24

BRIDGE

This pattern was previously described in GoF95.

DESCRIPTION

The Bridge pattern promotes the separation of an abstraction's interface from its implementation. In general, the term *abstraction* refers to the process of identifying the set of attributes and behavior of an object that is specific to a particular usage. This specific view of an object can be designed as a separate object omitting irrelevant attributes and behavior. The resulting object itself can be referred to as an *abstraction*. Note that a given object can have more than one associated abstraction, each with a distinct usage.

A given abstraction may have one or more implementations for its methods (behavior). In terms of implementation, an abstraction can be designed as an interface with one or more concrete implementers (Figure 24.1).

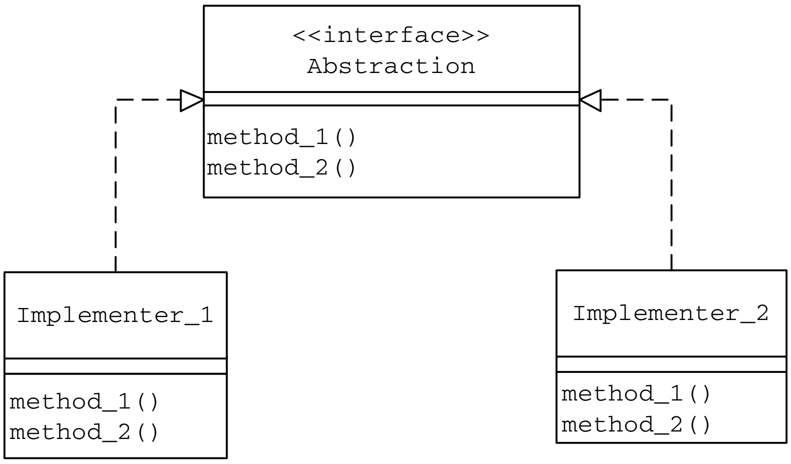

Figure 24.1 Abstraction as an Interface with a Set of Concrete Implementers

In the class hierarchy shown in Figure 24.1, the `Abstraction` interface declares a set of methods that represent the result of abstracting common features from different objects. Both `Implementer_1` and `Implementer_2` represent the set of `Abstraction` implementers. This approach suffers from the following two limitations:

1. When there is a need to subclass the hierarchy for some other reason, it could lead to an exponential number of subclasses and soon we will have an exploding class hierarchy.
2. Both the `abstraction` interface and its implementation are closely tied together and hence they cannot be independently varied without affecting each other.

Using the Bridge pattern, a more efficient and manageable design of an abstraction can be achieved. The design of an abstraction using the Bridge pattern separates its interfaces from implementations. Applying the Bridge pattern, both the interfaces and the implementations of an abstraction can be put into separate class hierarchies as in Figure 24.2.

From the class diagram in Figure 24.2, it can be seen that the Abstraction maintains an object reference of the `Implementer` type. A client application can

