25

VIRTUAL PROXY

This pattern was previously described in GoF95.

DESCRIPTION

The Virtual Proxy pattern is a memory saving technique that recommends postponing an object creation until it is needed; when creating such an object is expensive in terms of the memory usage or the processing involved. In a typical application, different objects make up different parts of the functionality. When an application is started, it may not need all of its objects to be available immediately. In such cases, the Virtual Proxy pattern suggests deferring object creation until it is needed by the application. The object is created the first time it is referenced in the application and the same instance is reused from that point onwards. This approach has advantages and disadvantages.

Advantage

The advantage of this approach is a faster application start-up time, as it is not required to create and load all of the application objects.

Disadvantage

Because there is no guarantee that a given application object is created, *everywhere* the application object is accessed it needs to be checked to make sure that it is not null, i.e., the object is created. The time penalty associated with this check is the main disadvantage.

Applying the Virtual Proxy pattern, a separate object referred to as a *virtual proxy* can be designed with its interface the same as that of the actual object. Different client objects can create an instance of the corresponding virtual proxy and use it in place of the actual object. The `Virtual Proxy` object maintains a reference to the actual object as one of its instance variables. The proxy does not automatically create the actual object. When a client invokes a method on the `Virtual Proxy` object that requires the services of the actual object, it checks to see if the actual object is created.

- If the actual object is already created, the proxy forwards the method call to the actual object.
- If the actual object is not already created:
  - It creates the actual object.
  - It assigns the object to its object reference variable.
  - It forwards the call to the actual object.

With this arrangement, details such as the existence of the actual object and the method forwarding are hidden from client objects. Client objects interact only with the `Proxy` object as if it is the actual object. As a result, client objects are free from checking if the actual object is null. Also, because the time and the processing overhead is less to create a virtual proxy than the actual object it is associated with, the virtual proxy can be instantiated at the beginning of a client application in place of the actual object.

EXAMPLE

Suppose that you are creating an IDE (Integrated Development Environment) for editing Java programs with features to compile, execute and generate javadocs. Most often when a Java program is created or edited, it is compiled and run, but javadocs may not be generated for every Java program. Hence, instead of creating and loading all the application objects that provide the entire IDE functionality, it might be a good idea to create only those objects that are required for editing, compiling and executing programs, leaving the other objects that offer the service of generating javadocs. This type of object creation strategy results in an efficient memory usage model and the IDE application can be started quickly as there is no need to load all of the application objects.

Let us suppose that the compile, run and javadoc generation functionalities are offered by three utility classes — `Compiler`, `Runtime` and `JavaDoc` — respectively. The interface for different IDE operations to be accessed by client objects can be designed in the form of an abstract `IDEOperation` class.

```
public abstract class IDEOperation {
  private Compiler cmp;
  private Runtime rtime;
  public void compile(String javaFile) {
    cmp.compile(javaFile);
  }
  public void run(String classFile) {
    rtime.run (classFile);
  }
  //to be delayed until needed.
  public abstract void generateDocs(String javaFile);
  public IDEOperation() {
    cmp = new Compiler();
    rtime = new Runtime();
```

```
  }
}
```

The IDEOperation class provides implementation for methods to compile and run Java programs. As part of its constructor, the IDEOperation creates and loads Compiler and Runtime objects required for the compile and execute operations. The javadoc generation method generateDocs is designed as an abstract method to be implemented by its subclasses.

Let us define a concrete subclass RealProcessor of the abstract IDEOperation class. The RealProcessor, as part of its constructor, creates a JavaDoc object that offers the javadoc generation service and implements the generateDocs method by using the JavaDoc object functionality.

```
public class RealProcessor extends IDEOperation {
  JavaDoc jdoc;
  public RealProcessor() {
    super();
    jdoc = new JavaDoc();
  }
  public void generateDocs(String javaFile) {
    jdoc.generateDocs(javaFile);
  }
}
```

With this implementation, the RealProcessor contains all the functionality to compile, run and generate javadocs for any Java program and can be readily used by client objects. As discussed earlier, however, the javadoc generation functionality may not be required for every Java program and the set of objects created when the RealProcessor is instantiated includes a JavaDoc object that is responsible for the javadoc generation. Creation of the JavaDoc objects can be deferred with the following advantages:

- Faster creation time of a RealProcessor object, as it needs to create fewer objects as part of its constructor.
- Efficient memory usage, as there is no need to hold an object in memory when there may be no need for its services.

