Kevlin Henney / khenney@qatraining.com

Something for nothing

IT HAS BEEN said that in the days of punched cards, if you wanted to find out how heavy a piece of software was you weighed all the holes. In spite of its apparent lack of substance, there are features of software that make it more substantial to its users and developers: You can develop classes, deploy them, reuse them, etc.; you can run applications and interact with them; you can describe software—especially software built with components and running with objects—in very physical terms.

If you want something really insubstantial you have to scratch the software development surface a little deeper: What are the successful design structures, concepts, and practices that go into making up a system? How do you communicate a piece of design to a colleague or express your ideas in code? What are the benefits of one solution over another, and how does applying a solution affect the subsequent design? For software, patterns seem to offer a discipline based on applied thought; naming, structuring, and communicating recurring solutions to problems.

In software development, the most visible popularization of patterns is the Gang of Four's (GoF) *Design Patterns* catalog,[1] containing 23 common design patterns found in object-oriented (OO) systems. However, these are not the only patterns, and there is a wealth of literature produced by the patterns community that focuses on patterns at all levels and across many different domains. This column is going to look beyond the relatively familiar GoF territory, presenting other examples that demonstrate patterns and pattern concepts of use to the Java developer, as well as some revisits of GoF patterns from a different perspective.

So, without further ado, let's look at a useful tactical pattern: Null Object[2] is perhaps best described as something based on nothing.

Kevlin Henney is a Principal Technologist with QA Training in the UK.

Null Object

The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable object that offers suitable default, do nothing behavior. In short, a design where, as William Shakespeare's King Lear would say, "nothing will come of nothing."

Problem

Given that an object reference may optionally be null, and that the result of a null check is to do nothing or use a default value, how can the presence of a null be treated transparently?

Example

Consider providing a logging facility for some kind of simple server service that can be used to record the outcome of operations and any significant events in the operation of the service. One can imagine many different kinds of log, such as a log that writes directly to the console or one that uses RMI to log a message on a remote server. However, a server is not required to use a log, so the association between the server and log is optional. Figure 1 shows the relationships diagrammed in UML, and Listing 1 shows the Log interface and two possible simple implementing classes.

In Listing 2, we can see that in using a Log object we must always ensure that we first check it is not null, because we know it is optional, i.e., logging need not be enabled for a service.

Forces

There is a great deal of procedural clunkiness in code, such as:

```
if(log != null)
   log.write(message);
```

This can be repetitive, as well as error prone: It is possible to forget to write such a null guard, and repeated checks for null references can clutter and obfuscate your code.

The user is required to detect the condition and

*Complete references are available at *Java Report Online*—www.javareport.com

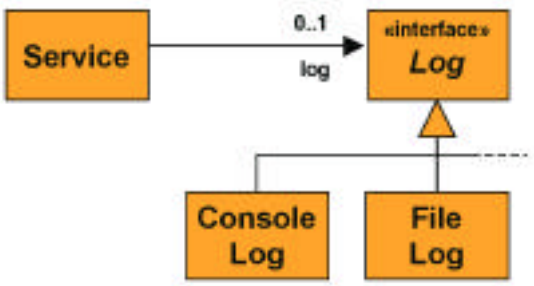

Figure 1. *UML class diagram of a service with optional support for logging.*

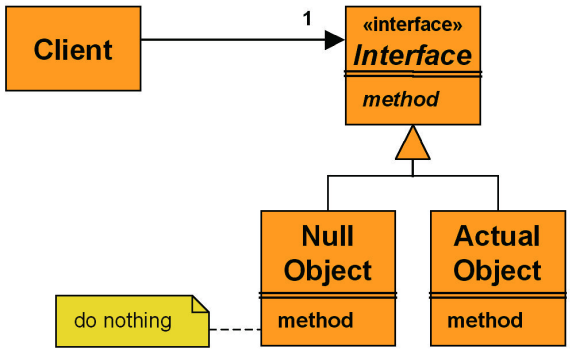

Figure 2. *Key classes in a* Null Object *collaboration.*

take appropriate inaction, even though the condition is not in any way exceptional, i.e., having a null log is a normal and expected state of the relationship. The condition makes the use of the logging facility less uniform.

A feature of conditional code is that it makes the decision flow of the code explicit. However, this is only a benefit if the decisions taken are important to the logic of the surrounding code; if they are not, then the resulting code is less rather than more direct, and the core algorithm becomes obscured.

Where conditional code does not serve the main purpose of a method, it tends to get in the way of the method's own logic, making the method longer and harder to understand.

LISTING 1.

