35

STATE

This pattern was previously described in GoF95.

DESCRIPTION

The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed.

A simple example of this would be the case of a user selecting a specific font style or color in an HTML editor. When a user selects a different font style or color, the properties of the editor object change. This can be considered as a change in its internal state.

The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in. In other words, in the case of an object of such a class, some or all of its behavior is completely influenced by its current state. In the State design pattern terminology, such a class is referred to as a *Context* class. A `Context` object can alter its behavior when there is a change in its internal state and is also referred as a `Stateful` object.

STATEFUL OBJECT: AN EXAMPLE

Most of the HTML editors available today offer different views of an HTML page at the time of creation. Let us consider one such editor that offers three views of a given Web page as follows:

1. *Design view* — In this view, a user is allowed to visually create a Web page without having to know about the internal HTML commands.
2. *HTML view* — This view offers a user the basic structure of the Web page in terms of the HTML tags and lets a user customize the Web page with additional HTML code.
3. *Quick page view* — This view provides a preview of the Web page being created.

When a user selects one of these views (change in the state of the Editor object), the behavior of the Editor object changes in terms of the way the current Web page is displayed.

The State pattern suggests moving the state-specific behavior out of the Context class into a set of separate classes referred to as *State classes*. Each of the many different states that a Context object can exist in can be mapped into a separate State class. The implementation of a State class contains the context behavior that is specific to a given state, not the overall behavior of the context itself.

The context acts as a client to the set of State objects in the sense that it makes use of different State objects to offer the necessary state-specific behavior to an application object that uses the context in a seamless manner.

In the absence of such a design, each method of the context would contain complex, inelegant conditional statements to implement the overall context behavior in it. For example,

```
public Context{
           …
           …
  someMethod(){
    if (state_1){
      //do something
    }else if (state_2){
      //do something else
    }
           …
           …
  }
           …
           …
}
```

By encapsulating the state-specific behavior in separate classes, the context implementation becomes simpler to read: free of too many conditional statements such as if-else or switch-case constructs. When a Context object is first created, it initializes itself with its initial State object. This State object becomes the current State object for the context. By replacing the current State object with a new State object, the context transitions to a new state. The client application using the context is not responsible for specifying the current State object for the context, but instead, each of the State classes representing specific states are expected to provide the necessary implementation to transition the context into other states.

When an application object makes a call to a Context method (behavior), it forwards the method call to its current State object.

```
public Context{
           …
           …
  someMethod(){
    objCurrentState.someMethod();
  }
           …
           …
}
```

EXAMPLE

The following State pattern example takes advantage of polymorphism to implement such state-specific behavior. Polymorphism allows two objects with the same method signatures and completely different implementations to be treated in an identical manner.

To use polymorphism, classes that implement the same method differently are derived from a common parent class. Let us say that a client program is written to operate on objects of the superclass type. What the client program thinks of as a parent class object could in reality be an instance of any of its subclasses. The client remains oblivious to this fact. When the client program invokes a method defined in the superclass, the method that gets called is actually the subclass method that overrides the superclass version. In other words, polymorphism encapsulates (hides) the type of the object.

Let us consider a business account at a bank with the overdraft facility. Such an account can exist in any one of the following three states at any given point of time:

1. *No transaction fee state* — As long as the account balance remains greater than the minimum balance, no transaction fee will be charged for any deposit or withdrawal transaction. The example application has the minimum balance as $2,000.
2. *Transaction fee state* — An account is considered to be in the transaction fee state when the account balance is positive but below the minimum balance. A transaction fee will be charged for any deposit or withdrawal transaction in this state. The example application has the transaction fee in this state as $2.
3. *Overdrawn state* — This is the state of the account when an account balance is negative but within the overdraft limit. A transaction fee will be charged for any deposit or withdrawal transactions in this state. The example application has the transaction fee in this state as $5 and the overdraft limit is maintained as $1,000.

In all three states, a withdrawal transaction that exceeds the overdraft limit is not allowed. Figure 35.1 depicts possible state transitions for an account and Table 35.1 shows how these transitions can occur.

