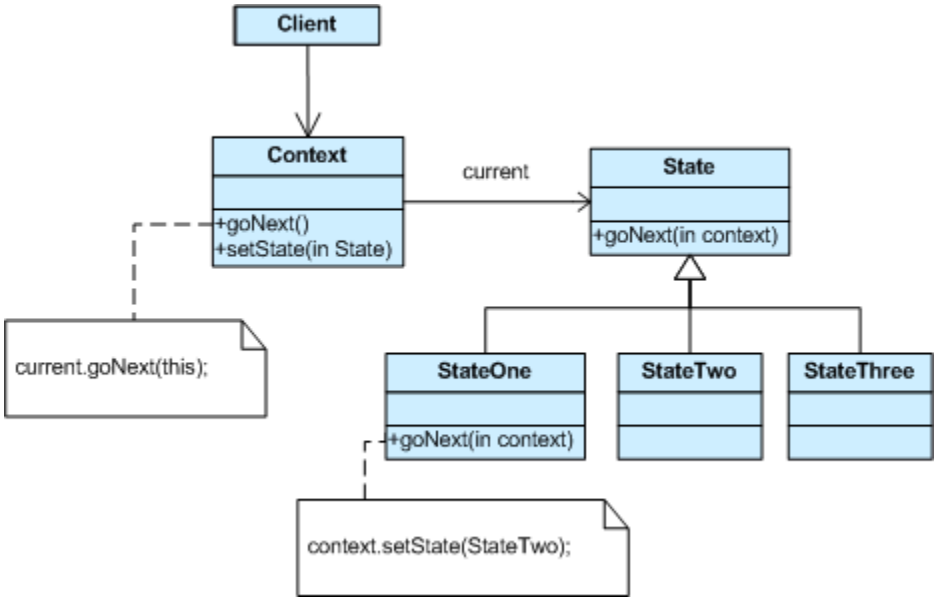
CPSC 501 – Assignment #9

State and Observer Patterns

State Pattern:

The purpose of state pattern is to model the state transitions in a finite state machine. Each state is represented by its own class which defines the actions or inputs that will cause the state to change to another state. There is a central context object (that every state object has a reference to) which indicates the current state and the behavior of the system. The state objects indicate change of state by setting the state to another state via the context object.
The UML for the state pattern appears as:



Create a new windows forms application project called StatePattern. Add an interface to the project called ICheckingActivity with the following code in it.

```
interface ICheckingActivity
{
    void WithDrawMoney(double amt); // may cause to change state
    void DepositMoney(double amt);  // may cause to change state
    void AddInterest();             // may cause to change state
}
```

Add the context state to the project called CheckingAccount. The context maintains the non state information such as customer name, account number, and maintains a state object for the current state of the system. It also initializes the starting state of the system.

```
// context
 class CheckingAccount : ICheckingActivity
 {
     public CheckingAccount(string cname, int acctNum)
     {
```

```
            customerName = cname;
            accountNum = acctNum;
            state = new SilverState(0, this);
        }

        // current state
        BaseState state;
        internal BaseState State
        {
            get { return state; }
            set { state = value; }
        }

        string customerName;
        public string CustomerName
        {
            get { return customerName; }
        }

        int accountNum;
        public int AccountNum
        {
            get { return accountNum; }
        }

        double balance;
        public double Balance
        {
            get { return balance; }
        }

        //----------ICheckingActivity methods---------
        public void WithDrawMoney(double amt)
        {
            state.WithDrawMoney(amt);
        }

        public void DepositMoney(double amt)
        {
            state.DepositMoney(amt);
        }

        public void AddInterest()
        {
            state.AddInterest();
        }
    }
```

