38

TEMPLATE METHOD

This pattern was previously described in GoF95.

DESCRIPTION

The Template Method pattern is one of the simplest and most frequently used design patterns in object-oriented applications.

The Template Method pattern can be used in situations when there is an algorithm, some steps of which could be implemented in multiple different ways. In such scenarios, the Template Method pattern suggests keeping the outline of the algorithm in a separate method referred to as a *template method* inside a class, which may be referred to as a *template class*, leaving out the specific implementations of the variant portions (steps that can be implemented in multiple different ways) of the algorithm to different subclasses of this class.

The `Template` class does not necessarily have to leave the implementation to subclasses in its entirety. Instead, as part of providing the outline of the algorithm, the `Template` class can also provide some amount of implementation that can be considered as invariant across different implementations. It can even provide default implementation for the variant parts, if appropriate. Only specific details will be implemented inside different subclasses. This type of implementation eliminates the need for duplicate code, which means a minimum amount of code to be written.

Using the Java programming language, the `Template` class can be designed in one of the following two ways.

Abstract Class

This design is more suitable when the `Template` class provides only the outline of the algorithm without any default implementation for its variant parts. Assuming that different steps of the algorithm can be made into individual methods:

- The `Template` method can be a concrete, nonabstract method with calls to other methods that represent different steps of the algorithm.
- The `Template` class can implement invariant parts of the algorithm as a set of nonabstract methods.

- The set of variant steps can be designed as abstract methods. Specific implementations can be provided for these abstract methods inside a set of concrete subclasses of the abstract `Template` class.

In this design, the `Abstract` class declares methods and each of the subclasses implement these methods in a manner that is specific to it without altering the outline of the algorithm.

Concrete Class

This design is more suitable when the `Template` class provides, besides the outline of the algorithm, the default implementation for its variant parts. Assuming that different steps of the algorithm can be made into individual methods:

- The `Template` method can be a concrete, nonabstract method with calls to other methods that represent different steps of the algorithm.
- The `Template` class can implement invariant parts of the algorithm as a set of nonabstract methods.
- The set of variant steps can be designed as nonabstract methods with the default implementation. Subclasses of the `Template` class can override these methods to provide specific implementations without altering the outline of the algorithm.

From both the design strategies, it can be seen that the Template pattern implementation relies heavily on inheritance and function overriding. Hence, whenever inheritance is used for implementing the specifics, it can be said that Template Method pattern is used in its simplest form.

EXAMPLE

Let us design an application to check the validity of a given credit card. For simplicity, let us consider only three types of credit cards — Visa, MasterCard and Diners Club. The application carries out a series of validations on the input credit card information. Table 38.1 lists different steps in the process of validating different credit cards.

As can be seen from Table 38.1, some steps of the validation algorithm are the same across all three of the credit cards while some are different. The Template Method pattern can be applied in designing this process.

Let us define an abstract `CreditCard` class (Figure 38.1 and Listing 38.1) with:

- The `Template` method `isValid` that outlines the validation algorithm.
- A set of concrete methods implementing Step 1, Step 4 and Step 5 from Table 38.1.
- A set of abstract methods designated to implement Step 2, Step 3 and Step 6 from Table 38.1. It is to be noted that even after the `CheckSum` validation is successful, it cannot be guaranteed that a given credit card is valid. It is possible that the account may have been revoked or over the limit.

Table 38.1 Different Steps in the Validation Process

Step	*Check*	*Visa*	*MasterCard*	*Diners Club*
1	Expiration date	>Today	>Today	>Today
2	Length	13, 16	16	14
3	Prefix	4	51 through 55	30, 36, 38
4	Valid characters	0 through 9	0 through 9	0 through 9
5	Check digit algorithm	Mod 10	Mod 10	Mod 10
6	Account in good standing	Use custom Visa API	Use custom MasterCard API	Use custom Diners Club API

