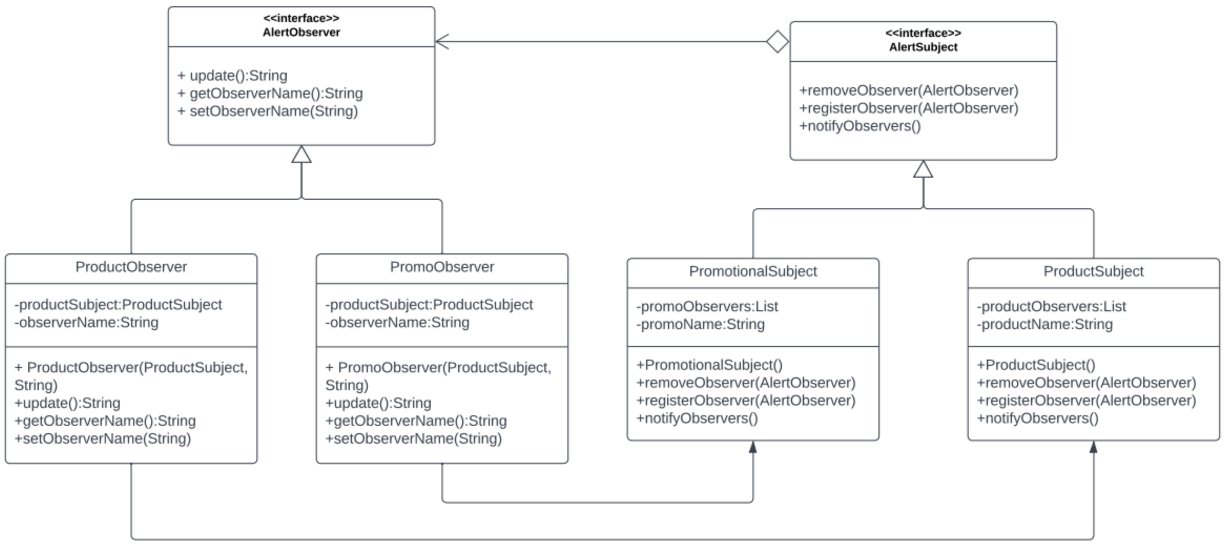
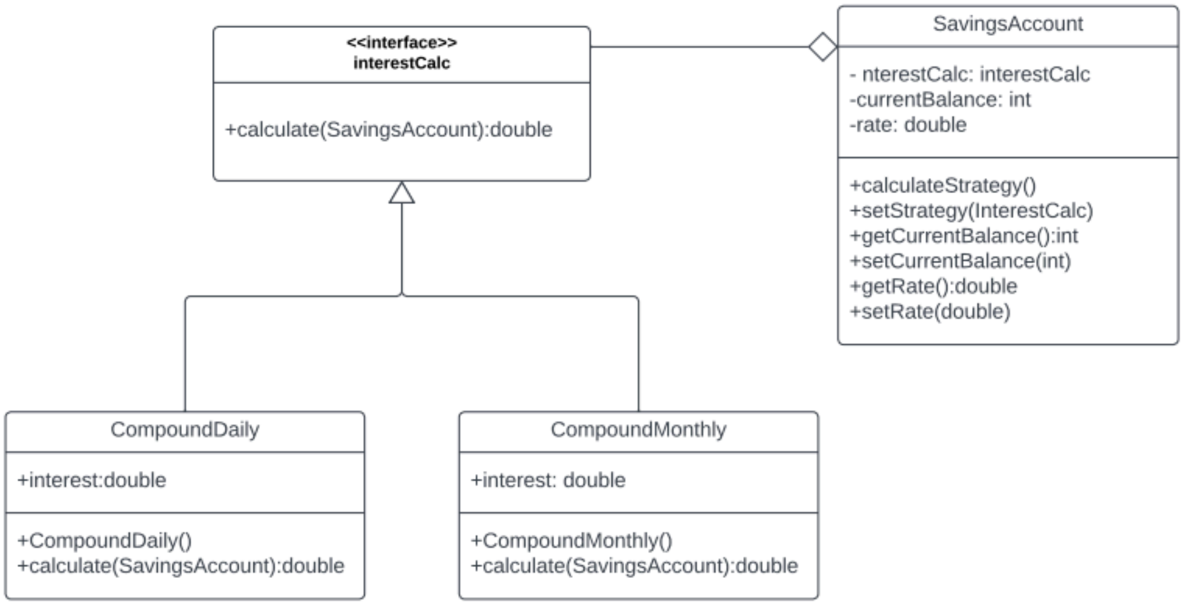
Marisa Paone CS665 HW5

1) Since each bank wants to provide alerts such as new products, changes in interest rates to its customers - the observer design pattern was selected. The observer pattern specifies communications between objects, the subjects and the observers. If a bank wants to send a notice to all of their promotional subscribed customers, the banks promotion would be the subject and the promotional subscribed customers are the observers. With this pattern, observers can be added or removed at any time. This particular functionality of the observer pattern is exactly what we need for customers, because we want them to be able to "unsubscribe" from promotional emails if they chose to. The observer pattern is used when we have a one to many relationship between objects, and if an objects is modified, its dependent objects are to be notified.



<<interface>> AlertObserver
+ update():String
+ getObserverName():String
+ setObserverName(String)
<<interface>> AlertSubject
+removeObserver(AlertObserver)
+registerObserver(AlertObserver)
+notifyObservers()
ProductObserver
-productSubject:ProductSubject
-observerName:String
+ ProductObserver(ProductSubject, String)
+update():String
+getObserverName():String
+setObserverName(String)
PromoObserver
-productSubject:ProductSubject
-observerName:String
+ PromoObserver(ProductSubject, String)
+update():String
+getObserverName():String
+setObserverName(String)
PromotionalSubject
-promoObservers:List
-promoName:String
+PromotionalSubject()
+removeObserver(AlertObserver)
+registerObserver(AlertObserver)
+notifyObservers()
ProductSubject
-productObservers:List
-productName:String
+ProductSubject()
+removeObserver(AlertObserver)
+registerObserver(AlertObserver)
+notifyObservers()

2) To represent different ways to calculate the interest amount for the savings account, the strategy pattern is best choice. This pattern is used when we have multiple algorithms for a specific task (different ways to calculate interest amount, compounded daily or compounded at the end of the month). We also want the client to be able to choose which way to calculate it, at runtime. In the strategy pattern we are creating objects which represent various strategies and a context object whose behavior varies depending on its strategy object. The strategy object in our case is the Savings Account, and the algorithms or the different strategies are the classes CompoundDaily and CompoundMonthly. The interestCalc interface is the strategy interface.



<<interface>>
interestCalc
+calculate(SavingsAccount):double
SavingsAccount
- nterestCalc: interestCalc
-currentBalance: int
-rate: double
+calculateStrategy()
+setStrategy(InterestCalc)
+getCurrentBalance():int
+setCurrentBalance(int)
+getRate():double
+setRate(double)
CompoundDaily
+interest:double
+CompoundDaily()
+calculate(SavingsAccount):double
CompoundMonthly
+interest: double
+CompoundMonthly()
+calculate(SavingsAccount):double