Add a class called BaseState that is responsible for the state information of a checking account. It maintains the balance and has a reference to the context object (CheckingAccount) which actually maintains the current state of the system.

```
abstract class BaseState : ICheckingActivity
{
    // the decision to move to another state
    // is managed by the individual state itself
    // The context i.e., CheckAccount in this
    // example maintains the current state of the
    // system
    // context has a reference to the current state
    // and state has a reference to the context
    // so that it can update the current state

    CheckingAccount account;  // context
    public CheckingAccount Account
    {
        get { return account; }
        set { account = value; }
    }

    double balance;  // so that state change can be determined
    public double Balance
    {
        get { return balance; }
        set { balance = value; }
    }

    // ICheckingActivity members
    public abstract void WithDrawMoney(double amt);
    public abstract void DepositMoney(double amt);
    public abstract void AddInterest();

    public void CheckStateChange()
    {
        // decide what the next state will be
        if (balance < 0)
            account.State = new OverDrawState(this);
        if ((balance >= 0) && (balance < 1000))
            account.State = new SilverState(this);
        if ((balance >= 1000) && (balance < 5000))
            account.State = new GoldState(this);
        if (balance >= 5000)
            account.State = new PlatinumState(this);
    }
}
```

Add a concrete state class called OverDrawState which inherits from the BaseState as shown below. It charges a fee of $20 for every withdrawl. You cannot withdraw money if the balance drops below $100.

```
class OverDrawState : BaseState
{
    double overdrawFee = 0;
    double overdrawLimit = -100;
```

```
    public OverDrawState(BaseState state)
    {
        overdrawFee = 20;
        this.Account = state.Account;
        this.Balance = state.Balance;
    }

    public override void WithDrawMoney(double amt)
    {
        Balance = Balance - overdrawFee;
        if ((Balance - amt) < overdrawLimit)
            Balance = Balance - amt;
        else
            MessageBox.Show("cannot withdraw, limit reached");
    }

    public override void DepositMoney(double amt)
    {
        Balance = Balance + amt;
        CheckStateChange();
    }

    public override void AddInterest()
    {
        // no interest in overdraw state
    }

    public override string ToString()
    {
        return "balance = " + Balance.ToString() + "\n" +
            "State = OverDrawState";
    }
}
```

Add another concrete state class called SilverState with the following code in it. This class charges a fee of one dollar for each withdrawl and requires balance to be between $0 and $1000.

```
class SilverState : BaseState
{
    public SilverState(double bal, CheckingAccount acct)
    {
        this.Balance = bal;
        this.Account = acct;
    }

    public SilverState(BaseState state)
    {
        this.Account = state.Account;
        this.Balance = state.Balance;
    }
```

```
    public override void WithDrawMoney(double amt)
    {
        Balance = Balance – amt -1 ; // $1 transaction fee for Silver
        CheckStateChange();
    }

    public override void DepositMoney(double amt)
    {
        Balance = Balance + amt;
        CheckStateChange();
    }

    public override void AddInterest()
    {
        // no interest in silver state
    }

    public override string ToString()
    {
        return "balance = " + Balance.ToString() + "\n" +
            "State = SilverState";
    }
}
```

Add another concrete state class called GoldState with the following code in it. This state pays an interest of 3% and does not charge any transaction fee.

```
class GoldState : BaseState
{
    public GoldState(BaseState state)
    {
        this.Account = state.Account;
        this.Balance = state.Balance;
    }

    public override void WithDrawMoney(double amt)
    {
        Balance = Balance - amt;
        CheckStateChange();
    }

    public override void DepositMoney(double amt)
    {
        Balance = Balance + amt;
        CheckStateChange();
    }

    public override void AddInterest()
    {
        Balance = Balance + 0.03 * Balance; // 3% interest
        CheckStateChange();
    }
```

```
        public override string ToString()
        {
            return "balance = " + Balance.ToString() + "\n" +
                "State = GoldState";
        }
    }
```