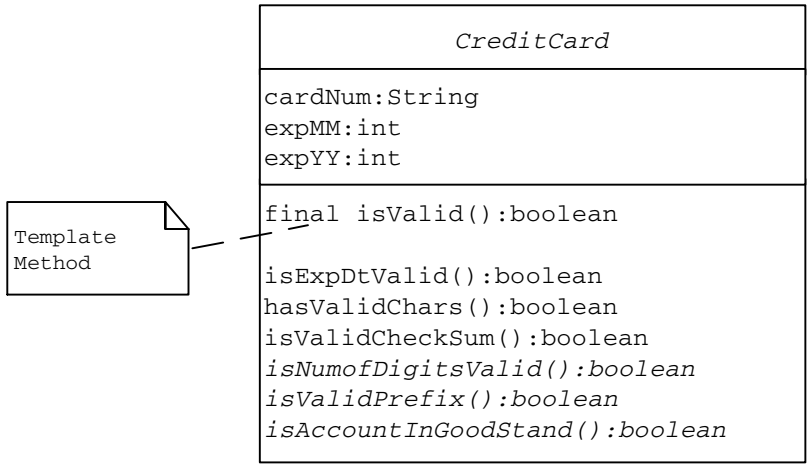

Figure 38.1 `CreditCard` Template Class

Hence, a check with the credit card company (Visa, MasterCard, Diners Club) is required to make sure that the account is in good standing. This step requires custom programming to interface with the credit card company database and is considered to be different for different credit card types. Hence the `isAccountInGoodStand` method is designed as an abstract method to be implemented by different subclasses.

The most significant method in the design is the `isValid Template` method. This method invokes different methods designed to implement different steps of the algorithm. It is to be noted that the `Template` method `isValid` is specified as a *final* method to prevent subclasses from overriding it. Subclasses are expected to override only abstract methods to provide specific implementation and are *not* supposed to alter the outline of the algorithm.

Listing 38.1 Abstract `CreditCard` Class

```
public abstract class CreditCard {
  protected String cardNum;
  protected int expMM, expYY;
  public CreditCard(String num, int expMonth, int expYear) {
    cardNum = num;
    expMM = expMonth;
    expYY = expYear;
  }
  public boolean isExpDtValid() {
    Calendar cal = Calendar.getInstance();
    cal.setTime(new Date());
    int mm = cal.get(Calendar.MONTH) + 1;
    int yy = cal.get(Calendar.YEAR);
    boolean result =
      (yy > expYY) || ((yy == expYY) && (mm > expMM));
    return (!result);
  }
  private boolean hasValidChars() {
    String validChars = "0123456789";
    boolean result = true;
    for (int i = 0; i < cardNum.length(); i++) {
      if (validChars.indexOf(cardNum.substring(i, i + 1)) <
          0) {
        result = false;
        break;
      }
    }
    return result;
  }
  private boolean isValidCheckSum() {
    boolean result = true;
    int sum = 0;
    int multiplier = 1;
    int strLen = cardNum.length();
    for (int i = 0; i < strLen; i++) {
      String digit = cardNum.substring(strLen - i - 1,
                     strLen - i);
```

(continued)

Listing 38.1 Abstract `CreditCard` Class (Continued)

```
    int currProduct =
      new Integer(digit).intValue() * multiplier;
    if (currProduct >= 10)
      sum += (currProduct% 10) + 1;
    else
      sum += currProduct;
    if (multiplier == 1)
      multiplier++;
    else
      multiplier – ;
  }
  if ((sum% 10) != 0)
    result = false;
  return result;
}
/* methods to be overridden by sub-classes. */
public abstract boolean isNumOfDigitsValid();
public abstract boolean isValidPrefix();
public abstract boolean isAccountInGoodStand();
/* Final method - subclasses cannot override
   ***TEMPLATE METHOD***
*/
public final boolean isValid() {
  if (!isExpDtValid()) {
    System.out.println(" Invalid Exp Dt. ");
    return false;
  }
  if (!isNumOfDigitsValid()) {
    System.out.println(" Invalid Number of Digits ");
    return false;
  }
  if (!isValidPrefix()) {
    System.out.println(" Invalid Prefix ");
    return false;
  }
```