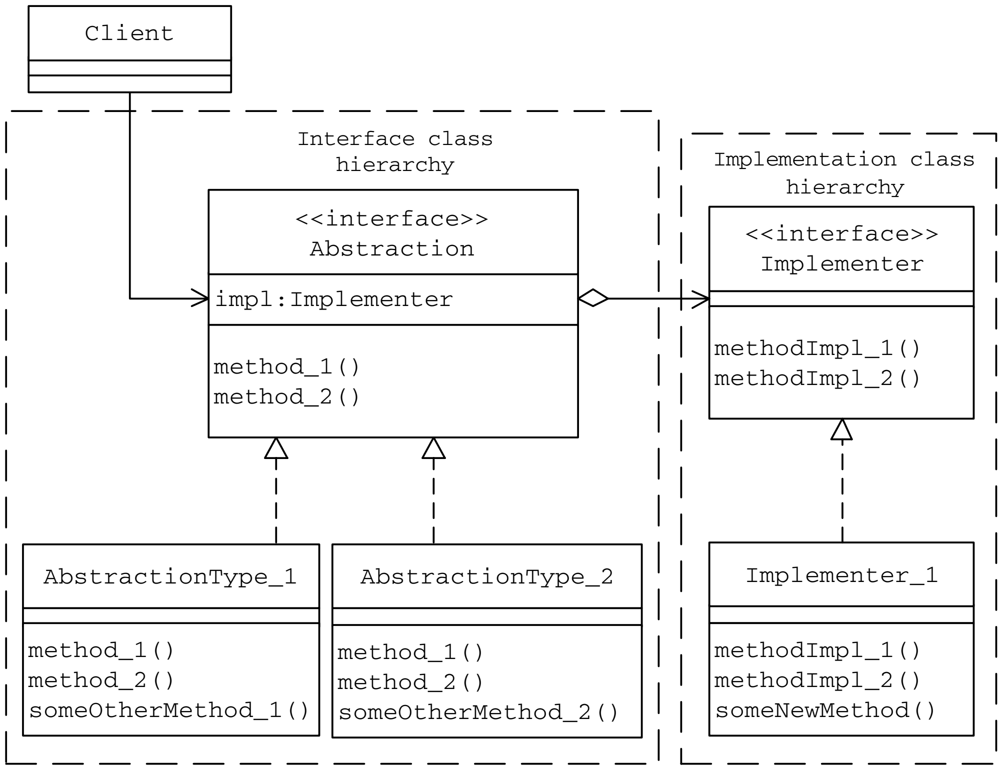

Figure 24.2 Interface and Implementations in Two Separate Class Hierarchies

choose a desired abstraction type from the `Abstraction` class hierarchy. The abstraction object can then be configured with an instance of an appropriate implementer from the `Implementer` class hierarchy. This ability to combine abstractions and implementations dynamically can be very useful in terms of extending the functionality without subclassing. When a client object invokes a method on the `Abstraction` object, it forwards the call to the `Implementer` object it contains. The `Abstraction` object may offer some amount of processing before forwarding the call to the `Implementer` object.

This type of class arrangement completely decouples the interface and the implementation of an abstraction and allows the classes in the interface and the implementation hierarchy to vary without affecting each other.

EXAMPLE

We designed the message logging functionality for an application during the discussion of the Factory Method pattern. Logging can be used for various purposes at different stages of an application and hence many different objects that are part of the application may need to have the ability to log messages. Because many different objects within an application may need the ability to log messages, the logging feature may be put into a separate class. The resulting class is an abstraction of the message logging functionality. From here on, we use the phrase *logger abstraction* to refer to the abstraction of the message logging functionality.

A message can be logged to different types of destinations such as a file, console and others. Depending on the destination type, a different implementation of the logger abstraction is needed. This requirement can be designed with a common `Logger` interface that declares the interface (methods) of the abstraction and different implementers corresponding to different destination types provide implementation for the logger abstraction. Let us define two such implementers — `FileLogger` and `ConsoleLogger` — to log messages to a file and console, respectively. Figure 24.3 depicts the resulting class hierarchy.

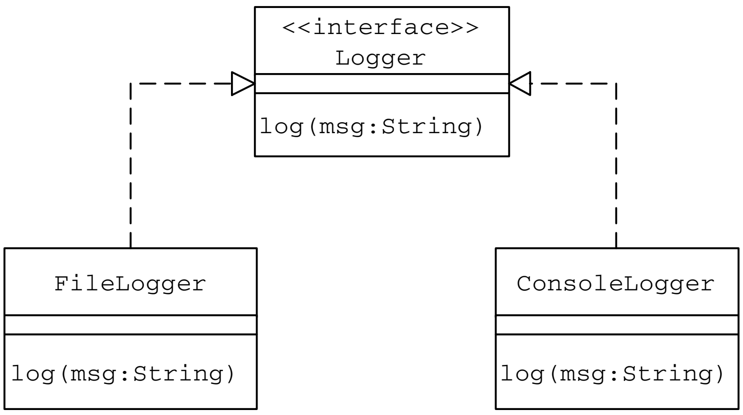

Figure 24.3 Logger Abstraction before Applying the Bridge Pattern

Different client objects can use one of the implementer (`FileLogger` or `ConsoleLogger`) objects to log messages to a desired destination in plain text format. After this design is implemented, let us suppose that an application object needs to log messages in a different format (e.g., in an encrypted form). The existing messaging logging functionality design is not sufficient without either:

