State Pattern

Estimated time for completion: 60 minutes

Overview:

This lab comprises of two exercises the first will be to implement a very simple finite state machine using the state pattern, the second part will be taking a more complex example where code has already been written not using the state pattern but simple if/else your task here will be to refactor the code to use the state pattern.

Goals:

- Develop and put to use the State pattern

Lab Notes:

Part 1: Create a simple Finite State Machine

For this part you will create a simple type and make use of a finite state machine that simply has two states: Editing and Viewing. The client will only be able to change the state of the object when the object is in its edit state.

1. Load the EditablePerson Solution, familiarize yourself with the code and run it. As you can see there is an Edit and a Commit method that contain no implementation. These methods are intended to be used by the client when it wishes to change the state of the Person. By first calling Edit this will allow the client to perform an update, for example changing the Name or Age, when they are done the client calls commit and the object returns back to a readonly/view state. Currently the code does not enforce this rule. Your job is to make it do this. One way would be to simply have a Boolean that defines if you are in edit mode or not, whilst this would work you are going to use the state pattern.
2. The first step in making this code use the state pattern is to define an abstract class to represent the various states an object can be in, and all the possible state based operations. These operations include the Name and Age Property accessors and the Edit and Commit methods. (You will not worry about ToString() as it has the same behavior in all states). You want this state base class to be only visible to the Person type. To do this you will make it an inner class of the Person type. You should define the inner class in a separate file. To do that make the Person class a partial class, then, in a separate file, define the abstract state class wrapping it in a partial Person class definition.
3. This makes the code more maintainable by having each (inner) class defined in its own file.
4. The code should look like this:

```
public partial class Person
{
   private abstract class PersonState
   {
```

```
        protected Person innerPerson;

        protected PersonState(Person person)
        {
            innerPerson = person;
        }

        public virtual string Name
        {
            get {throw new InvalidOperationException("Get Name not supported
   at this time");}
            set {throw new InvalidOperationException("Set Name not supported
   at this time");}
        }

        public virtual int Age
        {
            get { throw new InvalidOperationException("Get Age is not
   supported at this time"); }
            set { throw new InvalidOperationException("Set Age is not
   supported at this time"); }
        }

        public virtual void Edit()
        {
            throw new InvalidOperationException("Edit  is not supported at
   this time");
        }

        public virtual void Commit()
        {
            throw new InvalidOperationException("Commit is not supported at
   this time");
        }
    }
}
```

5. Notice that you have added a constructor that takes a reference to a Person, whilst inner classes have access to the private parts of their parent they do need to be given a reference to their parent.
6. Now let's refactor the Person class to delegate all requests to a state as opposed to implementing it directly inside the class. To do this, add a new private field to the Person class to hold the current state: private PersonState currentState=null. Change each method/property accessor to now call the equivalent method on the currentState reference

```
partial class Person
{
    private PersonState currentState;

    public Person()
    {
        currentState = null;
```



```
    }

    private string name;

    public string Name
    {
        get { return currentState.Name; }
        set { currentState.Name = value; }
    }

    private int age;

    public int Age
    {
        get { return currentState.Age; }
        set { currentState.Age = value; }
    }

    public override string ToString()
    {
        return String.Format("{0}, Aged {1} years", name, age);
    }

    public void Edit()
    {
        currentState.Edit();
    }

    public void Commit()
    {
        currentState.Commit();
    }
}
```

7. Now create two additional inner classes that derive from PersonState one to represent EditState and one to represent ViewState. Don't implement any methods simply create the types. You will however have to add a constructor so that you can initialize the reference to the parent.

```
public partial class Person
{
   private class EditState : PersonState
   {
      public EditState(Person person) : base(person)    {   }
   }
}
```

8. Once you have created the two state classes Edit and View, you now need to modify the Person class again to hold references to instances of these states

```
partial class Person
{
   private PersonState currentState;
```



```
   private PersonState editState;
   private PersonState viewState;

   public Person()
   {
      editState = new EditState(this);
      viewState = new ViewState(this);
   }
...
```

9. Now add an additional method to the Person class called SetState, this will take a new PersonState as a parameter, and will change the current state of the Person object to a new state. This will be called by any operation that results in a state change.

```
private void SetState(PersonState newState)
{
   currentState = newState;
}
```

10. Modify the Person constructor to call SetState with the initial state, SetState(viewState);
11. Now implement the appropriate methods for the EditState and ViewState types. The EditState type should implement the Name and Age properties along with the Commit method, the ViewState type should implement get accessors for Name and Age properties and the Edit method. If you now re-run the project it should fail at run time as you are attempting to update the Person whilst in a view state, confirm this happens. Ensure you are getting the appropriate exception, if you have made a mistake with your implementation you may well get a StackOverflowException.
12. Now modify Main to call Edit and Commit around the calls that modify the person's Name and Age properties, and confirm that the code works.
13. There is a possible enhancement that could be made to this solution. The fact that view and edit state share some common implementation suggests that you could define a class that implements that common functionality and have both states then derive from that as opposed from directly deriving from the abstract class.

Part 2: State Bank

In this part you will take a piece of code that implements a state machine that is using enumerations and lots of if statements and refactor it to use the state pattern. Hopefully when you look at the code you too will feel it is rather smelly and in desperate need of some refactoring to make it clear what operations are possible in what states.

