HUMAYUN AHMED

PRINCIPAL SOFTWARE ENGINEER

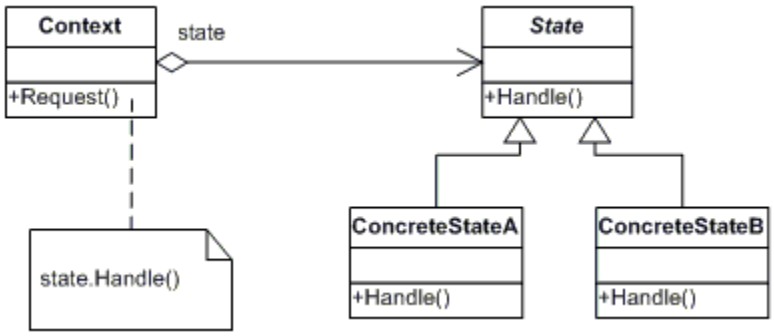
CEFALO

State Pattern

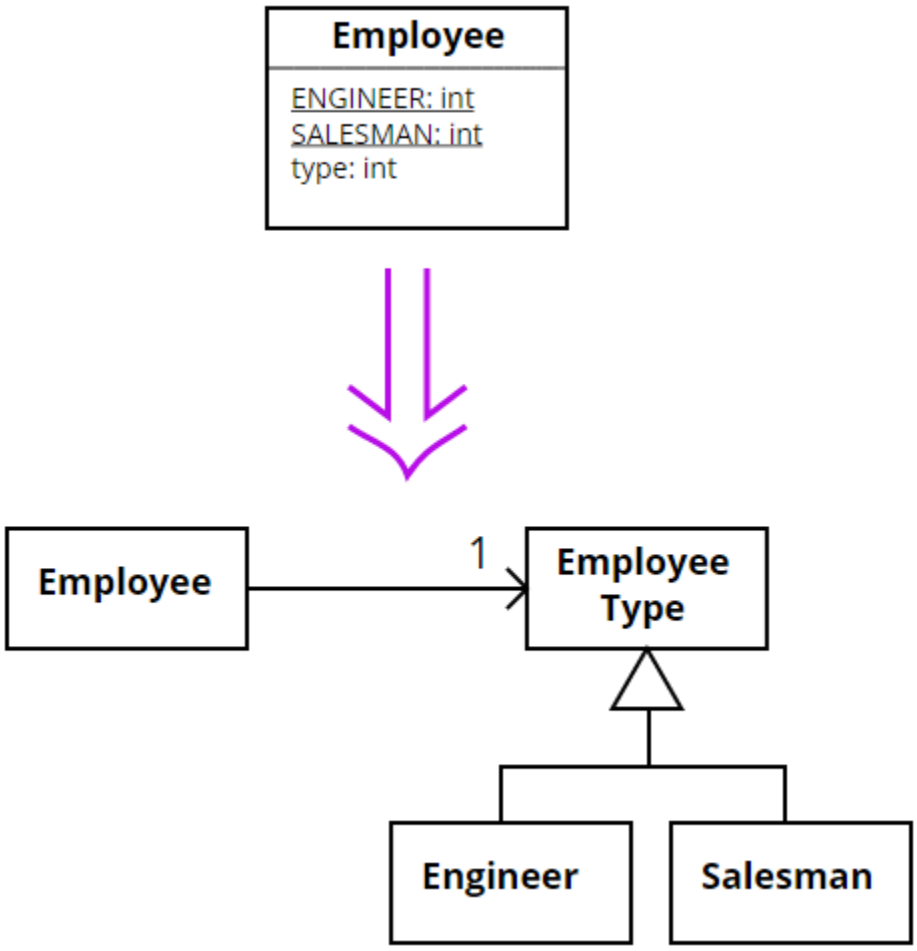
Definition

Allow an object to alter its behavior when it's internal state changes. The object will appear to change its class.

Diagram



Replace Type Code with State/Strategy



Structural code

```
/// <summary>
 /// MainApp startup class for Structural
 /// State Design Pattern.
 /// </summary>
 class MainApp
 {
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
```

```
    // Setup context in a state
    Context c = new Context(new ConcreteStateA());

    // Issue requests, which toggles state
    c.Request();
    c.Request();
    c.Request();
    c.Request();

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'State' abstract class
/// </summary>
abstract class State
{
  public abstract void Handle(Context context);
}

/// <summary>
/// A 'ConcreteState' class
/// </summary>
class ConcreteStateA : State
{
  public override void Handle(Context context)
  {
    context.State = new ConcreteStateB();
  }
}

/// <summary>
/// A 'ConcreteState' class
/// </summary>
class ConcreteStateB : State
{
  public override void Handle(Context context)
  {
    context.State = new ConcreteStateA();
  }
}

/// <summary>
/// The 'Context' class
/// </summary>
class Context
```

```
{
  private State _state;

  // Constructor
  public Context(State state)
  {
    this.State = state;
  }

  // Gets or sets the state
  public State State
  {
    get { return _state; }
    set
    {
      _state = value;
      Console.WriteLine("State: " +
        _state.GetType().Name);
    }
  }

  public void Request()
  {
    _state.Handle(this);
  }
}
```