- Modifying different implementers
- Extending the entire class hierarchy

Having to modify the existing code in order to extend the functionality is not advisable and violates the basic object-oriented open-closed principle.

The open-closed principle states that a software module should be:

- *Open for extension* — It should be possible to alter the behavior of a module or add new features to the module functionality.
- *Closed for modification* — Such a module should not allow its code to be modified.

In a nutshell, the open-closed principle helps in designing software modules whose functionality can be extended without having to modify the existing code.

This also means that whenever there is a change to be made to the `Logger` (Java)interface for a different type of (application)interface, each of its implementations needs to be modified, making the logger abstraction interface and its implementation dependent on each other.

Subclassing the class hierarchy for every different type of message format is also not recommended as it could result in an exponential number of subclasses and soon there will be an exploding class hierarchy. The Bridge pattern can be used in this case to provide the ability to add new message formats and new types of implementations to the logger abstraction. The Bridge pattern separates the interface and implementations into two separate class hierarchies so that they both can be modified without affecting each other.

Applying the Bridge pattern the interface and the implementation of the logger abstraction can be arranged into two separate class hierarchies.

Abstraction Implementation Design

Implementers of the logger abstraction need to provide the actual implementation required to log messages to different destination types. Let us define two such implementers — `FileLogger` and `ConsoleLogger` — to log messages to a file and console, respectively. These abstraction implementers can be designed as two concrete implementers of a common `MessageLogger` (Java)interface (Listing 24.1).

The common `MessageLogger` interface declares a method `logMsg(String msg)`, which can be used by objects that represent the interface of the logger abstraction. Figure 24.4 depicts the resulting abstraction implementation class hierarchy.

As part of its implementation of the `logMsg` method (Listing 24.2):

Listing 24.1 `MessageLogger` Interface

```
public interface MessageLogger {
  public void logMsg(String msg);
}
```

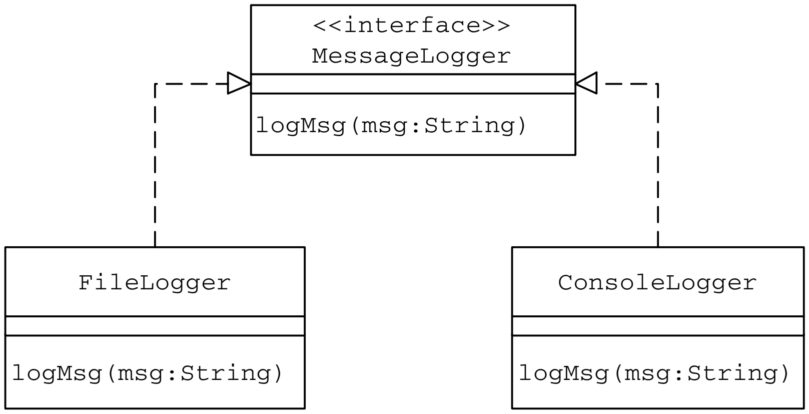

Figure 24.4 Logger Abstraction: Implementer Hierarchy

Listing 24.2 `MessageLogger` Implementers: `FileLogger` and `ConsoleLogger`

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

- The `FileLogger` writes a given message to a log file using a helper `FileUtil` class.
- The `ConsoleLogger` writes a given message on the screen.