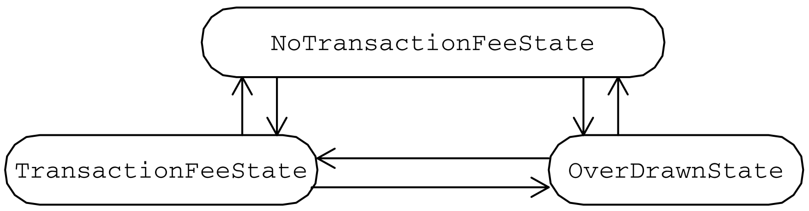


Figure 35.1 State Transitions among Different Account States

Table 35.1 State Transitions among Different Account States

From	*To*	*What Causes the Transition*
No transaction fee state	Transaction fee state	A withdrawal that can make the balance positive but less than the minimum balance.
	Overdrawn state	A withdrawal that can make the balance negative.
Transaction fee state	No transaction fee state	A deposit that can make the balance greater than the minimum balance.
	Overdrawn state	A withdrawal that can make the balance negative.
Overdrawn state	No transaction fee state	A deposit that can make the balance greater than the minimum balance.
	Transaction fee state	A deposit that can make the balance positive but less than the minimum balance.

Let us design a representation for the business account in the form of the `BusinessAccount` class as in Figure 35.2 and Listing 35.1.

The `BusinessAccount` class offers the basic functionality in the form of methods to enable a client object to perform deposit and withdrawal operations. In addition, the `BusinessAccount` class defines some of the transaction limits and offers accessor methods to read its state.

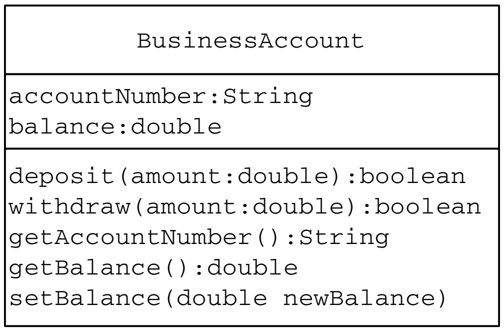

BusinessAccount
accountNumber:String
balance:double
deposit(amount:double):boolean
withdraw(amount:double):boolean
getAccountNumber():String
getBalance():double
setBalance(double newBalance)

Figure 35.2 `BusinessAccount` Representation

Listing 35.1 `BusinessAccount` **Class**

```
public class BusinessAccount {
  public static final double MIN_BALANCE = 2000.00;
  public static final double OVERDRAW_LIMIT = -1000.00;
  public static final double TRANS_FEE_NORMAL = 2.00;
  public static final double TRANS_FEE_OVERDRAW = 5.00;
  public static final String ERR_OVERDRAW_LIMIT_EXCEED =
    "Error: Transaction cannot be processed. " +
      "Overdraw limit exceeded.";
  private State objState;
  private String accountNumber;
  private double balance;
  public void setState(State newState) {
    objState = newState;
  }
  public State getState() {
    return objState;
  }
  public String getAccountNumber() {
    return accountNumber;
  }
  public boolean deposit(double amount) {
    //…
  }
  public boolean withdraw(double amount) {
    //…
  }
  public BusinessAccount(String accountNum) {
    accountNumber = accountNum;
    objState = State.InitialState(this);
  }
  public double getBalance() {
    return balance;
  }
  public void setBalance(double newBalance) {
    balance = newBalance;
  }
}
```

Let us define a common State class (Listing 35.2) that contains the business account behavior that is common across all states.

Instead of keeping the state-specific behavior inside the BusinessAccount class, by applying the State pattern, the behavior specific to each of the three states can be implemented in the form of three separate subclasses — NoTransactionFeeState, TransactionFeeState and OverDrawnState (Listing 35.3 through Listing 35.5) — of the State class. Figure 35.3 shows the resulting class hierarchy. The common parent State class declares the interface to be used by different client objects to access the services of the objects in the State class hierarchy. If a client object is designed to use the services of an object of the common parent State class type, it can access the services offered by its subclasses in a seamless manner.

Each of the State subclasses is designed to contain the behavior specific to a given state of the business account. In addition, these subclasses know the state it should transition to and when to make that transition. Each of these State subclasses implements this state transition functionality by overriding the parent class transitionState method as per the state transition rules detailed in Table 35.1.

