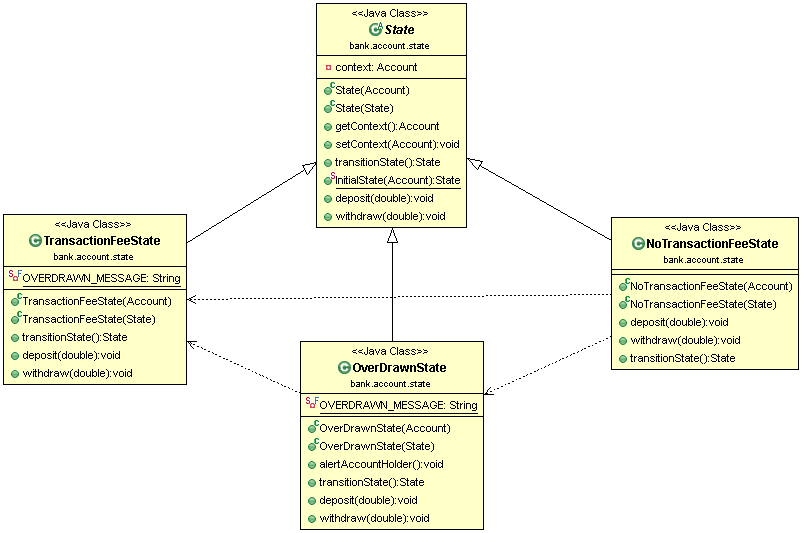
DESIGN PATTERNS PROJECT SUMMARY

Andres Cordoba, Kevin Acevedo, Neal Rideout, Oskar Danielsson, Sanjay Sohal, Tyler Serio

Humber College

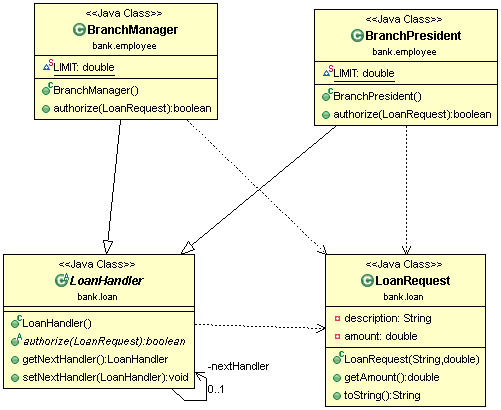
Main Title: Design Patterns Project Summary

# State Pattern



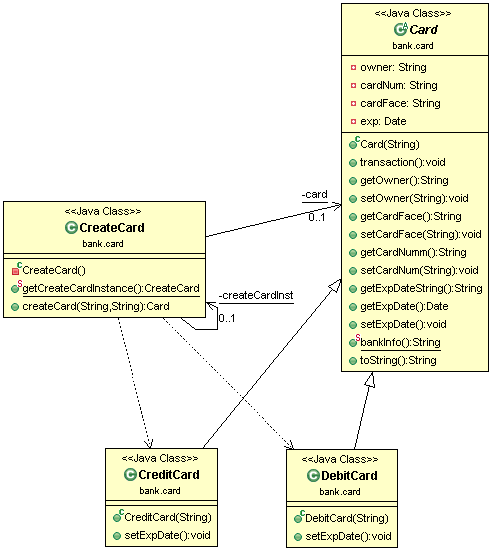
We have used the state pattern to make the bank account have different states when the client is trying to withdraw money. Depending on the amount of money in the account the bank account will have a different action. So in our assignment the bank account is the context class, the context objects come from the state classes such as NoTransactionFee class, TransactionFee class, and OverDrawn class. These classes make the action of bank account change when the user goes to withdraw by changing the context of the bank account class class.The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in.