(continued)

Listing 38.1 Abstract `CreditCard` Class (Continued)

```
    if (!hasValidChars()) {
      System.out.println(" Invalid Characters ");
      return false;
    }
    if (!isValidCheckSum()) {
      System.out.println(" Invalid Check Sum ");
      return false;
    }
    if (!isAccountInGoodStand()) {
      System.out.println(
        " Account is Inactive/Revoked/Over the Limit ");
      return false;
    }
    return true;
  }
}
```

In Java programming language, a subclass cannot override the following two types of methods of its parent class:

- private methods
- final methods irrespective of the associated access specifier

Let us define three subclasses —`VisaCard`, `MasterCard` and `DinersCard` — of the `CreditCard Template` class, each providing implementation for all abstract methods declared in the parent class (Listing 38.2 through Listing 38.4).

The resulting class association can be depicted as in Figure 38.2.

With the above design in place, any client looking to validate credit card information would simply create an instance of an appropriate `CreditCard` subclass and invoke the `isValid` method.

```
public class Client {
  public static void main(String[] args) {
    CreditCard cc =
      new VisaCard("1234123412341234,"11, 2004);
    if (cc.isValid())
      System.out.println("Valid Credit Card Information");
  }
}
```

Listing 38.2 `VisaCard` Class

```
public class VisaCard extends CreditCard {
  public VisaCard(String num, int expMonth, int expYear) {
    super(num, expMonth, expYear);
  }
  public boolean isNumOfDigitsValid() {
    if ((cardNum.length() == 13) ||
        (cardNum.length() == 16)) {
      return true;
    } else {
      return false;
    }
  }
  public boolean isValidPrefix() {
    String prefix = cardNum.substring(0, 1);
    if (prefix.equals("4")) {
      return true;
    } else {
      return false;
    }
  }
  public boolean isAccountInGoodStand() {
    /*
      Make necessary VISA API calls to
      perform other checks.
    */
    return true;
  }
}
```

ADDITIONAL NOTES

Mod 10 Check Digit Algorithm

In general, a check digit is a digit added to a number that helps in checking the authenticity of the number. The Mod 10 check digit algorithm can be used to validate such a number associated with a check digit.

Listing 38.3 `MasterCard` Class

```
public class MasterCard extends CreditCard {
  public MasterCard(String num, int expMonth, int expYear) {
    super(num, expMonth, expYear);
  }
  public boolean isNumOfDigitsValid() {
    if (cardNum.length() == 16) {
      return true;
    } else {
      return false;
    }
  }
  public boolean isValidPrefix() {
    String prefix = cardNum.substring(0, 1);
    String nextChar = cardNum.substring(1, 2);
    String validChars = "12345";
    //51-55
    if ((prefix.equals("5")) &&
         (validChars.indexOf(nextChar) >= 0)) {
      return true;
    } else {
      return false;
    }
  }
  public boolean isAccountInGoodStand() {
    /*
      Make necessary MASTER CARD API calls to
      perform other checks.
    */
    return true;
  }
}
```

Listing 38.4 `DinersCard` Class

```
public class DinersCard extends CreditCard {
  public DinersCard(String num, int expMonth, int expYear) {
    super(num, expMonth, expYear);
  }
  public boolean isNumOfDigitsValid() {
    if (cardNum.length() == 14) {
      return true;
    } else {
      return false;
    }
  }
  public boolean isValidPrefix() {
    String prefix = cardNum.substring(0, 1);
    String nextChar = cardNum.substring(1, 2);
    String validChars = "068";
    //51-55
    if ((prefix.equals("3")) &&
         (validChars.indexOf(nextChar) >= 0)) {
      return true;
    } else {
      return false;
    }
  }
  public boolean isAccountInGoodStand() {
    /*
      Make necessary DINERS CARD API calls to
      perform other checks.
    */
    return true;
  }
}
```

