Null Object

Something for Nothing

Kevlin Henney
March 2003

kevlin@curbralan.com
kevlin@acm.org

Abstract

The intent of a NULL OBJECT is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default *do nothing* behavior. In short, a design where "nothing will come of nothing" [Shakespeare1605].

NULL OBJECT is a tactical pattern that has been discovered time and again, and not only in object-oriented systems: null file devices (`/dev/null` on Unix and `NUL` on Microsoft systems), no-op machine instructions, terminators on Ethernet cables, etc.

The pattern has been documented in a variety of forms by many authors, varying widely in structure and length: from a thorough and structured GOF-like form [Woolf1998] to a brief thumbnail-like production-rule form [Henney1997]. This paper is derived from a previously published article [Henney1999] and includes the aforementioned thumbnail. The aim of the current work is to update and capture the latest understanding of the pattern and its implications, also addressing a wide audience by documenting the pattern primarily with respect to Java and UML.

Thumbnail

if

- An object reference may be optionally null ***and***
- This reference must be checked before every use ***and***
- The result of a null check is to do nothing or assign a suitable default value

then

- Provide a class derived from the object reference's type ***and***
- Implement all its methods to do nothing or provide default results ***and***
- Use an instance of this class whenever the object reference would have been null

Problem

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

Example

Consider a logging facility for some kind of simple server-housed service. It can be used to record exceptional events, housekeeping activities, and the outcome of operations during the course of the service's operation. One can imagine many different kinds of log, such as a log that writes directly to the console or one that uses RMI to send a message to a remote logging server. However, a server is not necessarily required to use a log, so the association between the server and the log is optional. *Figure 1* shows the relationships diagrammed in UML, and *Listing 1* shows the `Log` interface and two simple implementations.

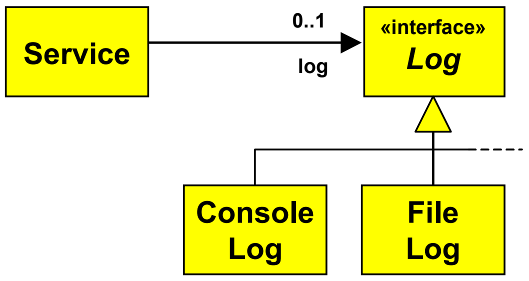

Figure 1. UML class diagram of a service with optional support for logging.

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

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

```java
public class FileLog implements Log
{
    public FileLog(String logFileName)
    {
        try
        {
            out = new FileWriter(logFileName, true);
        }
        catch(IOException caught)
        {
            throw new RuntimeException("Failed to open log file: " + caught);
        }
    }
    public void write(String messageToLog)
    {
        try
        {
            out.write("[" + new Date() + "] " + messageToLog + "\n");
            out.flush();
        }
        catch(IOException caught)
        {
            throw new RuntimeException("Failed to write to log: " + caught);
        }
    }
    private final FileWriter out;
}
```

Listing 1. The root `Log` interface and sample concrete implementations.

In *Listing 2*, it can be seen that because the option exists for not having logging enabled for a service, a check against `null` is required before every use of `log`.

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

Listing 2. Initializing and using a `Log` object in the `Service`.

Forces

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

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

This style is repetitive as well as error prone: repeated checks for null references can clutter and obfuscate code; it is too easy to forget to write the null guard. However, the user is required to detect the condition and take appropriate inaction, even though the condition is not in any way exceptional — 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 the explicitness of its conditions and the visibility of the resulting control flow. However, this is only a benefit if the decisions taken are important to the logic of the surrounding code, otherwise the resulting code is less — rather than more — direct. Such minor but essential decisions become distractions rather than attractions, obscuring the core algorithm.

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. This is especially true if the code is repeated frequently, as one might expect from the logging facility in the example, or if in-house coding guidelines encourage the use of blocks to surround 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" [Fowler1999]. Duplication works against simple changes, such as fixes or improvements, and is in violation of the DRY principle (Don't Repeat Yourself) [Hunt+2000].

For an explicit conditional on a null reference the cost of execution amounts to no more than a test and a 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 not possible to set debugging breakpoints, or introduce diagnostic statements, consistently for all uses of the null case.