1. Open the solution and familiarize yourself with the project. Run the project, it should open a form displaying information about a bank account. If you hit the show workflow button a window will appear describing the possible states the bank account can be in, and what operations are valid in a given state, have a play with the UI validating that the operations you are allowed to do match the state machine in the picture.
2. Examine the BankAccount class. This class implements the state machine. Whilst the code does what it's supposed to do, when you look at it you should see how complicated

it is. There are lots of if/else blocks making it hard to see what operations are valid in a given state. Adding additional states will require modification to the existing logic for all other states. You would rather have this code rewritten to take advantage of the state pattern, thus allowing you to clearly see what is possible in a given state and also allow you to add new states in the future without the risk of breaking existing code.

3. Your first step in refactoring this code is to create the abstract class to represent each state, remembering to:
    a. Make the base state class an abstract inner class of BankAccount in a separate file, taking advantage of partial class functionality
    b. Make the class have a protected field to hold a reference to the BankAccount instance, and create the necessary constructor required to initialize this field
    c. Add a virtual method for every possible operation, where the implementation of each method simply throws an InvalidOperationException

```
public abstract class BankAccountState
{
   protected BankAccount account;

   protected BankAccountState(BankAccount account)
   {
      this.account = account;
   }

   public virtual void IdentityConfirmed()
   {
      throw new InvalidOperationException("Account does not support
   IdentityConfirmation in current state");
   }

   public virtual void Credit(decimal amount)
   {
      throw new InvalidOperationException("Account does not support Credit in
   current state");
   }

   public virtual void Debit(decimal amount)
   {
      throw new InvalidOperationException("Account does not support Debit in
   current state");
   }

   public virtual void Freeze()
   {
      throw new InvalidOperationException("Account does not support Freeze in
   current state");
   }

   public virtual void UnFreeze()
   {
      throw new InvalidOperationException("Account does not support UnFreeze
   in current state");
   }
```

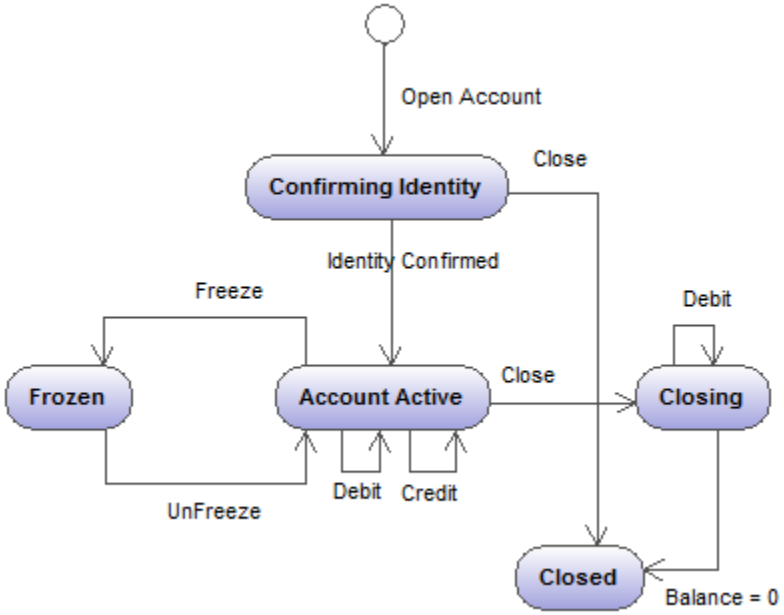


```
   public virtual void Close()
   {
      throw new InvalidOperationException("Account does not support Close in
   current state");
   }
}
```

4. Remove the use of the AccountStates enumeration to track current state and use a reference to your new state abstract class instead. Rework all the methods in the BankAccount class to use this reference to implement the required functionality. (Tip, Comment out the existing functionality you will need it in some form or another later.)
5. Modify the State property on the BankAccount class to return the current state in terms of the abstract State class. Make the get public, but make the set private. This will allow the inner state classes the ability to change state but no one else.

```
public BankAccountState State
{
   get { return accountState; }
   private set
   {
      accountState = value;
      OnPropertyChanged(new PropertyChangedEventArgs("State"));
   }
}
```

6. Now create classes to represent each of the states for a bank account, remembering to derive from the state abstract class. Take the commented out code and place it into the appropriate methods, or encapsulate the functionality inside the BankAccount class as private methods that you invoke from the state classes.



7. Modify the BankAccount class to initialize all the states on its creation, and set the initial state to ConfirmingIdentity. Now run the application assuming everything has been done correctly you should pretty much have a working application. Test it out.
8. The only place where things may not quiet go well is if you perform the following steps.
9. Run the application. Click Confirmed Identity, and then hit Close Account. This will take you to the closing state, but really it should go straight to closed since the balance is zero.
10. One way to solve this problem would be to change the logic in every state which moves you into the closing state to check for a balance of zero and if so go straight to closed, but that breaks the finite state machine and means repeating the logic.
11. Alternatively you can somehow detect the balance is zero on entering the Closing state and if so move to Closed state. This is a common requirement in finite state to have some code run when you enter or exit a state. So add two additional virtual methods to the base State class called Enter and Exit

```
public abstract class BankAccountState
{
   public virtual void EnterState()
   {
      // Called on entry to the state
   }

   public virtual void LeaveState()
   {
      // Called on leaving the state
   }
   ...
}
```

12. The code above provides the appropriate hooks that each state can override in order to be informed when their given state is entered or exited. All you need to do now is invoke the methods when the state changes, by modifying the setter for the State property on the BankAccount class.

```
public BankAccountState State
{
   get { return accountState; }
   private set
   {
      if (accountState != null)
      {
      accountState.LeaveState();
      }

      accountState = value;
      OnPropertyChanged(new PropertyChangedEventArgs("State"));
      accountState.EnterState();
   }
}
```


13. Now implement the EnterState method inside your closing state to detect if the Balance is zero and if so move straight to the closed state. Re-run the test you did before and you should find it goes straight to Closed State.