Note that client objects should not directly access the message logging service offered by different logger abstraction implementers. To log a message, different

Listing 24.3 Message Interface

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

client objects interact with instances of the classes representing the logger abstraction interface. These abstraction interface objects in turn use the services of the abstraction implementer classes.

Abstraction Interface Design

The interface for the logger abstraction can be designed in the form of a set of classes representing different types of messages that a client object would like to log. These classes can be designed as implementers of a common `Message` (Java)interface (Listing 24.3).

The `Message` interface declares a method `log(String msg)`, which can be used by different client objects to log messages.

Let us define two logger abstraction interface classes — `TextMessage` and `EncryptedMessage` — (Listing 24.4) representing a plain text message and an encrypted message, respectively. These abstraction interface classes can be designed as concrete implementers of a common `Message` (Java)interface.

Figure 24.5 shows the resulting logger abstraction interface class hierarchy.

Design Highlights of the Abstraction Interface Classes

- Logger abstraction interface classes — `TextMessage` and `Encrypted-Message` — do not provide implementation for the actual message logging service. As seen earlier, classes such as the `FileLogger` and `Console-Logger` in the abstraction implementer class hierarchy provide the actual message logging implementation.
- Client objects do not directly use the interface exposed by the abstraction implementer classes.
- Each abstraction interface class maintains an object reference of the `Mes-sageLogger` (abstraction implementer) type. Whenever a client object creates an abstraction interface object, it configures the interface object with a `MessageLogger` object.
- Whenever a client object invokes the log method on an abstraction interface object, the interface object does any required preprocessing and uses the message logging service of the `MessageLogger` object it contains.
- The preprocessing functionality is meant to be used internally by abstraction interface objects only and it should not be available to client objects. To ensure this, the `preProcess` method in both the `TextMessage` and `EncryptedMessage` abstraction interface classes is designed as a private method. As part of its implementation of the `preProcess` method, the

Listing 24.4 Message Implementers: `TextMessage` and `EncryptedMessage`

```
public class TextMessage implements Message {
  private MessageLogger logger;
  public TextMessage(MessageLogger l) {
    logger = l;
  }
  public void log(String msg) {
    String str = preProcess(msg);
    logger.logMsg(str);
  }
  private String preProcess(String msg) {
    return msg;
  };
}
public class EncryptedMessage implements Message {
  private MessageLogger logger;
  public EncryptedMessage(MessageLogger l) {
    logger = l;
  }
  public void log(String msg) {
    String str = preProcess(msg);
    logger.logMsg(str);
  }
  private String preProcess(String msg) {
    msg = msg.substring(msg.length() - 1) +
          msg.substring(0, msg.length() - 1);
    return msg;
  };
}
```

`EncryptedMessage` encrypts an incoming message by shifting all characters to the right by one position.

As a result of keeping the interface and the implementation of the logger abstraction in two separate class hierarchies, the interfaces and the implementations of the logger abstraction are completely decoupled.

Whenever a client (Listing 24.5) needs to log a message:

1. It creates an instance of an appropriate `MessageLogger` implementer class such as `FileLogger` or `ConsoleLogger`.

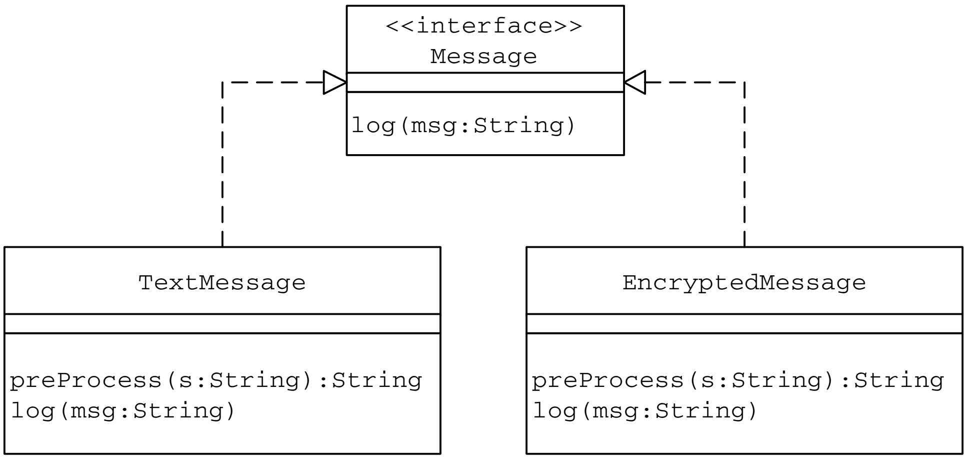

Figure 24.5 Logger Abstraction: Interface Hierarchy

Listing 24.5 Test Client Class

```
public class Client {
  public static void main(String[] args) {
    //Create an appropriate implementer object
    MessageLogger logger = new FileLogger();
    //Choose required interface object and
    //configure it with the implementer object
    Message msg = new EncryptedMessage(logger);
    msg.log("Test Message");
  }
}
```

2. It creates an instance of an appropriate `Message` implementer class such as `TextMessage` or `EncryptedMessage`.
3. It configures the `Message` implementer object with the `MessageLogger` implementer object created in Step 1. This object is maintained inside the `Message` implementer object.
4. It calls the `log(String)` method on the `Message` implementer object created in Step 2.
5. The `Message` implementer object carries out the required processing to transform the incoming message to the desired format (encrypt the input message in the case of the `EncryptedMessage`) and forwards the transformed message to the `MessageLogger` implementer object it contains by invoking its `logMessage(String)` method. This relationship between classes in the interface and the implementer class hierarchy can be viewed as a Bridge in this case.

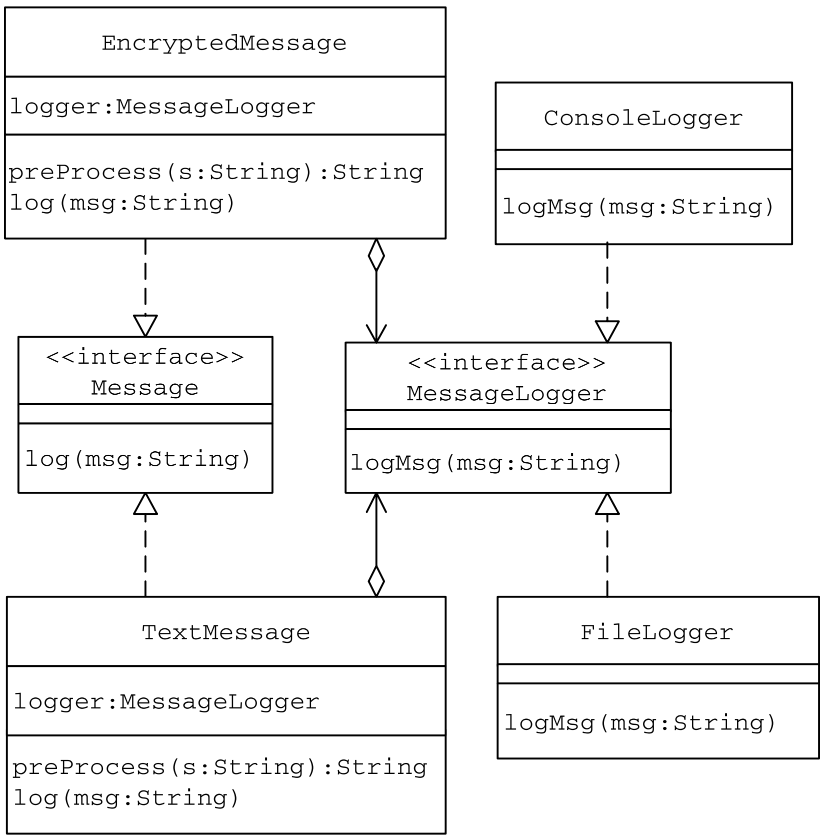

Figure 24.6 Logger Abstraction after Applying the Bridge Pattern

Figure 24.6 shows the overall class association.