While the state-specific behavior is separated out from the BusinessAccount, the state (i.e., the account balance) is still maintained within the BusinessAccount class. Because the behavior contained in each of the State objects is specific to a state of the business account represented by the BusinessAccount class, a State object should be able to read the BusinessAccount object state. To facilitate this, each of the State objects is designed to contain an object reference of the BusinessAccount type. When a State object is created, it is configured with a BusinessAccount instance. Using this BusinessAccount object, a state object can check or alter the state of the business account it represents.

Because the state-specific behavior of a business account is contained in the State class hierarchy, the BusinessAccount needs a way to access the behavior specific to its current state. This requirement can be addressed by enhancing the BusinessAccount class design so that a BusinessAccount object maintains an object reference instance variable of type State to store its current state object. When a BusinessAccount object is first created, it sets an instance of the NoTransactionFeeState class (the default state) as its current State object. Whenever a client object invokes a method such as deposit or withdraw on the BusinessAccount object, it forwards the method call to its current State object. Figure 35.4 and Listing 35.6 show the revised BusinessAccount class representation.

The BusinessAccount class represents the business account and acts as the context in this example. Figure 35.5 shows the overall class association.

Let us design a test client AccountManager to allow a user to perform different transactions on a business account. When executed, the AccountManager:

- Creates a BusinessAccount object that represents a business account.
- Displays the necessary user interface as in Figure 35.6 to allow a user to perform deposit and withdrawal transactions that can make the business account go through different states.

Listing 35.2 `State` Class

```
public class State {
  private BusinessAccount context;
  public BusinessAccount getContext() {
    return context;
  }
  public void setContext(BusinessAccount newAccount) {
    context = newAccount;
  }
  public State transitionState() {
    return null;
  }
  public State(BusinessAccount account) {
    setContext(account);
  }
  public State(State source) {
    setContext(source.getContext());
  }
  public static State InitialState(BusinessAccount account) {
    return new NoTransactionFeeState(account);
  }
  public boolean deposit(double amount) {
    double balance = getContext().getBalance();
    getContext().setBalance(balance + amount);
    transitionState();
    System.out.println("An amount " + amount +
                       " is deposited ");
    return true;
  }
  public boolean withdraw(double amount) {
    double balance = getContext().getBalance();
    getContext().setBalance(balance - amount);
    transitionState();
    System.out.println("An amount " + amount +
                       " is withdrawn ");
     return true;
  }
}
```

Listing 35.3 `NoTransactionFeeState` Class

```
public class NoTransactionFeeState extends State {
  public NoTransactionFeeState(BusinessAccount account) {
    super(account);
  }
  public NoTransactionFeeState(State source) {
    super(source);
  }
  public boolean deposit(double amount) {
    return super.deposit(amount);
  }
  public boolean withdraw(double amount) {
    double balance = getContext().getBalance();
    if ((balance - amount) >
        BusinessAccount.OVERDRAW_LIMIT) {
      super.withdraw(amount);
      return true;
    } else {
      System.out.println(
        BusinessAccount.ERR_OVERDRAW_LIMIT_EXCEED);
      return false;
    }
  }
  public State transitionState() {
    double balance = getContext().getBalance();
    if (balance < 0) {
      getContext().setState(new OverDrawnState(this));
    } else {
      if (balance < BusinessAccount.MIN_BALANCE) {
        getContext().setState(
          new TransactionFeeState(this));
      }
    }
    return getContext().getState();
  }
}
```

