CHAPTER 7

Template Method, Factory Method, and Composite

Objectives

The objectives of this chapter are to identify the following:

- Introduce the template, factory, and composite patterns.
- Create basic UML diagrams for these design patterns.
- Review the UML collaboration and deployment diagrams.
**Template Method Pattern**

Use the *Template Method* pattern to:

- Define a skeleton of an algorithm in a base class but defer certain steps to derived classes. This allows a subclass to provide a specific implementation for a single step within an algorithm without using the *strategy* or *bridge* patterns.

*Template Methods* have two (2) participants:

- An *Abstract Class* which defines the abstract operations that will be implemented by the concrete subclasses. This class also defines the skeleton of the algorithm.
- A *Concrete Class* which implements the primitive abstract methods declared in the *abstract class*.

The results we gain from implementing a *Template Method* pattern are:

- An inverted control structure. The parent class, via inheritance and polymorphism, calls the derived class’ methods and not vice versa.

To illustrate this design pattern, the following trivial example:

```cpp
class ClassA {
  protected:
    virtual void doPrint(void) = 0;
  public:
    void print(void);
};
```

The Java equivalent might appear as:

```java
public class ClassA {
  protected abstract void doPrint();
  public void print() { ... }
};
```

Now we implement the *print()* method in the abstract class:

```cpp
void ClassA::print() {
  // do some stuff
  doPrint();
  // do more stuff
}
```

The corresponding Java code might appear as:
public void print() {
    // do some stuff
    doPrint();
    // do more stuff
}

No we need to define the doPrint() method. This will be handled by classes derived from ClassA. For instance, suppose we define a ClassB as follows:

class ClassB : public ClassA {
    protected:
    void doPrint(void) {
        cout << “Class B” << endl;
    }
};

The Java equivalent might appear as:

public class ClassB extends ClassA {
    protected void doPrint() {
        System.out.println(“Class B”);
    }
};

Now we can see how to use this class. As with all other design patterns, this one counts on the use of a base-class object reference an polymorphism to get the correct result.

void main() {
    ClassA * aptr = new ClassB();
    aptr->print();
}

The Java equivalent might appear as:

public static void main(String [] args) {
    ClassA aptr = new ClassB();
    aptr.print();
};

What will the output be?

In addition to the abstract methods, called primitive methods, we can also define hook methods. Hook methods are nothing more than methods for which we give a default behavior in the abstract class. These methods may then be overridden by the derive classes although there is no requirement that they do so. For exam-
ple, we could extend the previous example by implementing a hook method called `printHeader()`:

```cpp
class ClassA {
    protected:
        virtual void doPrint(void) = 0;
        virtual void printHeader(void) {}  
    public:
        void print(void);  
};
```

The Java equivalent might appear as:

```java
public class ClassA {
    protected abstract void doPrint();
    protected abstract void printHeader() {}
    public void print() { ... }
};
```

The default implementation for a hook method is frequently empty; it serves as a placeholder in the event that the client fails to implement its own behavior. Now we can change our template method slightly:

```cpp
void ClassA::print() {
    // do some stuff
    printHeader();
    doPrint();
    // do more stuff
}
```

The corresponding Java code might appear as:

```java
public void print() {
    // do some stuff
    printHeader();
    doPrint();
    // do more stuff
}
```

And finally we can now override the hook method in the derived class:
class ClassB : public ClassA {
    protected:
    void doPrint(void) {
        cout << "Class B" << endl;
    }
    void printHeader(void) {
        cout << "Header" << endl;
    }
};

The Java equivalent might appear as:

public class ClassB extends ClassA {
    protected void doPrint() {
        System.out.println("Class B");
    }
    protected void printHeader() {
        System.out.println("Header");
    }
};

What will the output be this time? What would happen if ClassB failed to override the printHeader() hook method?

In summary, there are three (3) types of methods in the template method pattern:

- **Template methods.** These are never overridden by derived classes.
- **Primitive (abstract) methods.** These must be overridden by derived classes.
- **Hook methods.** These may be overridden by derived classes.

**Factory Method Pattern**

Frequently application frameworks (such as AWT or MFC) use abstract classes to define and maintain their various relationships. However, these frameworks are also responsible for creating these objects. It would prefer to avoid creating a specific kind of object until it knows specifically what it is dealing with. In other words, it can only predict when a new object is needed and not what kind of object is required. For example, a document management application might be derived by many developers, each creating a new kind of document. There is no possible means for the base application to know about the infinite variety of documents that it may be required to handle.