Add a concrete state class called PlatinumState that provides a 4% interest.

```
    class PlatinumState : BaseState
    {
        public PlatinumState(BaseState state)
        {
            this.Account = state.Account;
            this.Balance = state.Balance;
        }

        public override void WithDrawMoney(double amt)
        {
            Balance = Balance - amt;
            CheckStateChange();
        }

        public override void DepositMoney(double amt)
        {
            Balance = Balance + amt;
            CheckStateChange();
        }

        public override void AddInterest()
        {
            Balance = Balance + 0.04 * Balance;
            CheckStateChange();
        }
        public override string ToString()
        {
            return "balance = " + Balance.ToString() + "\n" +
                "State = PlatinumState";
        }
    }
```

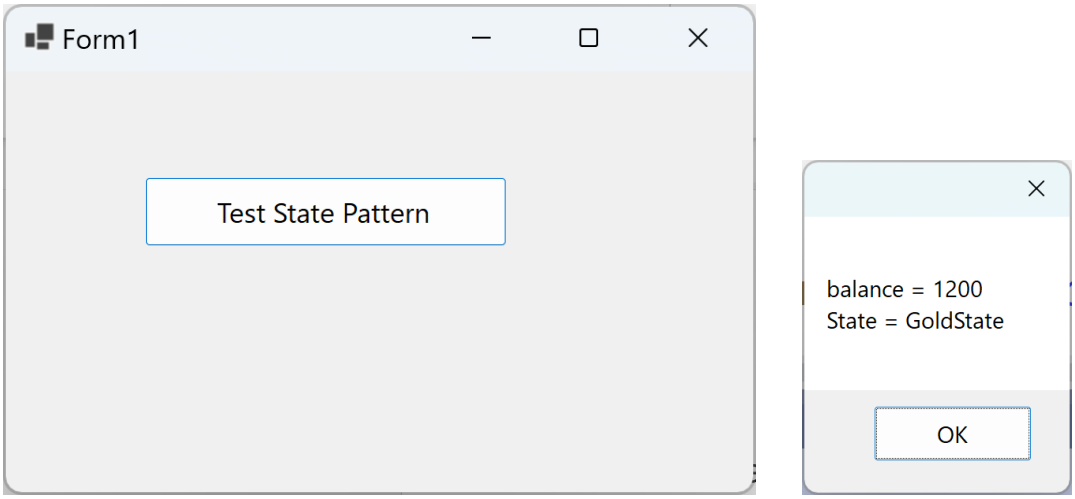
Note that each state decides what the next state should be based on the activity. The current state is always maintained by the context (CheckingAccount), so the ode in CheckStateChange (in BaseState) sets the state in the context object (account in above code).

To test the state pattern, add a button to the form called btnTestStatePattern with the following code in it.

```
        private void btnTestStatePattern_Click(object sender, EventArgs e)
        {
            // client interacts with the context i.e., CheckingAccount
            CheckingAccount acct = new CheckingAccount("Bill",
```

```
            1234);
        acct.DepositMoney(1200);
        //acct.WithDrawMoney(500);
        //acct.DepositMoney(6000);
        MessageBox.Show(acct.State.ToString());
    }
```

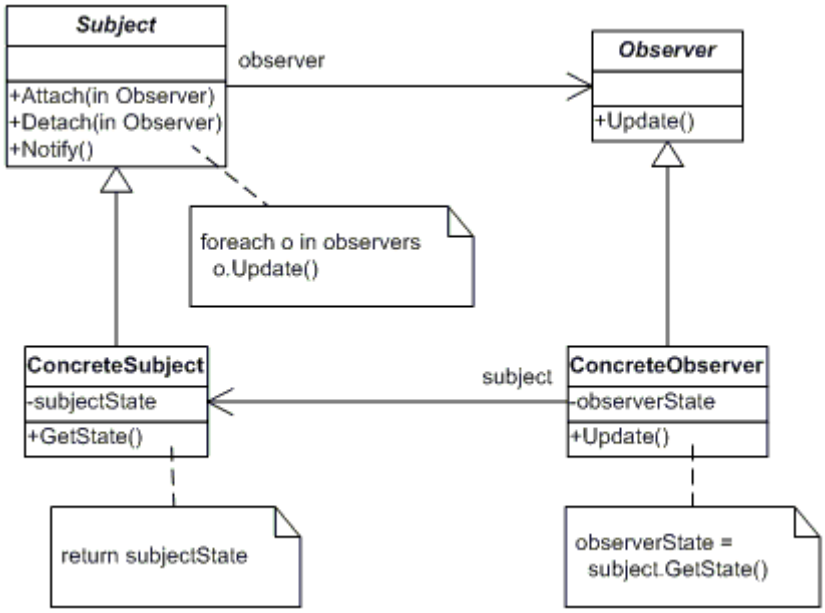
You can uncomment the lines above to see how the account changes between different states.