Listing 35.4 `TransactionFeeState` Class

```
public class TransactionFeeState extends State {
  public TransactionFeeState(BusinessAccount account) {
    super(account);
  }
  public TransactionFeeState(State source) {
    super(source);
  }
  public State transitionState() {
    double balance = getContext().getBalance();
    if (balance < 0) {
      getContext().setState(new OverDrawnState(this));
    } else {
      if (balance >= BusinessAccount.MIN_BALANCE) {
        getContext().setState(
          new NoTransactionFeeState(this));
      }
    }
    return getContext().getState();
  }
  public boolean deposit(double amount) {
    double balance = getContext().getBalance();
    getContext().setBalance(balance -
        BusinessAccount.TRANS_FEE_NORMAL);
    System.out.println(
      "Transaction Fee was charged due to " +
      "account status " +
      "(less than minimum balance)");
    return super.deposit(amount);
  }
  public boolean withdraw(double amount) {
    double balance = getContext().getBalance();
    if ((balance - BusinessAccount.TRANS_FEE_NORMAL -
        amount) > BusinessAccount.OVERDRAW_LIMIT) {
      getContext().setBalance(balance -
        BusinessAccount.TRANS_FEE_NORMAL);
```

(continued)

Listing 35.4 `TransactionFeeState` Class (Continued)

```
        System.out.println(
          "Transaction Fee was charged due to " +
          "account status " +
          "(less than minimum balance)");
        return super.withdraw(amount);
      } else {
        System.out.println(
          BusinessAccount.ERR_OVERDRAW_LIMIT_EXCEED);
        return false;
      }
    }
  }
```

Every deposit or withdrawal transaction initiated through the user interface translates to a `deposit(double)` or `withdraw(double)` method call on the `BusinessAccount` object that is created when the `AccountManager` is executed. The `BusinessAccount` object in turn forwards this call to its internal current `State` object. The current `State` object executes the behavior it contains and sets an appropriate `State` object as the `BusinessAccount` object's current `State` object. In this manner the `Context` class (`BusinessAccount`) and its state-specific behavior (`State` class hierarchy) are completely separated from each other. When a new state-specific behavior is added or the behavior specific to a state is altered, the actual `Context` class `BusinessAccount` remains unaffected.

Listing 35.5 `OverDrawnState` Class

```
public class OverDrawnState extends State {
  public void sendMailToAccountHolder() {
    System.out.println (
      "Attention: Your Account is Overdrawn");
  }
  public OverDrawnState(BusinessAccount account) {
    super(account);
    sendMailToAccountHolder();
  }
  public OverDrawnState(State source) {
    super(source);
    sendMailToAccountHolder();
  }
```

(continued)

Listing 35.5 `OverDrawnState` Class (Continued)

```
      public State transitionState() {
     double balance = getContext().getBalance();
     if (balance >= BusinessAccount.MIN_BALANCE)
       getContext().setState(
         new NoTransactionFeeState(this));
     else if (balance >= 0)
       getContext().setState(new TransactionFeeState(this));
     return getContext().getState();
   }
   public boolean deposit(double amount) {
     double balance = getContext().getBalance();
     getContext().setBalance(balance -
         BusinessAccount.TRANS_FEE_OVERDRAW);
     System.out.println("Transaction Fee was charged " +
                        "due to account status(Overdrawn)");
     return super.deposit(amount);
   }
   public boolean withdraw(double amount) {
     double balance = getContext().getBalance();
     if ((balance - BusinessAccount.TRANS_FEE_OVERDRAW -
           amount) > BusinessAccount.OVERDRAW_LIMIT) {
       getContext().setBalance(balance -
         BusinessAccount.TRANS_FEE_OVERDRAW);
 System.out.println(
         "Transaction Fee was charged due to " +
         "account status(Overdrawn)");
       return super.withdraw(amount);
     } else {
       System.out.println(
         BusinessAccount.ERR_OVERDRAW_LIMIT_EXCEED);
       return false;
     }
   }
 }
```

