10

FACTORY METHOD

This pattern was previously described in GoF95.

DESCRIPTION

In general, all subclasses in a class hierarchy inherit the methods implemented by the parent class. A subclass may override the parent class implementation to offer a different type of functionality for the same method. When an application object is aware of the exact functionality it needs, it can directly instantiate the class from the class hierarchy that offers the required functionality.

At times, an application object may only know that it needs to access a class from within the class hierarchy, but does not know exactly which class from among the set of subclasses of the parent class is to be selected. The choice of an appropriate class may depend on factors such as:

- The state of the running application
- Application configuration settings
- Expansion of requirements or enhancements

In such cases, an application object needs to implement the class selection criteria to instantiate an appropriate class from the hierarchy to access its services (Figure 10.1).

This type of design has the following disadvantages:

- Because every application object that intends to use the services offered by the class hierarchy needs to implement the class selection criteria, it results in a high degree of coupling between an application object and the service provider class hierarchy.
- Whenever the class selection criteria change, every application object that uses the class hierarchy must undergo a corresponding change.
- Because class selection criteria needs to take all the factors that could affect the selection process into account, the implementation of an application object could contain inelegant conditional statements.

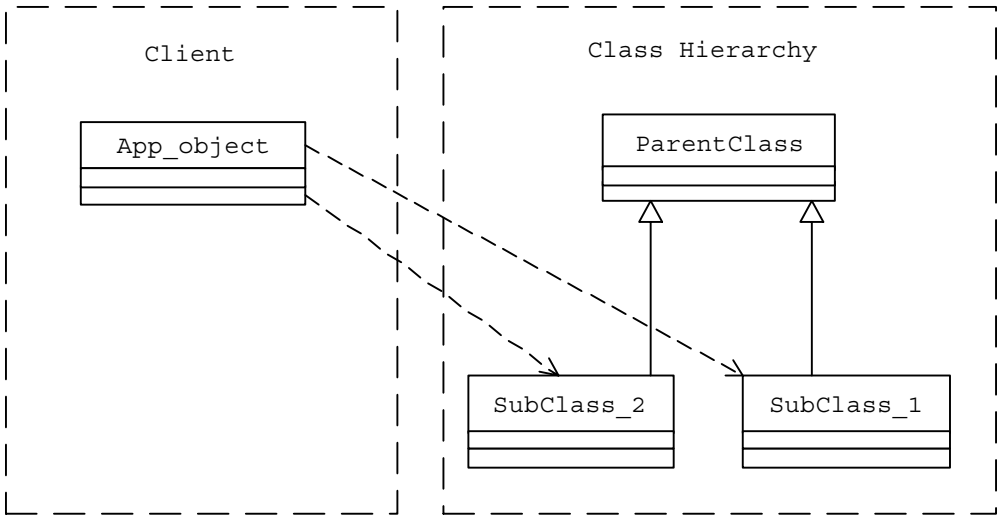

Figure 10.1 Client Object Directly Accessing a Service Provider Class Hierarchy

- If different classes in the class hierarchy need to be instantiated in diverse manners, the implementation of an application object can become more complex.
- It requires an application object to be fully aware of the existence and the functionality offered by each class in the service provider class hierarchy.

In such cases, the Factory Method pattern recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a *factory method*. Thus, a factory method can be defined as a method in a class that:

- Selects an appropriate class from a class hierarchy based on the application context and other influencing factors
- Instantiates the selected class and returns it as an instance of the parent class type

Encapsulation of the required implementation to select and instantiate an appropriate class in a separate method has the following advantages:

- Application objects can make use of the factory method to get access to the appropriate class instance. This eliminates the need for an application object to deal with the varying class selection criteria.
- Besides the class selection criteria, the factory method also implements any special mechanisms required to instantiate the selected class. This is applicable if different classes in the hierarchy need to be instantiated in different ways. The factory method hides these details from application objects and eliminates the need for them to deal with these intricacies.