This is where the Factory Method pattern can be used. Use this pattern when:

- One class cannot anticipate the class of objects that it might need to create.
• A class wants its subclasses to define the specific types of objects that it creates.

From this description the factory method pattern appears similar to the template method pattern. It is very similar save that the factory method is creational while the template method is behavioral in nature.

Factory Methods have four (4) participants:
• A Product, the document in our example, which defines the interface that the Application uses.
• A Concrete Product which implements the Product interface. In our example, each specific kind of document would (e.g. Word, Excel, Text, etc.) represent one of these classes.
• A Creator which declares the factory method that is used to return an object of type Product. The creator uses this factory method to create a Product instance. We might define this method to support a default implementation for common objects.
• A Concrete Creator that overrides the base class factory method to return a specific instance of Concrete Product.

The results we gain from implementing a Factory Method pattern are:
• Hooks for subclasses. Be moving the creation of the object off to the subclasses, we gain flexibility. This also allows us to build a default implementation into the base class that can easily be overridden by the derived classes.
• Connects parallel class hierarchies. Sometimes we delegate certain areas of functionality to other classes. Recall the iterator pattern where we isolated the navigation of a set of data from the maintenance of that set. The Aggregate class defined a createIterator() method that was used to construct the actual iterator object. This is a special case of the factory method pattern.

Assume the existence of the following class:

class Creator {
    public:
    virtual Product * Create(int productId);
};

The Java equivalent might appear as:
public class Creator {
    public Product Create(int productId) { ... }
};

Assume a default behavior of the Create() method such that given an Id it generates the appropriate product type:

    Product * Creator::Create(int productId) {
        if (productId == MINE)  return new MyProduct();
        if (productId == YOURS) return new YourProduct();
        // ...
        return (null);
    }

The corresponding Java code might appear as:

    public Product Create(int productId) {
        if (productId == MINE)  return new MyProduct();
        if (productId == YOURS) return new YourProduct();
        // ...
        return (null);
    }

Now a new subclass could be declared:

class MyCreator : public Creator {
    public:
        Product * Create(int productId);
};

The Java equivalent might appear as:

    public class MyCreator extends Creator {
        public Product Create(int productId) { ... }
    }

This new class might override the default behavior as shown below:

    Product * MyCreator::Create(int productId) {
        if (productId == MINE)  return new YourProduct();
        if (productId == YOURS) return new YMyProduct();
        // ...
        return ( Creator::Create(productId) );
    }

The corresponding Java code might appear as:
public Product Create(int productId) {
    if (productId == MINE)  return new MyProduct();
    if (productId == YOURS) return new YourProduct();
    // ...
    return ( super.Create(productId) );
}

The final call to the parent’s Create() method allows the base class processing to handle any cases not covered by the child class. Thus the derived class need only provide code to handle the situations not covered adequately by the parent class.

The factory method pattern can also help reduce the creation of new classes. New creators would ordinarily need to have derived classes built for each concrete instance. This can lead to complex hierarchies containing many objects differing only the type of object being created. Using templates, we can avoid this problem as in the following example:

class Creator {
    public:
        virtual Product * Create() = 0;
};

template <class TheProduct>
class StandardCreator : public Creator {
    public:
        virtual Product * createProduct();
};

template <class TheProduct>
Product * StandardCreator<TheProduct>::CreateProduct() {
    return (new TheProduct);
}

To return new kinds of products we can now create subclasses based on only the product:

class MyProduct : public Product {
    public:
        MyProduct();
};

And now we can build new Creators without subclassing. We simply provide the Product type as the parameter:
Composite Pattern

The Composite pattern allows a developer to compose objects into a tree in order to represent whole-part hierarchies.

Use the Composite pattern when:

- there is a need to represent part-whole hierarchies of objects.
- clients should be able to ignore the differences between objects and compositions of objects. Clients treat all objects in the composite structure uniformly.

Composites have four (4) participants:

- A Component which declares an interface for objects in the composition. It may also implement default behavior within the interface that is shared by all subclasses. Its interface will also contain the methods needed to manage the child components (e.g. add, remove, etc.). This class may also allow a bottom-up traversal by providing a mechanism for accessing a component’s parent.
- A Leaf which represents leaf objects in the composition. Leaves have no children. This class defines behavior common to all primitive objects in the composition.
- The Composite which defines behavior for components that might have children, stores the children, and implements child-related operations in the Component interface. This implementation is frequently just a pass-through to each child within the composite. This continues recursively down the tree until the last child has performed the requested operation.
- The Client which manipulates the objects in the composition through the Component interface.