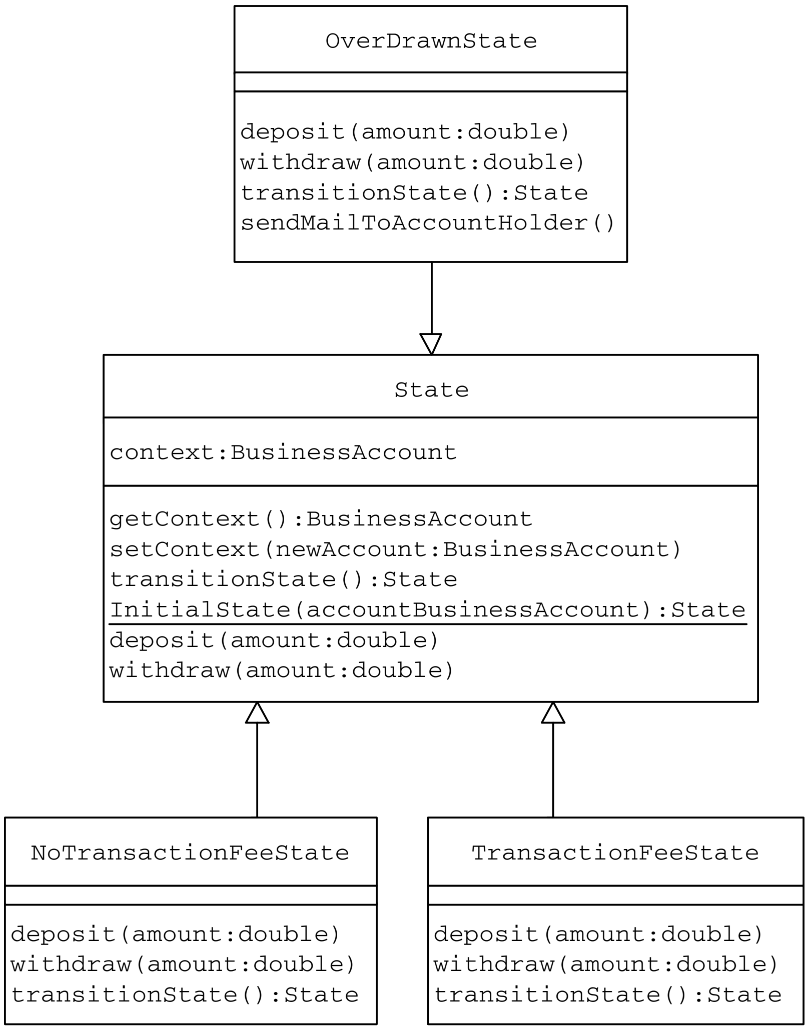


Figure 35.3 `BusinessAccount` `State` **Class Hierarchy**

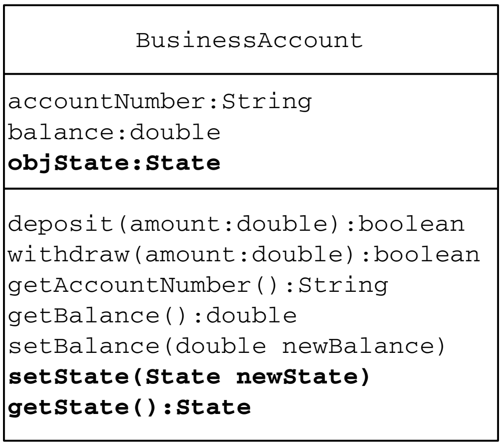

BusinessAccount
accountNumber:String
balance:double
objState:State
deposit(amount:double):boolean
withdraw(amount:double):boolean
getAccountNumber():String
getBalance():double
setBalance(double newBalance)
setState(State newState)
getState():State

Figure 35.4 `BusinessAccount` Representation: Revised

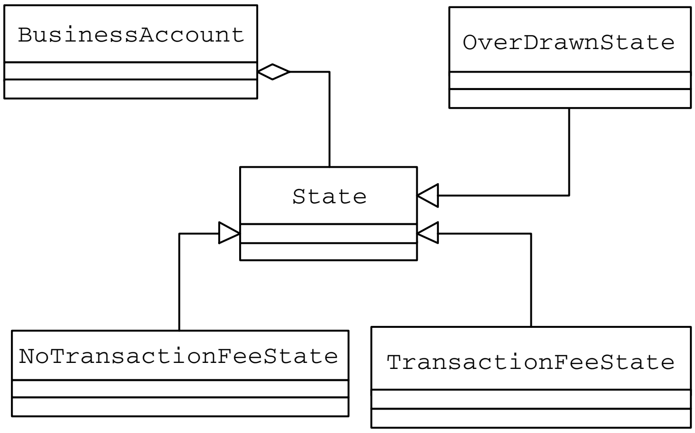
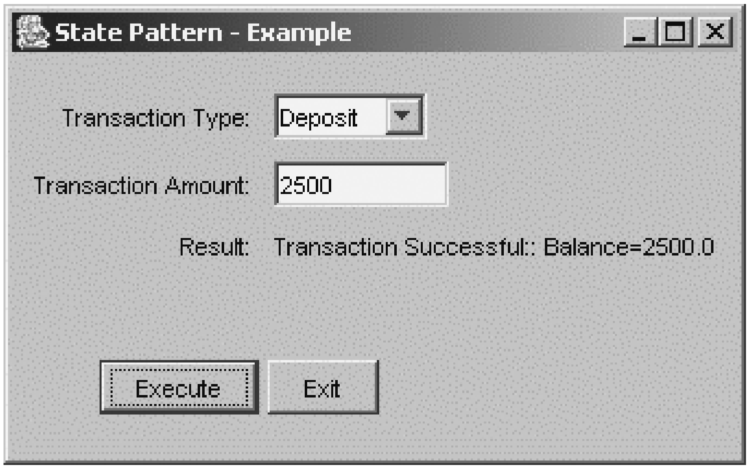


