HUMAYUN AHMED

PRINCIPAL SOFTWARE ENGINEER

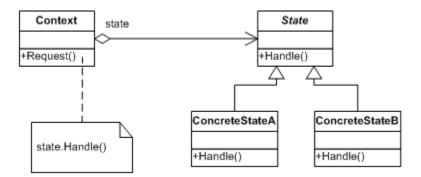
CEFALO

State Pattern

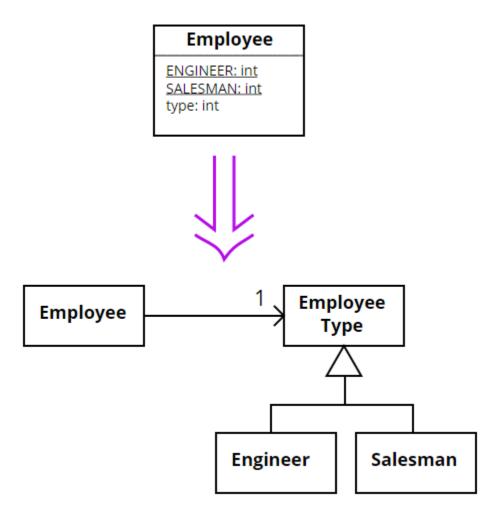
Definition

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

```
1. /// <summary>
2. /// MainApp startup class for Structural
3. /// State Design Pattern.
4. /// </summary>
5.
    class MainApp
6. {
     /// <summary>
7.
     /// Entry point into console application.
8.
     /// </summary>
9.
     static void Main()
11.
     {
```

```
12.
       // Setup context in a state
       Context c = new Context(new ConcreteStateA());
13.
14.
15.
       // Issue requests, which toggles state
       c.Request();
17.
       c.Request();
18.
       c.Request();
19.
       c.Request();
20.
21.
       // Wait for user
22.
       Console.ReadKey();
23.
     }
24. }
25.
26. /// <summary>
27. /// The 'State' abstract class
28. /// </summary>
29. abstract class State
30. {
31.
      public abstract void Handle(Context context);
32.
33.
34. /// <summary>
35. /// A 'ConcreteState' class
36. /// </summary>
37. class ConcreteStateA: State
38. {
39.
      public override void Handle(Context context)
40.
       context.State = new ConcreteStateB();
41.
     }
42.
43.
44.
45. /// <summary>
46. /// A 'ConcreteState' class
47. /// </summary>
48. class ConcreteStateB: State
49. {
      public override void Handle(Context context)
       context.State = new ConcreteStateA();
     }
54.
56. /// <summary>
57. /// The 'Context' class
58. /// </summary>
59. class Context
```

```
60. {
61.
      private State _state;
62.
63.
      // Constructor
      public Context(State state)
64.
65.
       this.State = state;
67.
      }
68.
      // Gets or sets the state
      public State State
70.
71.
72.
       get { return _state; }
73.
       set
74.
75.
        _state = value;
76.
        Console.WriteLine("State: " +
          _state.GetType().Name);
77.
78.
       }
      }
79.
80.
      public void Request()
81.
82.
       _state.Handle(this);
83.
84.
      }
85. }
```