# Chain of Responsibility



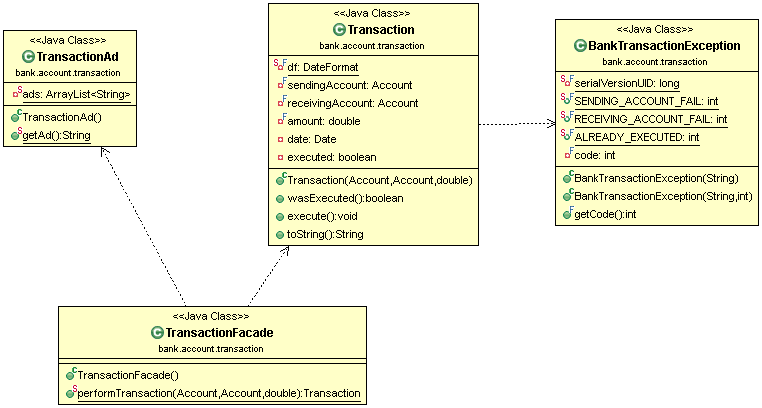
The chain of Responsibility (COR) pattern was implemented inside of our LoanRequest class. The idea is that when a Client requests a loan from our banking system, they will need to get approval from the Manager or the President, depending on how much the Loan Request is for. We first create two listeners, one for the Manager, and one for the President. If the Manager does not have the required authorization for a certain loan, than the COR pattern hands the loan off from the Manager listener, to the President listener. This pattern is useful because it allows us to not worry about what exact listener we have to hand it off to, it instead starts at the bottom of the chain, and works it way up to the listener that can handle the request.

# Flyweight Pattern



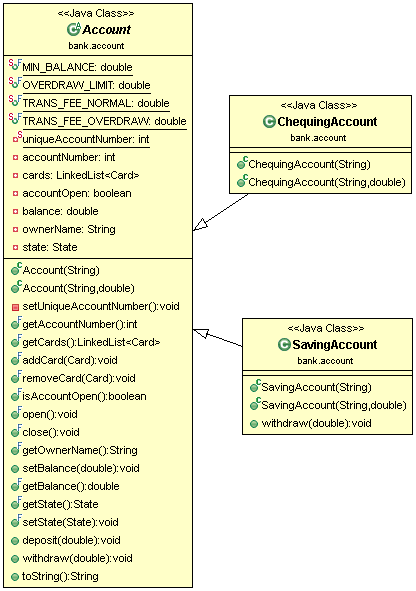
The Flyweight pattern we implemented goes along with the Cards class. A Flyweight pattern allows us to save memory by storing any values or parameters that might be the same throughout any type of Card. Some examples may include the Account name, Bank name, bank address, expiration date, etc. Having two subclasses of cards (one for Debit, and one for Credit), these both inherit any of the values that the main Card class may have. It can also inherit it’s methods to retrieve the information.

# Facade Pattern



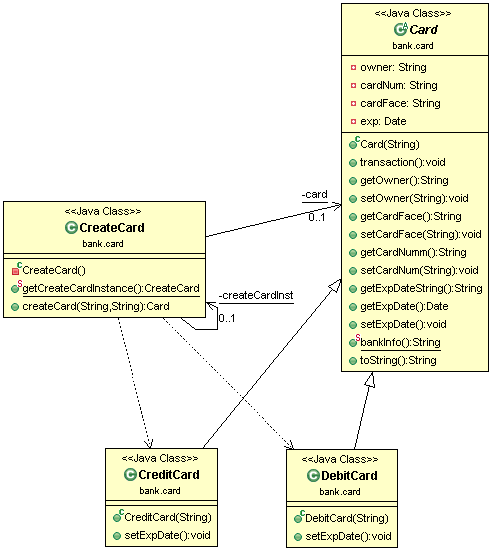
We implemented the Facade pattern into our Transaction class. Because a Facade pattern hides and may use multiple sub system parts, we implemented it into our Transaction class because some Transactions may require other methods to be run. For instance, all transactions are followed by an Advertisement, and some transactions may include service charges and fees. Because these are separate from the Transaction itself, the facade makes it easier for us to run any methods that may not be apart of the transaction, and keep the client from knowing how or where these methods are being run from.

# Template Pattern



Using the Template pattern for the Account class, we can make sure that any parameters that need to be used for any type of account are not duplicated in multiple subclasses. Minimum balances and Fees can apply to any account, so we make sure not to duplicate these fields for each individual account type. Withdrawing money with a SavingAccount costs $2.

# Singleton pattern



The Singleton pattern is implemented into our CreateCard class. A singleton makes sure we only have one instance of an object. Because the CreateCard class can create any type of card, we only need one object of it. If we first run the program, and the CreateCard is no instantiated, it creates an instance for us. From then on, if we need to create new cards, it will continue to use this instance. We can create an unlimited amount of cards through this CreateCard Singleton.