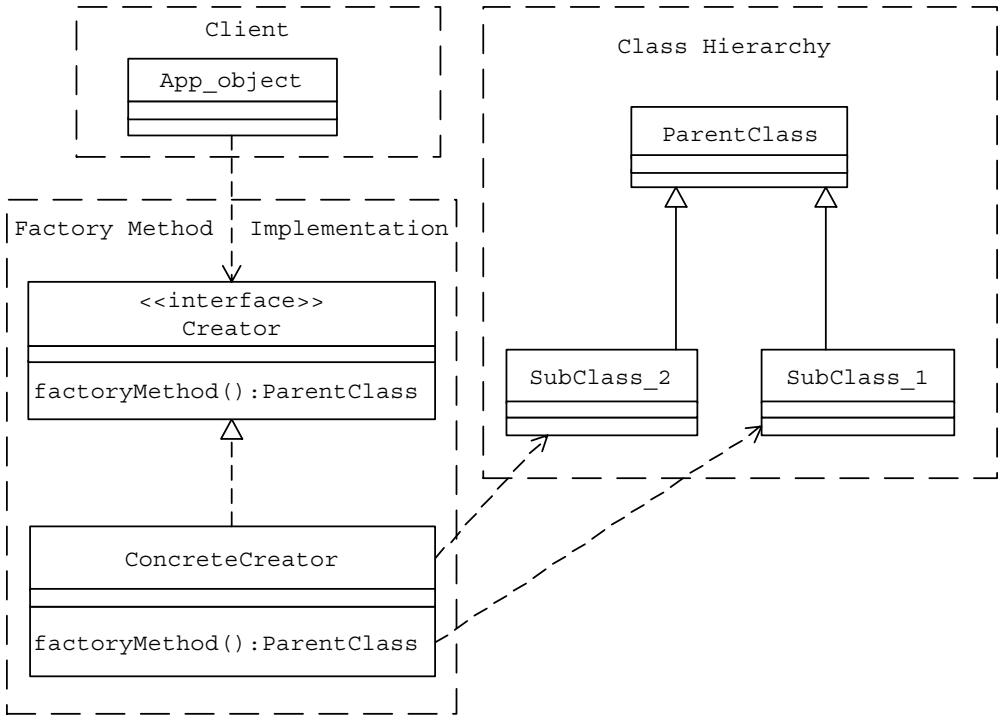

Figure 10.2 A Client Object Accessing a Service Provider Class Hierarchy Using a Factory Method

- Because the factory method returns the selected class instance as an object of the parent class type, an application object does not have to be aware of the existence of the classes in the hierarchy.

One of the simplest ways of designing a factory method is to create an abstract class or an interface that *just declares* the factory method. Different subclasses (or implementer classes in the case of an interface) can be designed to implement the factory method in its entirety as depicted in Figure 10.2. Another strategy is to create a concrete creator class with default implementation for the factory method in it. Different subclasses of this concrete class can override the factory method to implement specialized class selection criteria.

EXAMPLE

Let us design the functionality to log messages in an application. In general, logging messages is one of the most commonly performed tasks in software applications. Logging appropriate messages at appropriate stages can be extremely useful for debugging and monitoring applications.

Because the message logging functionality could be needed by many different clients, it would be a good idea to keep the actual message logging functionality inside a common utility class so that client objects do not have to repeat these details.

Let us define a Java interface `Logger` (Listing 10.1) that declares the interface to be used by the client objects to log messages.

In general, an incoming message could be logged to different output media, in different formats. Different concrete implementer classes of the `Logger` interface can handle these differences in implementation. Let us define two such implementers as in Table 10.1. The resulting class hierarchy is depicted in Figure 10.3.

Each of the `Logger` implementer classes (Listing 10.2) offers the respective functionality stated in Table 10.1 inside the `log` method declared by the `Logger` interface.

Consider an application object `LoggerTest` that intends to use the services provided by the `Logger` implementers. Suppose that the overall application message logging configuration can be specified using the `logger.properties` property file.

Listing 10.1 Logger Interface

```
public interface Logger {
  public void log(String msg);
}
```

Table 10.1 Logger Implementers

Implementer	*Functionality*
`FileLogger`	Stores incoming messages to a log file
`ConsoleLogger`	Displays incoming messages on the screen