Figure 35.5 Class Association

Listing 35.6 `BusinessAccount` Class: Revised

```
public class BusinessAccount {
  public static final double MIN_BALANCE = 2000.00;
  public static final double OVERDRAW_LIMIT = -1000.00;
  public static final double TRANS_FEE_NORMAL = 2.00;
  public static final double TRANS_FEE_OVERDRAW = 5.00;
  public static final String ERR_OVERDRAW_LIMIT_EXCEED =
    "Error: Transaction cannot be processed. " +
      "Overdraw limit exceeded.";
  private State objState;
  private String accountNumber;
  private double balance;
  public void setState(State newState) {
    objState = newState;
  }
  public State getState() {
    return objState;
  }
  public String getAccountNumber() {
    return accountNumber;
  }
  public boolean deposit(double amount) {
    return getState().deposit(amount);
  }
  public boolean withdraw(double amount) {
    return getState().withdraw(amount);
  }
  public BusinessAccount(String accountNum) {
    accountNumber = accountNum;
    objState = State.InitialState(this);
  }
  public double getBalance() {
    return balance;
  }
  public void setBalance(double newBalance) {
    balance = newBalance;
  }
}
```



Result: Transaction Successful:: Balance=2500.0

Figure 35.6 **`AccountManager` User Interface**

Table 35.2 Membership Account State Transitions

From	*To*	*What Causes the Transition*
Active	Due	From the 5th to the 10th of every month.
	Canceled	If a member wants to cancel the account explicitly.
Due	Active	If payment is made before the 10th of the month.
	Unpaid	If payment is not made before the 10th of the month.
	Canceled	If a member wants to cancel the account explicitly.
Unpaid	Active	If payment is made before the account is canceled.
	Canceled	If a member wants to cancel the account explicitly. If the account stays in the unpaid state for more than 15 days.
Canceled	Active	If all previous payment dues are cleared.

PRACTICE QUESTIONS

1. Assume that a membership account at a Web site can exist in one of four different states:
   - *Active* — This is the state of an account when it is in good standing.
   - *Due* — Every account is supposed to be paid for by the 5th of every month, but members are given up to the 10th to make the payment. An account remains in the due state until the 10th of every month, if not paid before the 10th.
   - *Unpaid* — If the payment is not made by the 10th of every month, the account enters into the unpaid state. In this state, members are not allowed to use premium services. But the membership still remains active and members can use basic services.

- *Canceled* — If an account remains in the unpaid state for more than 15 days then it is canceled. Table 35.2 lists different membership account state transitions.

  a. Design a `MemberAccount` class whose instances can be used to represent membership accounts.

  b. Apply the State pattern to design the state-specific behavior of a membership account in the form of a group of `State` classes that are part of a class hierarchy with a common parent.

2. An order at an online store can be in one of the following states:
   - Not Submitted
   - Submitted
   - Received
   - Processed
   - Shipped
   - Canceled

   a. Define a state transition table (similar to Table 35.1) for an order.

   b. Design an `Order` class whose instances can be used to represent orders. Design the state-specific behavior of an order in the form of a set of `State` classes with a common parent class.