The results we gain from implementing the Composite pattern are:

- Primitive object can be composed into more complex objects. Complex objects and primitive objects may be referred to in the same way since they implement the same interface.
- The client is simplified. Clients need not treat composites differently from primitives; the difference is transparent since the same interface is supported throughout.
- New components can be added relatively easily. New composite or leaf subclasses can be added without disrupting the existing classes.
• The design may be too general. Because we can easily add new components and leaves, this pattern may result in a hierarchy that cannot easily enforce certain rules. For example, we may have a rule that a given composite can only hold specific kinds of leaves. To perform that kind of checking in this pattern relies on RTTI (run-time type information) and specialized methods within the composite class.

To illustrate this class, consider the following example: a message can be defined as being composed of various fields. Sometimes a message may contain another message (consider email and how an original message may be contained within a reply). We might represent a message as:

```c++
class Message {
    private:
        const char * _text;
    protected:
        Message(const char *);
    public:
        virtual ~Message();
        virtual void Add(Message *);
        virtual void Remove(Message *);
        virtual void Iterator<Message*> * CreateIterator(void);
        virtual void Print(void);
};
```

One kind of message is a field. Consider this as the least possible message:

```c++
class Field : public Message {
    public:
        Field(const char *);
        virtual ~Field();
        virtual void Print(void);
};
```

Finally, we can declare a message composite that acts as the base for all messages containing other messages:
class CompositeMessage : public Message {
    private:
        List<Message *> _message;
    protected:
        CompositeMessage(const char *);
    public:
        virtual ~Message();
        virtual void Add(Message *);
        virtual void Remove(Message *);
        virtual void Iterator<Message*> * CreateIterator(void);
        virtual void Print(void);
    }

    The default implementation of the Print() method might consist of:

    void CompositeMessage::Print(void) {
        Iterator<Message *> * i = CreateIterator();
        for (i->First(); !i->IsDone(); i->Next())
        {
            i->CurrentItem()->Print();
        }
        delete i;
    }

    We can now define a specific kind of message as a subclass of the CompositeMessage class:

        class SpecificMessage : public CompositeMessage {
            public:
                SpecificMessage(const char *);
                virtual ~SpecificMessage();
                virtual void Print(void);
        }

        And now we can use this new specific message:

        SpecificMessage * sm = new SpecificMessage();
        Field * f = new Field("skippy");
        SpecificMessage->add(f);
        SpecificMessage->add(new Field("fred");
        SpecificMessage->Print();
Collaboration Diagrams

Collaboration diagrams are a form of interaction diagram the same way that a sequence diagram is. These two diagrams tend to capture the same content and one can be fairly easily converted into the other. In general collaboration diagrams are more free-form than sequence diagrams.

I tend to ignore these in favor of sequence diagrams, but I want to make sure you are exposed to the stuff. Also, this will give you enough information to sally forth on the last couple of diagrams needed for the project.

Objects
Within collaboration diagrams objects that interact are drawn within boxes. Because each box represents an object, the box contains the object’s name. This name is underlined (which signifies an object instance instead of a class).

Messages
Synchronous messages are drawn between objects as directional arrows pointing from the sender of the message to the target. Thus the message on the arrow represents a method on the target object.

You will sometimes see sequence numbers associated with the messages. I usually don’t bother simply because that’s what I use sequence diagrams for.

Polymorphism
This causes some problems for notation, but nothing major. Instead of the class in the box, we depict the lowest class in the hierarchy that is a superclass of all the classes to which the target might belong. In other words, we choose the class to be the lowest common denominator in the inheritance hierarchy. Some organizations will also place this class name in parenthesis to ensure that a reader knows that the class isn’t accurate. I favor that minor notational change since it lends clarity.

Iterated Messages
A message is sometimes sent to repeatedly to an object, particularly if that object is an aggregate and we are sending the message to each of its constituents. In such a case we represent the targets as one box on top of another with no specific object’s named. This is just a means of representing multiplicity. Also, the message itself is prefixed with an asterisk (*).

Self-Messaging
Sometimes an object will send a message to itself. Just draw this as a standard message arrow pointing from the source object back to itself. This is fairly standard although other approaches have been suggested.
Deployment Diagrams

There are many kinds of deployment diagrams but all I’m concerned with for this class is the software deployment diagram. Essentially this diagram shows logical collections of software components called packages and how those packages are related to or depend upon one another.

This kind of diagram can be useful in several contexts. First, it provides some organization around our components. Second we can easily see interdependencies between the components and plan accordingly (e.g. we might use this information when constructing a make file).

It is possible for one package to contain or be contained in another.