Without altering the RealProcessor implementation, this can be accomplished by defining another subclass ProxyProcessor of the IDEOperation class. Because both the RealProcessor and the ProxyProcessor share the same interface, client objects can use the ProxyProcessor in place of the RealProcessor. Figure 25.1 shows the resulting class hierarchy.

```
public class ProxyProcessor extends IDEOperation {
  private RealProcessor realProcessor;
  public void generateDocs(String javaFile) {
```

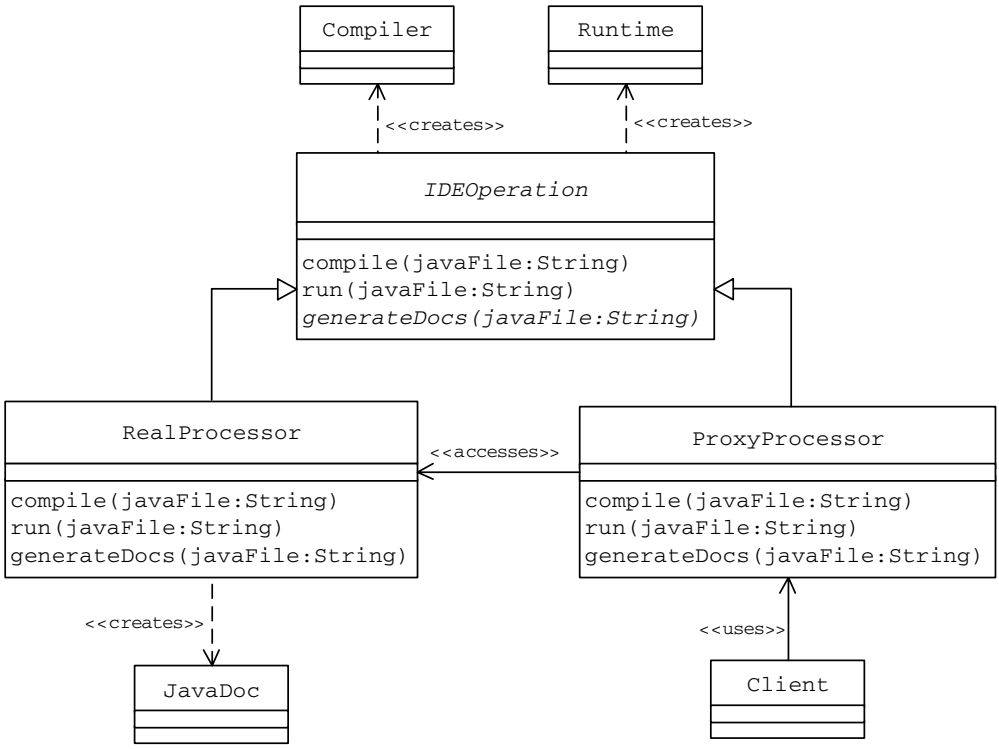

Figure 25.1 `IDEOperation` Class Hierarchy

```
        /*
          In order to generate javadocs
          the proxy loads the actual object and
          invokes its methods.
          */
        if (realProcessor == null) {
          realProcessor = new RealProcessor();
        }
          realProcessor.generateDocs(javaFile);
      }
    }
```

The `ProxyProcessor` maintains an object reference of the `RealProcessor` type as one of its instance variables. As part of the `generateDocs` method, the `ProxyProcessor` checks to see if this reference variable has been initialized with a `RealProcessor` object. If not, it creates a `RealProcessor` object and assigns it to the object reference instance variable. Once the `RealProcessor` object has been created, it invokes the `generateDocs` method on it.

In effect, it means that the `RealProcessor` is instantiated and loaded into the memory the first time when the javadoc generation functionality is requested

by a client object. In turn, this means that the JavaDoc object is not created and loaded into the memory until a client needs to generate javadocs for a Java program.

Client objects do not need to know the existence of the RealProcessor and can invoke methods on the ProxyProcessor as if it is the real processor. Details such as the validations and checks involved and the communication between the ProxyProcessor and the RealProcessor are completely hidden from client objects.

```
public class Client {
  public static void main(String[] args) {
  /*
    At this point objects required for
    the compile and run operations are
    created, but not the objects that provide the
    generate Javadoc functionality.
  */
    IDEOperation IDE = new ProxyProcessor();
    IDE.compile("test.java");
    IDE.run("test.class");
  /*
    The Javadoc functionality is accessed
    For the first time and hence the
    Object offering the Javadoc generation
    Functionality is loaded at this point.
  */
    IDE.generateDocs("test.java");
  }
}
```

PRACTICE QUESTIONS

1. Consider an application that uses a DBManager class, which encapsulates all of the database access details. As soon as the application is run, the DBManager may not be needed. Because creating a database connection is considered as an expensive operation, it might be a good idea to defer the instantiation of the DBManager class until the application needs to access the database for the first time. Design a virtual proxy for the DBManager class, which allows the postponement of the DBManager object creation, at the same time hiding such details from client objects.
2. Identify how the virtual proxy is involved in the following examples:
   - When a word processor such as Microsoft® Word is installed, it does not automatically create the index for help topics. When the help is accessed

for the first time, it builds the help topics index (in the case of MS Word, it clearly displays a message to this effect).

- Consider an application that uses JavaServer Pages™ technology. JSP scripts are not compiled automatically when they are placed in an application server specified directory. A JSP script is compiled the first time it is accessed.