Principles, Patterns, and Practices: The Strategy, Template Method, and Bridge Patterns

by Robert C. Martin
10/29/2004

One of the great benefits of object-oriented programming is polymorphism; i.e., the ability to send a message to an object without knowing the true type of the object. Perhaps no pattern illustrates this better than the Strategy pattern.

To illustrate the Strategy pattern let's assume that we are working on a debug logger. Debug loggers are often very useful devices. Programmers can send messages to these loggers at strategic places within the code. If the system misbehaves in some way, the debug log can provide clues about what the system was doing internally at the time of the failure.

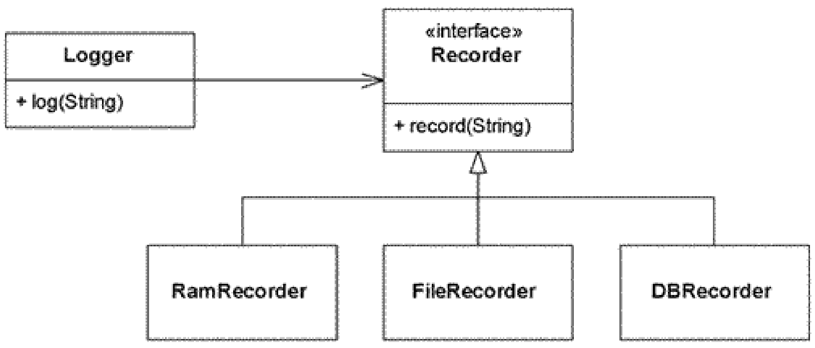
In order to be effective, loggers need to be simple for programmers to use. Programmers aren't going to frequently use something that is inconvenient. You should be able to emit a log message with something no more complicated than:

```
logger.log("My Message");
```

On the other hand, what we want to see in the log is quite a bit more complex. At very least we are going to want to see the time and date of the message. We'll also probably want to see the thread ID. Indeed, there may be a whole laundry list of system states that we want to log along with the message. So the logger needs to gather all of this peripheral information together, format it into a log message, and then add it to the growing list of logged messages.

Where should the logged messages be stored? Sometimes we might like them stored in a text file. Sometimes we might want to see them added to a database table. Sometimes we might like them accumulated in RAM. The choices seem endless. However, the final destination of the logged messages has nothing to do with the format of the messages themselves.

We have two algorithms: one formats the logged message, and the other records the logged message. These two algorithms are both in the flow of logging a message, but both can vary independently of each other. The formatter does not care where the message is recorded, and the recorder does not care about the format of the message. Whenever we have two connected but independent algorithms, we can use the Strategy pattern to connect them. Consider the following structure:



Here the user calls the log method of the Logger class. The log method formats the message and then calls the record method of the Recorder interface. There are many possible implementations of the Recorder interface. Each does the recording in a different way.

The structure of the Logger and Recorder is exemplified by the following unit test, which uses the Adapter pattern:

```
public class LoggerTest extends TestCase {
  private String recordedMessage;
  protected String message;

  public void testLogger() throws Exception {
    Logger logger = new Logger(new Recorder() {
      public void record(String message) {
        recordedMessage = message;
      }
    });

    message = "myMessage";
    logger.log(message);
    checkFormat();
  }

  private void checkFormat() {
    String datePattern = "\\d{2}/\\d{2}/\\d{4}
\\d{2}:\\d{2}:\\d{2}.\\d{3}";
    String messagePattern = datePattern + " " + message;
    if(!Pattern.matches(messagePattern, recordedMessage)) {
      fail(recordedMessage + " does not match pattern");
    }
  }
}
```

As you can see, the Logger is constructed with an instance of an object that implements the Recorder interface. Logger does not care what that implementation does. It simply builds the string to be logged and then calls the record method. This is very powerful decoupling. It allows the formatting and recording algorithms to change independently of each other.

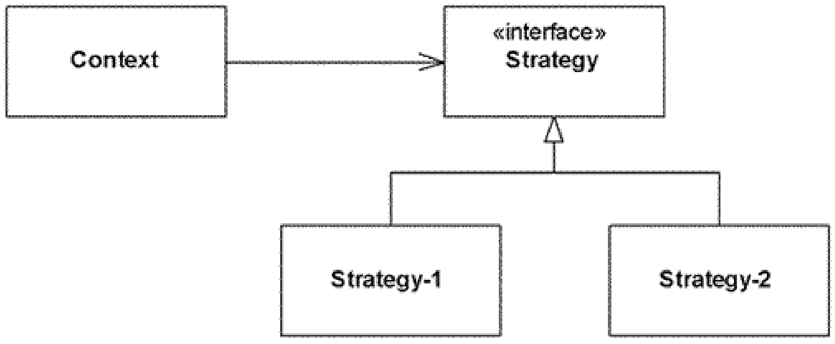
Logger is a simple class that simply formats the message and forwards it to the Recorder.

```
public class Logger {
  private Recorder recorder;

  public Logger(Recorder recorder) {
    this.recorder = recorder;
  }

  public void log(String message) {
    DateFormat format = new SimpleDateFormat("MM/dd/yyyy
kk:mm:ss.SSS");
    Date now = new Date();
    String prefix = format.format(now);
    recorder.record(prefix + " " + message);
  }
}
```

And Recorder is an even simpler interface.

```
public interface Recorder {
  void record(String message);
}
```

The canonical form of the Strategy pattern is shown below. One algorithm (the *context*) is shielded from the other (the *strategy*) by an interface. The context is unaware of how the strategy is implemented, or of how many different implementations there are. The context typically holds a pointer or reference to the strategy object with which it was constructed.