The code would be much clearer if the `null` tests could be ignored or hidden. The JVM already checks each access via any reference against `null`, and there can be a temptation to exploit this behavior (see *Listing 3*). This technique is specific to languages that guarantee that all accesses are checked, e.g. Java and C#, and will lead to undesirable (undefined) results where this guarantee is absent, e.g. C++.

```
public class Service
{
    ...
    public Result handle(Request request)
    {
        try
        {
            log.write("Request " + request + " received");
            ...
            log.write("Request " + request + " handled");
            ...
        }
        catch(NullPointerException ignored)
        {
        }
    }
    ...
}
```

Listing 3. Assuming a non-null `log` and ignoring any `NullPointerException`.

There are two objections to such piggybacking. There is the matter of style and taste, as well as the cost of overgeneralization at the expense of correctness:

- *Style*: In spite of the vagueness of the common advice that "exceptions should be exceptional", the reliance on `NullPointerException` for masking the absence of logging does seem to be a clear abuse of what is otherwise a hollow platitude. The normal motivation for adopting such a style is as a performance micro-optimization; such reasoning does not apply to *Listing 3*.
- *Correctness*: A `NullPointerException` is thrown for any access via `null`, and not just writing through `log`. Ignoring such exceptions runs the obvious risk of quashing genuine errors, throwing the baby out with the bath water. To check `log` against `null` in the `catch` clause further uglifies the code, and still does not offer a guarantee in all cases that a null `log` was the culprit.

What is needed of a solution is the ability to wish away the `null` so that nothing — rather than something exceptional — happens. The solution should be non-intrusive, easy to code, and inexpensive at runtime.

Solution

Provide something for nothing: A class that conforms to the interface required of the object reference, implementing all of its methods to do nothing or to return suitable default values. Use an instance of this class when the object reference would otherwise have been `null`. *Figure 2* shows an essential, typical NULL OBJECT configuration as a class model. Other expressions of this pattern are, of course, possible.

A NULL OBJECT can be used to complete many other common patterns and object structures: a null ITERATOR [Gamma+1995] goes nowhere; a null COMMAND [Gamma+1995] does (and undoes) nothing; a pointer to an empty function can be used to provide null callback behavior in C; a null collection is empty and cannot be changed; a null STRATEGY [Gamma+1995] provides no algorithmic behavior, a generalization that extends to compile-time binding of template policies in C++, e.g. non-locking behavior for STRATEGIZED LOCKING in a single-threaded environment [Schmidt+2000]; a null lamina can terminate LAYERS [Buschmann+1996], such as a null socket layer for a standalone workstation.

A NULL OBJECT should not be used indiscriminately as a replacement for null references. Its intent is to encapsulate the absence of an object where that absence is not profoundly significant to the user of an actual object. If the optionality has fundamental meaning that leads to different behavior, a NULL OBJECT would be an inappropriate. Any need for a runtime type query, such as an `isNull` method, indicates that absence is significant rather than transparent, suggesting a `null` rather than a NULL OBJECT.

For example, in a graphical editor, a null fill color leads to transparency and would be a good use of a NULL OBJECT. However, the state of being unmarried is not best represented with a null spouse object: marriage is optional, and being single really is different.

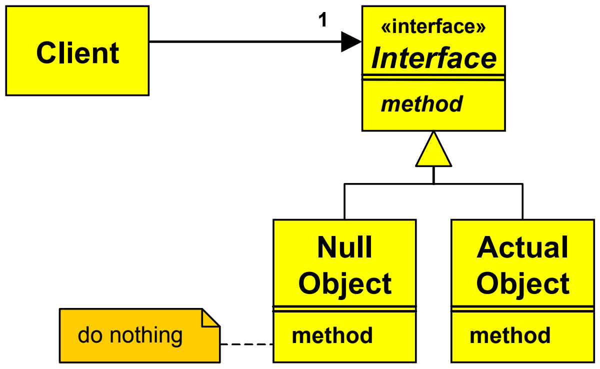

Figure 2. Key roles in a typical NULL OBJECT collaboration.

Resolution

Returning to the server-logging example, a NULL OBJECT class, `NullLog`, can be introduced into the `Log` hierarchy. Such a comfortably null class (see *Listing 4*) does nothing and does not demand a great deal of coding skill!