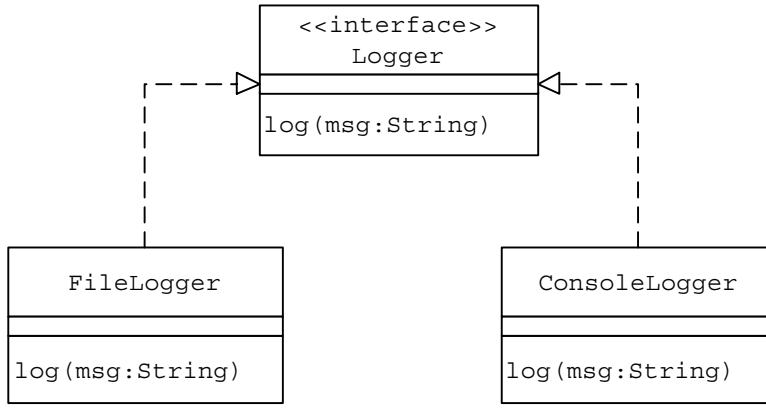

Figure 10.3 Message Logging Utility Class Hierarchy

Listing 10.2 Logger Implementer Classes

```
public class FileLogger implements Logger {
  public void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt", msg, true, true);
  }
}
public class ConsoleLogger implements Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}
```

Sample `logger.properties` file contents

```
FileLogging=OFF
```

Depending on the value of the `FileLogging` property, an appropriate `Logger` implementer needs to be used to log messages. For example, if the `FileLogging` property is set to `ON`, messages are to be logged to a file and hence a `FileLogger` object can be used to log messages. Similarly, if the `FileLogging` property is set to `OFF`, messages are to be displayed on the console and hence a `ConsoleLogger` object can be used.

To log messages, an application object such as the `LoggerTest` needs to:

- Identify an appropriate `Logger` implementer by reading the `FileLogging` property value from the `logger.properties` file
- Instantiate the `Logger` implementer and invoke the `log` method by passing the message text to be logged as an argument

This requires every application object to:

- Be aware of the existence and the functionality of all implementers of the `Logger` interface and their subclasses
- Provide the implementation required to select and instantiate an appropriate `Logger` implementer

Figure 10.4 depicts this design strategy.

Applying the Factory Method pattern, the necessary implementation for selecting and instantiating an appropriate `Logger` implementer can be encapsulated inside a separate `getLogger` method in a separate class `LoggerFactory`

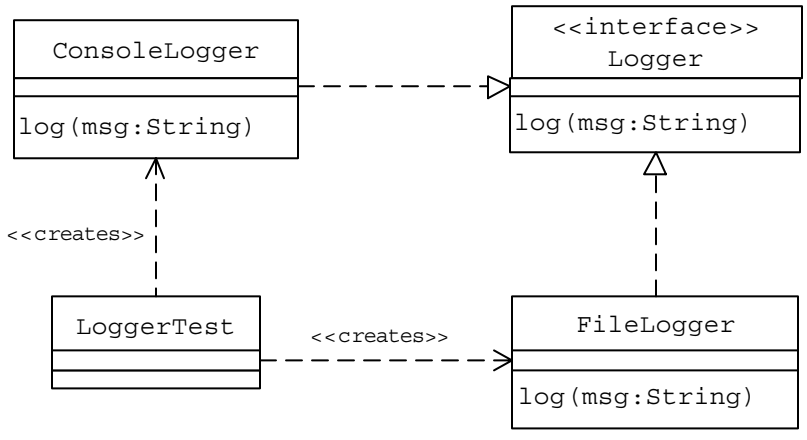

Figure 10.4 Client `LoggerTest` Accessing Logger Implementers Directly

(Listing 10.3). The `LoggerFactory`, with the `getLogger` factory method, plays the role of the `ConcreteCreator` shown in Figure 10.2.

As part of its implementation, the factory method `getLogger` checks the `logger.properties` property file to see if file logging is enabled and decides which `Logger` implementation is to be instantiated. The selected `Logger` implementer instance is returned as an object of type `Logger`.

With the factory method in place, client objects do not need to deal with the intricacies involved in selecting and instantiating an appropriate `Logger` implementer. Client objects do not need to know the existence of different implementers of the `Logger` interface and their associated functionality (Figure 10.5).

Whenever a client object such as the `LoggerTest` (Listing 10.4) needs to log a message, it can:

- Invoke the factory method `getLogger`. When the factory method returns, the client object does not have to know the exact `Logger` subtype that is instantiated as long as the returned object is of the `Logger` type.
- Invoke the `log` method exposed by the `Logger` interface on the returned object.

Figure 10.6 shows the message flow when the client object `LoggerTest` uses the `LoggerFactory` factory method to create an appropriate `Logger` implementer to log a message.