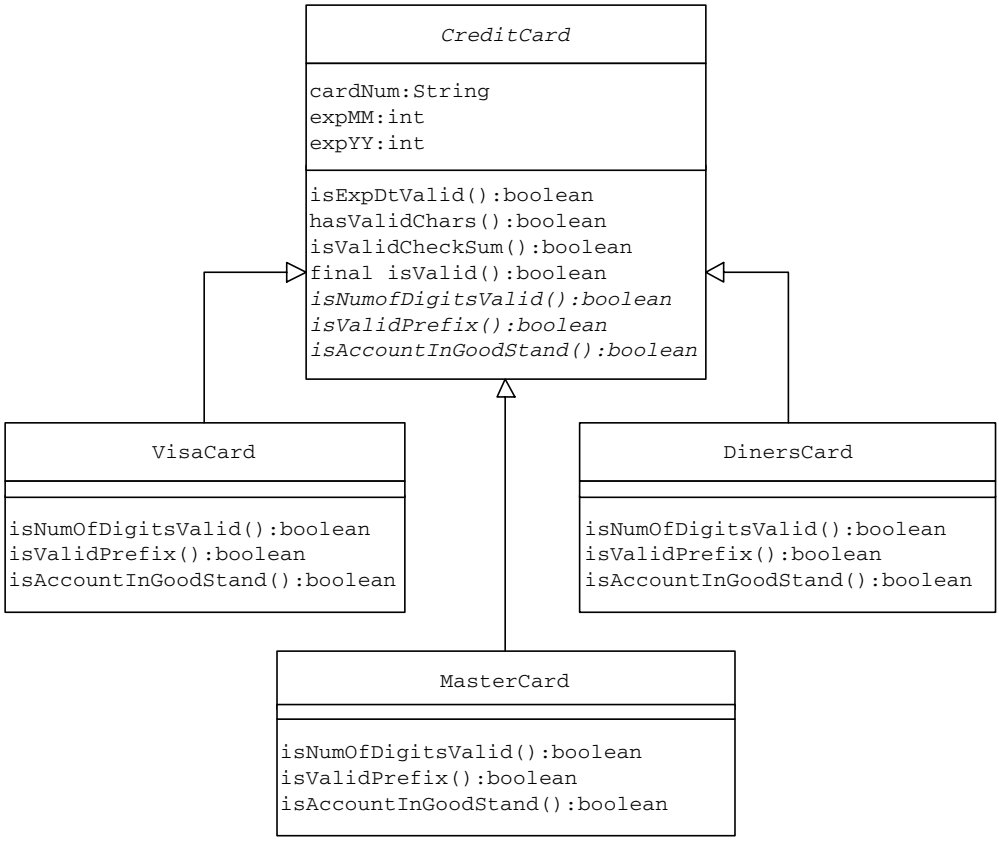

Figure 38.2 `CreditCard` Class Hierarchy

The following steps describe the validation process:

1. Use 194774915 (check digit 5 included) as an example.
   1 9 4 7 7 4 9 1 5
2. Starting from the second digit from right, multiply every alternate digit by 2.
   1 9x2 4 7x2 7 4x2 9 1x2 5
   Result:
   1 18 4 14 7 8 9 2 5
3. Add individual digits in the newly formed products.
   1 1+8 4 1+4 7 8 9 2 5
   Result:
   1 9 4 5 7 8 9 2 5
4. Now sum up all digits in the resultant number from the above step.
   1 +9 +4 +5 +7 +8 +9 +2 +5 = 50
5. Now divide the sum by 10.
   Result:
   50/10 leaves no remainder and a zero remainder proves that the number is valid.

PRACTICE QUESTIONS

1. Identify how the Template Method pattern is used when you design an applet with custom code in any of the applet life-cycle methods (init, start, paint, stop and destroy).
2. Some scenarios involving many different implementations for different steps of an algorithm could lead to a fast growing class hierarchy with a large number of subclasses. What alternatives would you consider in such cases?