**Oregon Institute of Technology**

State Lab Winter ‘16  
Due Tuesday March 8th

**Goals:**

To develop and use the State pattern.

**Overview:**

In this lab you will refactor the StateLabBankAccountStarterCode—BankAccount class (to a StateBankAccount class) that uses the state pattern instead of enumerations. At this point, you should realize that the BankAccount class is in desperate need of some refactoring.

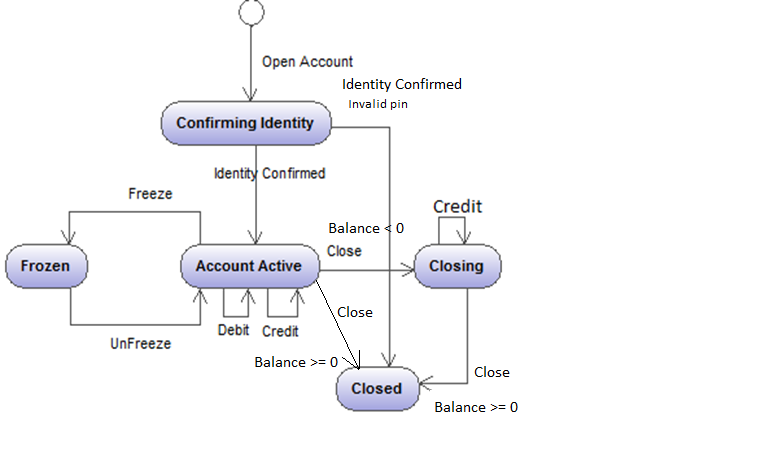
Open the StateLabBankAccountStarterCode.zip file and familiarize yourself with the code.

Examine the BankAccount class. This class implements the state machine (next page). While the code works, it’s obviously over complicated and difficult to extend. There are a lot of if/else blocks that make it difficult to see what operations are valid in a given state. Adding additional states will require modification to the existing logic for all other states.

Run all of the test methods in Main() (one at a time) and make sure you understand them.

The lab is written to implement a new StateBankAccount class and *then* all of the states. However, you may want to implement one StateBankAccount method with its corresponding state—and test as you go.

The logic for AccountActive Close() is as follows:   
  
--If the balance is equal to zero, the account closes and pops a msg box that notifies the balance is zero and the account has been closed.  
  
--If the balance is >0, the account closes and pops a msg box that a check has been cut for the outstanding balance and the account is closed.  
  
--If the balance is < 0, a msg box is displayed notifying that an overdrawn account cannot be closed and transitions to the Closing state where credits can be made until the balance is >= 0. Once the balance is >= 0, calling Close() in the Closing state functions the same as calling Close from the AccountActive state.



***Note: For the following steps, create a new C# console app. The name of the console app should be your last name. Since we have to refactor so much of the original code—it is easier to start fresh with a brand new console app. Use the starter code as a reference for your new code base.***

**Steps:**

1. To the new console app (with your last name) add a class library named “***StateLabBankAccoun***t.” Perform the following steps in your ***StateLabBankAccoun***t dll.
2. Add a file named “StateBankAccount.” Create a partial StateBankAccount class that will replace the BankAccount class of the starter code. It should have the same public methods as the BankAccount class.
3. Also, in this class declare private data members of type AbstractAccountState for each of the states you will implement later. Name the variables very similar to names on the state diagram. For example: AbstractAccountState confirmingIdentityState. Note: the code will not compile at this time since we have not implemented AbstractAccountState or any of the other state types.
4. Lastly, add a private AbstractAccountState variable named “state” that will track the current state and a private variable of type double named “balance.”
5. Implement a StateBankAccount c’tor that instantiates each of the state diagram variables declared in step 2 passing in a reference to the StateBankAccount as a c’tor parameter. Set the “state’ variable equal to the ConfirmingIdentify state and initialize “balance” to 0;
6. Rework all the public methods of the new StateBankAccount class so they delegate calls to the current state passing the appropriate parameters when necessary. For example: state.IndentityConfirmed(pin);
7. Add any necessary private internal methods (called by the concrete states) to handle the “accounting” logic of StateBankAccount class. These methods implement the console and message box outputs.
8. Add an abstract base state class that implements all the public methods of the new StateBankAccount class (call it AbstractAccountState) and throws an exception if not implemented by the concrete state classes. These methods need to be pure virtual so they can be overriden. This simplifies the error logic in the concrete state classes.
9. The AbstractAccountState base class needs to have a protected field to hold a reference to the StateBankAccount instance—the constructor initializes this field.
10. Add a new file—name it BankAccountStates. Rename the default class to a partial StateBankAccount class (this will insert the state classes inside the StateBankAccount class).
11. Inside the partial StateBankAccount class, create classes to represent each state of the old BankAccount enum and derive them from the AbstractAccountStates abstract class. Note: This is important! Your state class methods need to override the AbstractAccountState methods as appropriate and should only implement state logic. In other words, there should not be any Console.WriteLine statements or message boxes and use “if” statements *only* when needed for state change logic. All “accounting” logic should be handled by calling the StateBankAccount private internal methodsthis.account = account;}.

B ddddjlkj;lj

* + 1. public virtual void IdentityConfirmed()
    2. {
    3. throw new InvalidOperationException("Account does not support IdentityConfirmation in current state");
    4. }
    5. public virtual void Credit(decimal amount)
    6. {
    7. throw new InvalidOperationException("Account does not support Credit in current state");
    8. }
    9. public virtual void Debit(decimal amount)
    10. {
    11. throw new InvalidOperationException("Account does not support Debit in current state");
    12. }
    13. public virtual void Freeze()
    14. {
    15. throw new InvalidOperationException("Account does not support Freeze in current state");
    16. }
    17. public virtual void UnFreeze()
    18. {
    19. throw new InvalidOperationException("Account does not support UnFreeze in current state");
    20. }
    21. public virtual void Close()
    22. {
    23. throw new InvalidOperationException("Account does not support Close in current state");
    24. }
  1. }

1. changeCndkdkdkdkdkd Add the code to Main() to implement the test methods shown in the starter code. Your output should be identical (with additional message boxes) to what was produced by the original “messy” code.