In this example application design, the creator class `LoggerFactory` is designed as a concrete class with default implementation for the factory method `getLogger`. There can be variations in the way in which the class selection criterion is implemented. Such variations can be implemented by overriding the `getLogger` method in `LoggerFactory` subclasses.

Listing 10.3 `LoggerFactory` Class

```
public class LoggerFactory {
  public boolean isFileLoggingEnabled() {
    Properties p = new Properties();
    try {
      p.load(ClassLoader.getSystemResourceAsStream(
        "Logger.properties"));
      String fileLoggingValue =
        p.getProperty("FileLogging");
      if (fileLoggingValue.equalsIgnoreCase("ON") == true)
        return true;
      else
        return false;
    } catch (IOException e) {
      return false;
    }
  }
  //Factory Method
  public Logger getLogger() {
    if (isFileLoggingEnabled()) {
      return new FileLogger();
    } else {
      return new ConsoleLogger();
    }
  }
}
```

PRACTICE QUESTIONS

1. Add a new logger `DBLogger` that logs messages to a database.
2. Create a subclass of the `LoggerFactory` class and override the `getLogger` implementation to implement a different class selection criterion.

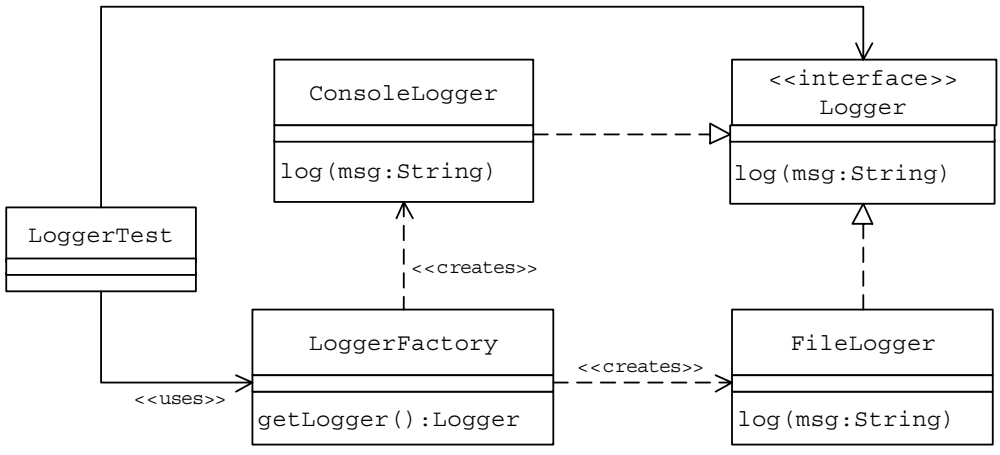

Figure 10.5 The Client `LoggerTest` Accessing the Logger Class Hierarchy after the Factory Method Pattern Is Applied

Listing 10.4 Client `LoggerTest` Class

```
public class LoggerTest {
  public static void main(String[] args) {
    LoggerFactory factory = new LoggerFactory();
    Logger logger = factory.getLogger();
    logger.log("A Message to Log");
  }
}
```

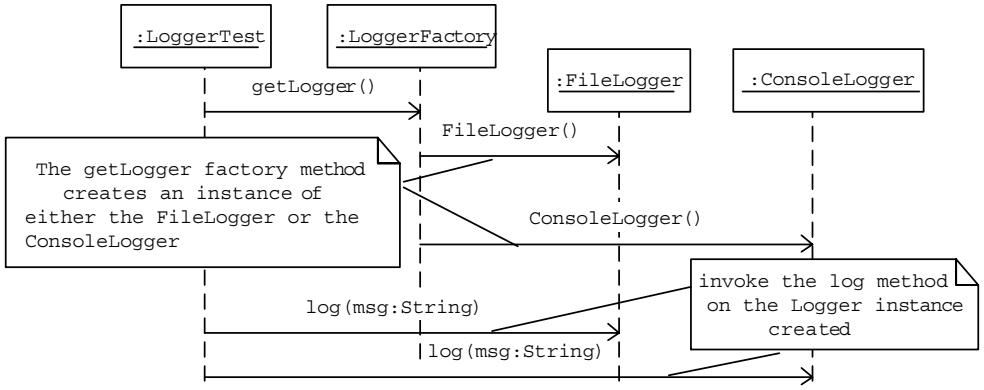

Figure 10.6 Message Flow When a Client Uses the `LoggerFactory` to Create an Appropriate Logger to Log a Message