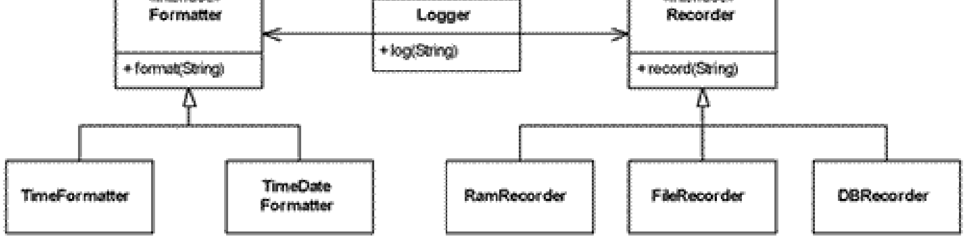


In our Logger example, the Logger is the context, the Recorder is the strategy interface, and the anonymous inner class within the unit test acts as one of the implemented strategies.

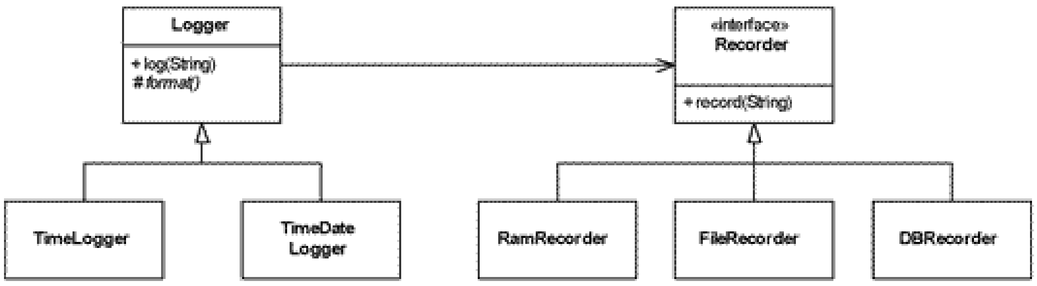
If you have been an object-oriented programmer for any length of time, you have seen this pattern many times. Indeed, it is so common that some folks shake their heads and wonder why it even has a name. It's rather like giving the name "DO NEXT STATEMENT" to the fact that execution proceeds statement by statement. However, there is a good reason to give this pattern a name. It turns out that there is another pattern that solves the same problem in a slightly different way; and the two names help us differentiate between them.

This second pattern is called *Template Method*, and we can see it by adding the next obvious layer of polymorphism to the Logger example. We already have one layer that allows us to change the way log messages are recorded. We could add another layer to allow us to change how log messages are formatted. Let's suppose, for instance, that we want to support two different formats. One prepends the time and date to the message as above; the other prepends only the time.

Clearly, this is just another problem in polymorphism, and we could use the Strategy pattern once again. If we did, the design might look like this:



Here we see two uses of the Strategy pattern. One provides polymorphic recording and the other provides polymorphic formatting. This is a common enough solution, but it is not the only solution. Indeed, we might have opted for a solution that looked more like this:



Notice the format method of Logger. It is protected (that's what the # means) and it is abstract (that's what the italics mean). The log method of Logger calls its own abstract format method, which deploys to one of the derived classes. The formatted string is then passed to the record method of the Recorder.

Consider the unit test below. It shows tests for both the TimeLogger and the TimeDateLogger. Notice that each test method creates the appropriate Logger derivative and passes a Recorder instance into it.

```
import junit.framework.TestCase;
import java.util.regex.Pattern;

public class LoggerTest extends TestCase {
  private String recordedMessage;
  protected String message;
  private static final String timeDateFormat =
    "\\d{2}/\\d{2}/\\d{4} \\d{2}:\\d{2}:\\d{2}.\\d{3}";
  private static final String timeFormat =
"\\d{2}:\\d{2}:\\d{2}.\\d{3}";
  private Recorder recorder = new Recorder() {
    public void record(String message) {
      recordedMessage = message;
    }
  }

public void testTimeDateLogger() throws Exception {
    Logger logger = new TimeDateLogger(recorder);
    message = "myMessage";
    logger.log(message);
    checkFormat(timeDateFormat);
  }
public void testTimeLogger() throws Exception {
    Logger logger = new TimeLogger(recorder);
    message = "myMessage";
    logger.log(message);
    checkFormat(timeFormat);
  }

private void checkFormat(String prefix) {
    String messagePattern = prefix + " " + message;
    if (!Pattern.matches(messagePattern, recordedMessage)) {
      fail(recordedMessage + " does not match pattern");
    }
  }
}
```

The Logger has changed as follows. Notice the protected abstract format method.