Real-world code

```
1. {
2. /// <summary>
    /// MainApp startup class for Real-World
4. /// State Design Pattern.
5.
    /// </summary>
    class MainApp
6.
7. {
8.
     /// <summary>
9.
     /// Entry point into console application.
10.
     /// </summary>
     static void Main()
11.
12.
      // Open a new account
13.
14.
       Account account = new Account("Jim Johnson");
```

```
15.
       // Apply financial transactions
16.
17.
       account.Deposit(500.0);
18.
       account.Deposit(300.0);
       account.Deposit(550.0);
19.
       account.PayInterest();
       account.Withdraw(2000.00);
       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.
      protected double interest;
37.
38.
      protected double lowerLimit;
      protected double upperLimit;
40.
41.
      // Properties
42.
      public Account Account
43.
       get { return account; }
44.
       set { account = value; }
45.
46.
47.
      public double Balance
48.
49.
       get { return balance; }
51.
       set { balance = value; }
52.
      }
53.
54.
      public abstract void Deposit(double amount);
      public abstract void Withdraw(double amount);
      public abstract void PayInterest();
57.
58.
59.
60.
    /// <summary>
61. /// A 'ConcreteState' class
62. /// <remarks>
```

```
63. /// Red indicates that account is overdrawn
64. /// </remarks>
65. /// </summary>
66. class RedState: State
67. {
68.
      private double _serviceFee;
69.
      // Constructor
71.
      public RedState(State state)
72.
73.
       this.balance = state.Balance;
74.
       this.account = state.Account;
75.
       Initialize();
76.
      }
77.
      private void Initialize()
78.
79.
80.
       // Should come from a datasource
       interest = 0.0;
81.
82.
       lowerLimit = -100.0;
       upperLimit = 0.0;
83.
       _serviceFee = 15.00;
84.
85.
      }
86.
      public override void Deposit(double amount)
87.
88.
89.
       balance += amount;
90.
       StateChangeCheck();
91.
92.
93.
      public override void Withdraw(double amount)
94.
95.
       amount = amount - _serviceFee;
       Console. WriteLine("No funds available for withdrawal!");
97.
98.
99.
      public override void PayInterest()
100. {
      // No interest is paid
102. }
104.
      private void StateChangeCheck()
106.
       if (balance > upperLimit)
107.
        account.State = new SilverState(this);
108.
109.
       }
110. }
```

```
111. }
112.
113. /// <summary>
114. /// A 'ConcreteState' class
115. /// <remarks>
116. /// Silver indicates a non-interest bearing state
117. /// </remarks>
118. /// </summary>
119. class SilverState: State
120. {
121. // Overloaded constructors
122.
123. public SilverState(State state):
124.
      this(state.Balance, state.Account)
125. {
126. }
127.
128.
      public SilverState(double balance, Account account)
129. {
130.
     this.balance = balance;
131.
      this.account = account;
      Initialize();
132.
133.
134.
      private void Initialize()
135.
136. {
137.
      // Should come from a datasource
138.
       interest = 0.0;
139.
       lowerLimit = 0.0;
140.
       upperLimit = 1000.0;
141. }
142.
143.
      public override void Deposit(double amount)
144. {
145.
       balance += amount;
       StateChangeCheck();
146.
147. }
148.
149.
      public override void Withdraw(double amount)
150. {
       balance -= amount;
152.
       StateChangeCheck();
153.
154.
      public override void PayInterest()
156. {
157.
       balance += interest * balance;
158.
       StateChangeCheck();
```

```
159. }
160.
      private void StateChangeCheck()
       if (balance < lowerLimit)</pre>
164.
        account.State = new RedState(this);
166.
       else if (balance > upperLimit)
168.
169.
        account.State = new GoldState(this);
170.
       }
171. }
172. }
173.
174. /// <summary>
175. /// A 'ConcreteState' class
176. /// <remarks>
177. /// Gold indicates an interest bearing state
178. /// </remarks>
179. /// </summary>
180. class GoldState: State
181. {
182. // Overloaded constructors
183. public GoldState(State state)
184.
      : this(state.Balance, state.Account)
185. {
186. }
187.
      public GoldState(double balance, Account account)
188.
189. {
190.
       this.balance = balance;
       this.account = account;
192.
       Initialize();
193. }
194.
      private void Initialize()
196. {
197.
      // Should come from a database
198.
       interest = 0.05;
199.
       lowerLimit = 1000.0;
200.
       upperLimit = 10000000.0;
201. }
202.
      public override void Deposit(double amount)
204.
205.
       balance += amount;
206.
       StateChangeCheck();
```

```
207. }
208.
209. public override void Withdraw(double amount)
210. {
       balance -= amount;
212.
       StateChangeCheck();
213. }
214.
      public override void PayInterest()
216. {
       balance += interest * balance;
217.
218.
       StateChangeCheck();
219. }
220.
      private void StateChangeCheck()
222. {
223.
       if (balance < 0.0)
224.
225.
       account.State = new RedState(this);
226.
227.
       else if (balance < lowerLimit)
228.
229.
       account.State = new SilverState(this);
231. }
232. }
233.
234. /// <summary>
235. /// The 'Context' class
236. /// </summary>
237. class Account
238. {
239. private State _state;
240. private string _owner;
241.
242. // Constructor
243. public Account(string owner)
244. {
245. // New accounts are 'Silver' by default
246.
      this. owner = owner;
247.
      this._state = new SilverState(0.0, this);
248.
249.
250. // Properties
251. public double Balance
252. {
253. get { return _state.Balance; }
254. }
```

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