Real-world code

```
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// State Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Open a new account
      Account account = new Account("Jim Johnson");
```

```
15.
16.     // Apply financial transactions
17.     account.Deposit(500.0);
18.     account.Deposit(300.0);
19.     account.Deposit(550.0);
20.     account.PayInterest();
21.     account.Withdraw(2000.00);
22.     account.Withdraw(1100.00);
23.
24.     // Wait for user
25.     Console.ReadKey();
26.   }
27. }
28.
29. /// <summary>
30. /// The 'State' abstract class
31. /// </summary>
32. abstract class State
33. {
34.   protected Account account;
35.   protected double balance;
36.
37.   protected double interest;
38.   protected double lowerLimit;
39.   protected double upperLimit;
40.
41.   // Properties
42.   public Account Account
43.   {
44.     get { return account; }
45.     set { account = value; }
46.   }
47.
48.   public double Balance
49.   {
50.     get { return balance; }
51.     set { balance = value; }
52.   }
53.
54.   public abstract void Deposit(double amount);
55.   public abstract void Withdraw(double amount);
56.   public abstract void PayInterest();
57. }
58.
59.
60. /// <summary>
61. /// A 'ConcreteState' class
62. /// <remarks>
```

```
/// Red indicates that account is overdrawn
/// </remarks>
/// </summary>
class RedState : State
{
  private double _serviceFee;

  // Constructor
  public RedState(State state)
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }

  private void Initialize()
  {
    // Should come from a datasource
    interest = 0.0;
    lowerLimit = -100.0;
    upperLimit = 0.0;
    _serviceFee = 15.00;
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    amount = amount - _serviceFee;
    Console.WriteLine("No funds available for withdrawal!");
  }

  public override void PayInterest()
  {
    // No interest is paid
  }

  private void StateChangeCheck()
  {
    if (balance > upperLimit)
    {
      account.State = new SilverState(this);
    }
  }
```

```
}

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Silver indicates a non-interest bearing state
/// </remarks>
/// </summary>
class SilverState : State
{
  // Overloaded constructors

  public SilverState(State state) :
    this(state.Balance, state.Account)
  {
  }

  public SilverState(double balance, Account account)
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  private void Initialize()
  {
    // Should come from a datasource
    interest = 0.0;
    lowerLimit = 0.0;
    upperLimit = 1000.0;
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    balance -= amount;
    StateChangeCheck();
  }

  public override void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
```

```
  }

  private void StateChangeCheck()
  {
    if (balance < lowerLimit)
    {
      account.State = new RedState(this);
    }
    else if (balance > upperLimit)
    {
      account.State = new GoldState(this);
    }
  }
}

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Gold indicates an interest bearing state
/// </remarks>
/// </summary>
class GoldState : State
{
  // Overloaded constructors
  public GoldState(State state)
    : this(state.Balance, state.Account)
  {
  }

  public GoldState(double balance, Account account)
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  private void Initialize()
  {
    // Should come from a database
    interest = 0.05;
    lowerLimit = 1000.0;
    upperLimit = 10000000.0;
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
```

```
  }

  public override void Withdraw(double amount)
  {
    balance -= amount;
    StateChangeCheck();
  }

  public override void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  private void StateChangeCheck()
  {
    if (balance < 0.0)
    {
      account.State = new RedState(this);
    }
    else if (balance < lowerLimit)
    {
      account.State = new SilverState(this);
    }
  }
}

/// <summary>
/// The 'Context' class
/// </summary>
class Account
{
  private State _state;
  private string _owner;

  // Constructor
  public Account(string owner)
  {
    // New accounts are 'Silver' by default
    this._owner = owner;
    this._state = new SilverState(0.0, this);
  }

  // Properties
  public double Balance
  {
    get { return _state.Balance; }
  }
```

```
255.
256.   public State State
257.   {
258.     get { return _state; }
259.     set { _state = value; }
260.   }
261.
262.   public void Deposit(double amount)
263.   {
264.     _state.Deposit(amount);
265.     Console.WriteLine("Deposited {0:C} --- ", amount);
266.     Console.WriteLine(" Balance = {0:C}", this.Balance);
267.     Console.WriteLine(" Status = {0}",
268.       this.State.GetType().Name);
269.     Console.WriteLine("");
270.   }
271.
272.   public void Withdraw(double amount)
273.   {
274.     _state.Withdraw(amount);
275.     Console.WriteLine("Withdrew {0:C} --- ", amount);
276.     Console.WriteLine(" Balance = {0:C}", this.Balance);
277.     Console.WriteLine(" Status = {0}\n",
278.       this.State.GetType().Name);
279.   }
280.
281.   public void PayInterest()
282.   {
283.     _state.PayInterest();
284.     Console.WriteLine("Interest Paid --- ");
285.     Console.WriteLine(" Balance = {0:C}", this.Balance);
286.     Console.WriteLine(" Status = {0}\n",
287.       this.State.GetType().Name);
288.   }
289. }
290.}
```

Home Work

Build a wizard which has many steps and sometime based on the input steps varies. At every step user should know either move forward and back.

Use State pattern to identify each step as a State and move along them.