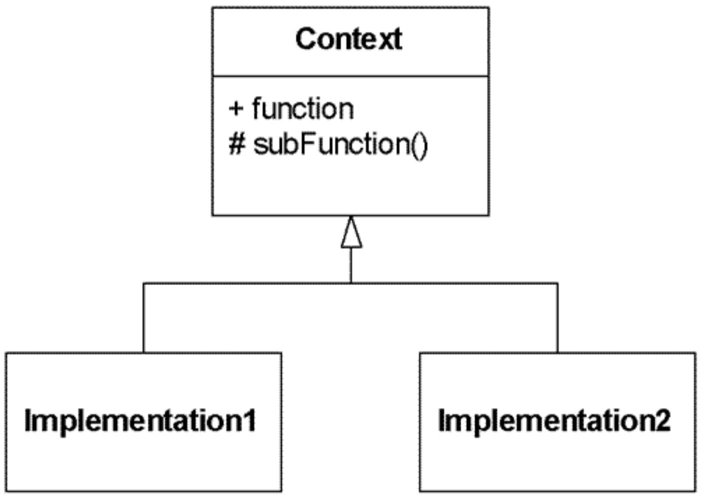
```
public abstract class Logger {
  private Recorder recorder;

  public Logger(Recorder recorder) {
    this.recorder = recorder;
  }

  public void log(String message) {
    recorder.record(format(message));
  }

  protected abstract String format(String message);
}
```

TimeLogger and TimeDateLogger simply implement the format method appropriate to their type, as shown below:

```
import java.text.*;
import java.util.Date;

public class TimeLogger extends Logger {
  public TimeLogger(Recorder recorder) {
    super(recorder);
  }

  protected String format(String message) {
    DateFormat format = new SimpleDateFormat("kk:mm:ss.SSS");
    Date now = new Date();
    String prefix = format.format(now);
    return prefix + " " + message;
  }
}
```

```
import java.text.*;
import java.util.Date;

public class TimeDateLogger extends Logger {
  public TimeDateLogger(Recorder recorder) {
    super(recorder);
  }

  protected String format(String message) {
    DateFormat format = new SimpleDateFormat("MM/dd/yyyy
kk:mm:ss.SSS");
    Date now = new Date();
    String prefix = format.format(now);
    return prefix + " " + message;
  }
}
```

The canonical form of Template Method looks like this:

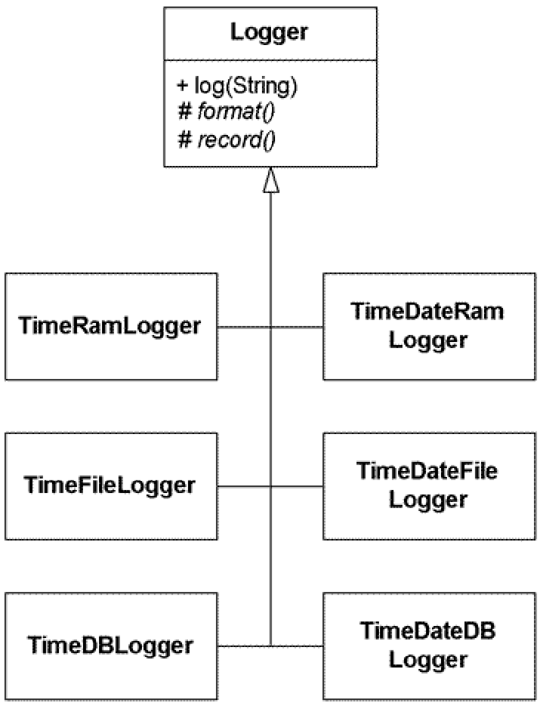


The Context class has at least two functions. One (here called function) is generally public, and represents some high-level algorithm. The other function (here called subFunction) represents some lower-level algorithm called by the higher-level algorithm. The derivatives of Context implement subFunction in different ways.

It should be clear how Strategy and Template Method solve the same problem. The problem is simply to separate a high-level algorithm from a lower-level algorithm in such a way that the two can vary independently. In the case of Strategy, this is solved by creating an interface for the lower-level algorithm. In the Template Method case, it is solved by creating an abstract method.

Strategy is preferable to Template Method when the lower-level algorithm needs to change at run time. This can be accomplished with Strategy simply by swapping in an instance of a different derivative. Template Method is not so fortunate; once created, its lower-level algorithm is locked in. On the other hand, Strategy has a slight time and space penalty compared to Template Method, and is more complex to set up. So Strategy should be used when flexibility is important, and Template Method should be used when time and space efficiency and simplicity are more important.

Could we have used Template Method to solve the whole Logger problem? Yes, but the result is not pleasant. Consider the following diagram.



Notice that there is one derivative for each possible combination. This is the dreaded m x n problem. Given two polymorphic degrees of freedom (e.g., recording and format) the number of derivatives is the product of those degrees.

This problem is common enough that the combined use of Strategy and Template Method to solve it (as we did in the previous example) is a pattern in and of itself, called Bridge.