commit f2523399b7d7719984b9fd8a7820bfb21f0ff132

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Sep 25 23:09:14 2019 +0100

Added subject and observer interfaces

commit 9dea07759336ec11833ab3bee01fa537599ba2eb

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Sep 25 22:46:58 2019 +0100

added Customer and Account class in relation to observer

commit cc7309c378acd28e7a9dc26f9ebedb014717654c

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Sep 25 15:41:49 2019 +0100

Added hibernate jars

commit e89084877cc8fa27c2d3b515baec7f62aacecb85

Author: Markslaw <Shaunalily9@gmail.com>

Date: Sun Sep 22 20:03:08 2019 +0100

Code cleaned up

commit c4b38b439def7020bc3ddb8dd6dbdb46c48b491f

Author: Markslaw <Shaunalily9@gmail.com>

Date: Fri Sep 20 00:09:14 2019 +0100

Strategy pattern implemented

commit 0c385700699c018b3af2f596293ef700d8526357

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Sep 19 20:34:16 2019 +0100

added functionality short term Loan

commit c5edf96b84692f6d3aba1faf96e3af0682b362c1

Author: Markslaw <Shaunalily9@gmail.com>

Date: Thu Sep 19 16:55:49 2019 +0100

iteration 1 functional loan application system

commit 0ca7d85833176386faf3bbf36b49465fde3c0878

Author: Markslaw <Shaunalily9@gmail.com>

Date: Wed Sep 18 18:39:31 2019 +0100

initial commit