The root Log **interface and sample concrete implementations.**

```
public interface Log
{
   void write(String messageToLog);
}

public class ConsoleLog implements Log
{
   public void write(String messageToLog)
   {
      System.out.println(messageToLog);
   }
}

public class FileLog implements Log
{
   public FileLog(String logFileName)
      throws IOException
   {
      out = new FileWriter(logFileName, true);
   }
   public void write(String messageToLog)
   {
      try
      {
         out.write("[" + new Date() + "] " +
                messageToLog + "\n");
         out.flush();
      }
      catch(IOException ignored)
      {
      }
   }
   private final FileWriter out;
}
```

This is especially true if the code is frequently repeated, as one might expect from the logging facility in the example, or if in-house coding guidelines encourage use of blocks in preference to single statements:

```
if(log != null)
{
   log.write(message);
}
```

The use of an explicit conditional means that the user may choose alternative actions to suit the context. However, if the action is always the same and if the optional relationship is frequently used, as one might expect of logging, it leads to duplication of the condition and its action. Duplicate code is considered to have a "bad smell,"[3] and works against simple changes (fixes or other improvements).

Where the relationship is null, there is no execution cost except a condition test and branch. On the other hand, the use of an explicit test means that there will always be a test. Because this test is repeated in separate pieces of code, it is

LISTING 2.

Initializing and using a Log **object in the** Service**.**

```
public class Service
{
   public Service()
   {
      this(null);
   }
   public Service(Log log)
   {
      this.log = log;
      ... // any other initialization
   }
   public Result handle(Request request)
   {
      if(log != null)
         log.write("Request " + request + " received");
      ...
      if(log != null)
         log.write("Request " + request + " handled");
      ...
   }
   ... // any other methods and fields
   private Log log;
}
```

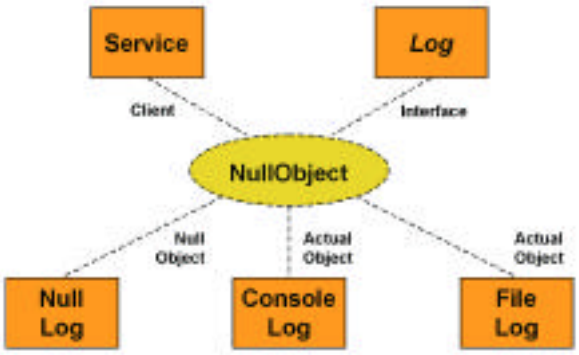

Figure 3. ***The relationship between the service structure and the roles in* Null Object.**

not possible to set debugging breakpoints or print statements that show all uses of a null case.

Solution

Provide a class that conforms to the interface required of the object reference, implementing all of its methods to do nothing or provide return default values. Use an instance of this class when the object reference would otherwise have been null. Figure 2 shows the essential configuration as a class model.

Resolution

Returning to the server example, we can introduce a Null Object class, NullLog, into the Log hierarchy. Such a comfortably null class (see Listing 3) does nothing and does not demand a great deal of coding skill!

The tactical benefit of a Null Object, once initialized, is in simplifying the use of this simple logging framework. It is now guaranteed that the relationship to a Log object is mandatory, i.e., the multiplicity from the Service to the Log is 1 rather than 0..1. This means that the conditional code can be eliminated, which makes the use of logging more uniform and the presence or absence of enabled logging transparent (see Listing 4).

Just as the essential structure for Null Object may be diagrammed in UML, its use in our code can be documented in a collaboration diagram. Figure 3 shows the classes and interfaces in our design and how they correspond, in terms of roles, to the elements outlined in Figure 2.

Consequences

Introducing a Null Object simplifies the client's code by eliminating superfluous and repeated conditionals that are not part of a method's core logic. The relationship moves from being optional to mandatory, and selection and variation are expressed through polymorphism and inheritance rather than procedural condition testing, making the use of the relationship more uniform. However, to preserve this invariant, care must be taken to correctly initialize the reference and to ensure either that it is not changed, i.e., declare it final, or that any change preserves the requirement that null is replaced by Null Object.

A Null Object is encapsulated and cohesive: It does one thing—nothing—and it does it well. This eliminates duplicate coding, makes the (absence of) behavior easier to reuse, and provides a suitable venue for debug breakpoints or print statements.

The absence of side effects in any method call to Null Object means that instances are immutable and therefore shareable and thread-safe, so that a Null Object is a Flyweight.[1] A Null Object is typically stateless, which means that all such instances will have the same behavior, and only one instance is required, suggesting that the object might be implemented as a Singleton.[1]

Using a Null Object in a relationship means that method calls will always be executed and arguments will always be evaluated. For the common case this is not a problem, but there will always be an overhead for method calls whose argument evaluation is complex and expensive.

The use of Null Object does not scale for distribution if passed by reference, e.g., if it implements java.rmi.Remote. Every method call would incur the overhead of a remote call and introduce a potential point of failure, both of which would swamp and undermine the basic *do nothing* behavior. Therefore, if used in a distributed context, either null should be passed in preference to a Null Object reference or pass by value should be used if RMI is the distribution mechanism, i.e., transparent replication rather than transparent sharing becomes the priority. Because a Null Object class is small and simple, class loading is cheap and the only change the programmer needs to make is to implement java.io.Serializable:

```
public class NullLog implements Log, Serializable
{
   ...
}
```

LISTING 3.

A NullLog class providing *do nothing* behavior.

```
public class NullLog implements Log
{
  public void write(String messageToLog)
  {
  }
}
```

LISTING 4.