Observer Pattern

The observer pattern provides a clean solution to one-to-many dependencies between objects, so that if the state of one object (subject) changes, all its dependents are notified and updated automatically. Observer pattern is useful in publisher subscriber type of situations, or in situations where if the state of a subject changes (e.g., stock price), the clients can be updated.

The UML for the observer pattern is shown below.



Example: Create a windows forms application project called ObserverPattern. Add an interface called Iobserver with the following code in it.

```
interface IObserver  // observer is the client i.e., subscriber
{
    void Notify(StockInfo sinfo);
}
```

Add a class called StockInfo with the following code in it.

```
class StockInfo
{
    public StockInfo(string sym, double pr)
    {
        symbol = sym; price = pr;
    }

    string symbol;  // subject info
    public string Symbol
    {
        get { return symbol; }
        set { symbol = value; }
    }

    double price;  // subject info
    public double Price
    {
```

```
            get { return price; }
            set { price = value; }
        }

        public override string ToString()
        {
            return symbol + " : " + price.ToString();
        }
    }
```

Add a class called StockSubject with the following code in it.

```
    class StockSubject  // base subject class
    {
        List<IObserver> OBList = new List<IObserver>();
        StockInfo stockInfo = null;
        public StockSubject(StockInfo sinfo)
        {
            stockInfo = sinfo;
        }

        public void UpdatePrice(double updateAmt)
        {
            stockInfo.Price = stockInfo.Price + updateAmt;
            // notify observers i.e., subscribers
            foreach (IObserver observer in OBList)
                observer.Notify(stockInfo);  // send state of stock
        }

        public void AddObserver(IObserver obs)
        {
            OBList.Add(obs);
        }

        public void RemoveObserver(IObserver obs)
        {
            OBList.Remove(obs);
        }
    }
```

Add a class called ConcreteSubjectGoog with the following code in it.

```
    class ConcreteStockSubjectGoogle : StockSubject
    {
        public ConcreteStockSubjectGoogle() :
            base(new StockInfo("Goog", 585))
        {
        }
    }
```

Add a class called ConcreteObserver with the following code in it.

```
    class ConcreteObserver : IObserver
    {
        public ConcreteObserver(string nm)
        {
            clientName = nm;
        }

        string clientName;
```

```
        public string ClientName
        {
            get { return clientName; }
            set { clientName = value; }
        }

        public void Notify(StockInfo sinfo)
        {
            MessageBox.Show("Notification sent to : " + clientName + "\n" +
                sinfo.ToString());
        }
    }
```

To test the observer pattern, add a button to the form with a name of btnObserver with the following test code in it.

```
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnObserver_Click(object sender, EventArgs e)
        {
            ConcreteStockSubjectGoogle csgoog = new
ConcreteStockSubjectGoogle();
            ConcreteObserver co1 = new ConcreteObserver("Bill");
            csgoog.AddObserver(co1);
            ConcreteObserver co2 = new ConcreteObserver("Sally");
            csgoog.AddObserver(co2);
            csgoog.UpdatePrice(7.50);
        }
    }
```

If you run the program and click on the above button, you will see the following output.