The message flow when an application object uses the logger abstraction to log an encrypted message can be depicted as in Figure 24.7.

The separation of the logger abstraction interface from its implementation allows them to be modified independently without having to modify the other.

After the design is implemented, if a client object needs to log messages in a new format, say HTML, this requirement can be addressed easily by designing a new logger abstraction interface class `HTMLMessage` as an implementer of the existing `Message` interface. The `HTMLMessage` class can be designed to provide the required processing to transform an incoming message to HTML text and use an abstraction implementer class to actually log the transformed message. This addition of a new interface class does not affect any existing abstraction implementers. In addition, adding a new class for every new type of message format keeps the class growth linear.

Similarly, a new logger abstraction implementation such as a `DBLogger` to log messages to a database can be added without having to modify or subclass the class hierarchy.

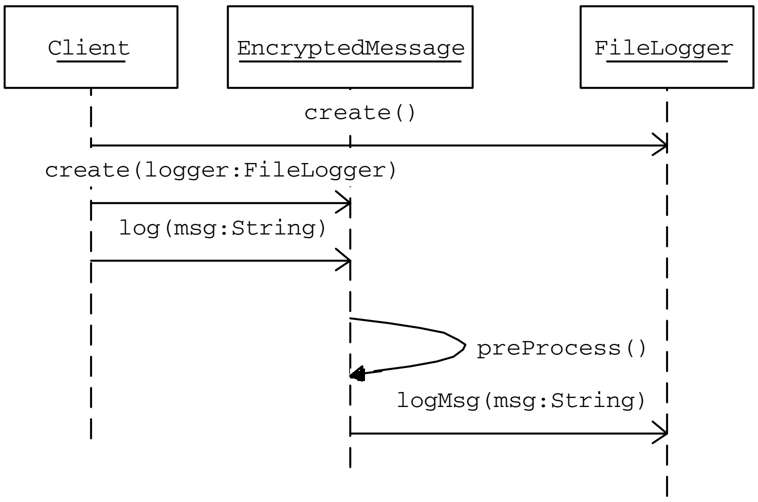

Figure 24.7 Message Flow When an Application Logs an Encrypted Message

BRIDGE PATTERN VERSUS ADAPTER PATTERN

Similarities:

- Both the Adapter pattern and the Bridge pattern are similar in that they both work towards concealing the details of the underlying implementation from the client.

Differences:

- The Adapter pattern aims at making classes work together that could not otherwise because of incompatible interfaces. An Adapter is meant to change the interface of an *existing object*. As we have seen during our discussion on the Adapter pattern, an Adapter requires an (existing) adaptee class, indicating that the Adapter pattern is more suitable for needs after the initial system design.
- The Bridge pattern is more of a design time pattern. It is used when the designer has control over the classes in the system. It is applied before a system has been implemented to allow both abstraction interfaces and its implementations to be varied independently without affecting each other.
- In the context of the Bridge pattern, the issue of incompatible interfaces does not exist. Client objects always use the interface exposed by the abstraction interface classes. Thus both the Bridge pattern and the Adapter pattern are used to solve different design issues.

PRACTICE QUESTIONS

1. Design an application that reads and writes different types of data (plain text, binary, etc.) to and from different destinations such as a file, a URL or a database. Apply the Bridge pattern in designing the data read/write abstraction.

2. Many applications with a database backend use ODBC/JDBC drivers from different vendors. Identify how the Bridge pattern is applied when an application uses an ODBC/JDBC driver.
3. Design a code formatting application using the Bridge pattern. In general, programs can be written in any computer language (e.g., Java, VB, etc.) and a given program can be formatted in different ways such as simple text formatting, HTML formatting, color formatting and others. In effect, the interface for code formatting can be implemented in many different ways. Apply the Bridge pattern to separate the interface from its implementations.