Introducing a NullLog object into Service.

```
public class Service
{
   public Service()
   {
      this(new NullLog());
   }
   ...
   public Result handle(Request request)
   {
      log.write("Request " + request + " received");
      ...
      log.write("Request " + request + " handled");
      ...
   }
   ...
   private Log log;
}
```

Users of a hierarchy that includes a Null Object will have more classes to take on board. The benefit is that with the exception of the point of creation, explicit knowledge of the new Null Object class is not needed. Where there are many ways of *doing nothing*, more Null Object classes can be introduced. Variations on this theme include the use of an Exceptional Value object,[4] that rather than doing nothing, raises an exception for any method call or returns Exceptional Value objects as a result.

Taking an existing piece of code that does not use Null Object and modifying it to do so may accidentally introduce bugs. By carefully following the Introduce Null Object refactoring,[3] developers can have greater confidence making such changes and avoiding pitfalls. When introducing a Null Object class into an existing system, a suitable base interface may not exist for the Null Object class to implement. Either that part of the system should be refactored to introduce one or the Null Object class must subclass the concrete class referred to by the reference. The latter approach is a little undesirable, as it implies that if there is any representation in the superclass it will be ignored. Such inheritance with cancellation can lead to code that is harder to understand because the subclass does not truly conform to the superclass, i.e., fails the *is a* or *is a kind of* litmus test for good use of inheritance and includes redundant implementation that cannot be uninherited. This approach also cannot be used if the concrete class or any of its methods are declared final. Note that a Null Object class should not be used as a superclass except for other Null Object classes.

Pattern Anatomy

Any pattern is a narrative, essentially a story that unfolds with *dramatis personae*, the to and fro movements of the plot elements, and a happy ending. A pattern is also a guide to construction, telling you when and how to build something, passing on strategies, tactics, and observations from experience. This is as true of software development patterns as it is of building architecture where patterns originated.[5,6]

What You Should Find in a Pattern

A pattern identifies a general approach to solving a problem, typically capturing a solution practice or collaborative structure. It identifies the general context of the problem, the nature of the problem, the interplay of conflicting forces that create the problem and binds its solution, the configuration that resolves the problem, and the resulting context of applying such a solution. In short: where, what, and how the problem arises and is solved. Rather than living in the abstract, a pattern needs to be illustrated by something concrete, such as an example with code and diagrams.

Patterns also tend not to live in isolation, and so a pattern typically contains references to other patterns that complete it. Such links increase the value and usability of a pattern because they lay out some of the possible tracks your development can follow, for instance, the use of Singleton with Null Object. A pattern can also contain discussion of patterns that are in some way similar to it or coincidentally related. However, there is a danger that a pattern written in this style can end up as a comparative pattern study or treatise, i.e., an academic exercise rather than a piece of knowledge focused on practice and problem solving.

Patterns are considered to be about real things, so they are empirical rather than theoretical, and they are about repeated rather than one-time solutions to problems. There is often an onus on pattern authors to demonstrate this, such as by including a listing of some known uses of a pattern that cite, for example, specific frameworks and libraries. We could cite a number of framework examples for Null Object, as it has been discovered time and time again, but as a general pattern—as opposed to a specifically OO design pattern—it is perhaps more common than people realize: Consider the role of the null file device on many operating systems (/dev/null on Unix and NUL on Microsoft systems), no-op machine instructions, terminators on Ethernet cables, etc.

Finally, one of the most significant parts of a pattern is its name. The name allows us to discuss solutions easily and at an appropriate level of abstraction. This suggests that a pattern's name should be drawn from the solution and should be an evocative noun phrase[7] that can be used in ordinary conversation, i.e., to support usage such as "if we use a Null Object..." as opposed to "if we apply the Null Object pattern...." It is not uncommon for a pattern to have different names, either because different people have documented it at different times or because other names have different connotations. For instance, Null Object has also been known as Active Nothing and Smart NIL.

Pattern Form

There are many forms for documenting patterns,[8] ranging from the highly structured heading-based template form[1] to more narrative, literary forms.[5] The essential elements of a pattern can be found in each form, and the choice of form tends to be a matter of personal preference as much as anything else.

A quick scan through pattern sources such as the Patterns Home Page[9] or the *Pattern Languages of Program Design* (PLoPD) books[10–12] should be enough to convince anyone of the diversity of pattern form. Null Object has been documented in a variety of forms, which vary widely in structure and length: from the structured GoF-like form,[2] to the shorter structured form used in this column, to the brief thumbnail-like production rule form.[13]

Conclusion

Patterns can quite literally help create a vocabulary for development, accelerating communication, and the understanding of ideas. They are pieces of literature that capture successful structures and practices, offering developers reusable ideas rather than reusable code. A pattern's form is essentially a vehicle for conveying these.

As a specific example, Null Object illustrates how substituting a stateless object that does nothing offers a surprisingly powerful solution. It can be used tactically in many systems to encapsulate and simplify control flow—something for nothing. ■