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

Listing 4. A `NullLog` class providing do nothing *behavior.*

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 strengthened relationship allows the explicit conditional code to be eliminated, making logging simpler and more uniform. Polymorphism takes up the responsibility for selection, making the presence (or absence) of logging transparent (see *Listing 5*).

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

Listing 5. Introducing a `NullLog` object into `Service`.

Figure 3 shows the classes and interfaces in the design (based on *Listing 4*, *Listing 5*, and additional classes from *Figure 1*) and how they correspond, in terms of roles, to the elements outlined in *Figure 2*.

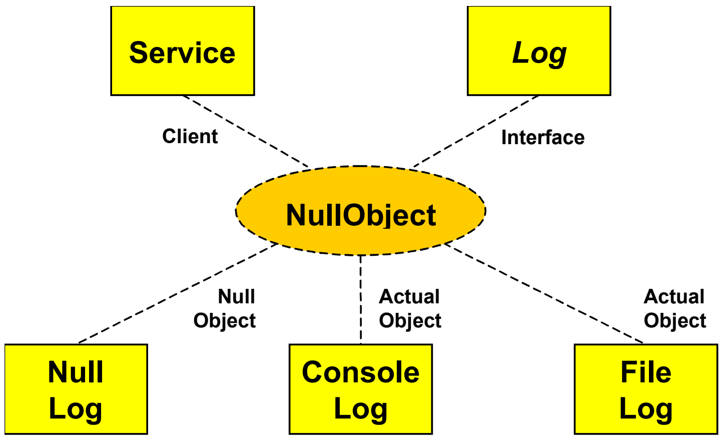

Figure 3. The relationship between the problem resolution and the roles described in Figure 2.

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. Selection and variation are expressed through polymorphism and inheritance rather than procedural condition testing. Taking a step back, polymorphism can be seen to magic away `switch` statements or `if else if` cascades. In the specific case of NULL OBJECT, the missing (but implied) empty `default` or `else` has been captured and concealed.

The object relationship moves from being optional to mandatory, making the use of the relationship more uniform. However, to preserve this invariant, care must be taken to ensure that the reference is never seen as a `null`:

- The reference may be declared `final` so that only the initialization is required to guard against a null reference, setting a NULL OBJECT in its place.
- Alternatively, only INDIRECT VARIABLE ACCESS [Beck1997] should be used to access the reference. A SETTING METHOD [Beck1997] can ensure that the reference is assigned a NULL OBJECT rather than a null reference, or a GETTING METHOD [Beck1997] can ensure that a null reference is returned as a NULL OBJECT.

A NULL OBJECT is encapsulated and cohesive: It does one thing — nothing — and it does it well. This reification of the void and encapsulation of emptiness eliminates duplicate coding, makes the (absence of) behavior easier to use, and provides a suitable venue for debug breakpoints or print statements.

The absence of any side effects on a NULL OBJECT means that instances are immutable, and are therefore shareable and intrinsically thread safe. A NULL OBJECT is typically stateless and its identity is not a significant part of its make up, which means that all instances are equivalent. If a time or space optimization is required a single `static` instance, but not necessarily a SINGLETON object [Gamma+1995], may be used in place of freshly instantiated NULL OBJECTs.

On the other hand, using a NULL OBJECT in a relationship means that method calls will always be executed and their arguments will always be evaluated. For the common case this is not a problem, but there will always be an identifiable overhead for method calls whose argument evaluation is complex and expensive.

The use of NULL OBJECT does not scale to remote objects. If a Java NULL OBJECT were to implement `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 or the NULL OBJECT should be passed by copy. In a distributed environment transparent replication rather than transparent sharing becomes the priority. For RMI this does not introduce a significant overhead, because a NULL OBJECT class is small and loading it will be relatively cheap. The only change to the code is to ensure that the NULL OBJECT class implements `java.io.Serializable`:

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

Users of a hierarchy that includes a NULL OBJECT class 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. A variation on this theme is the use of an EXCEPTIONAL VALUE object [Cunningham1995]. Rather than doing nothing when a method is called, an EXCEPTIONAL VALUE either raises an exception or returns a further EXCEPTIONAL VALUE. A generalization of this theme is the SPECIAL CASE [Fowler2003], where an object in some way represents a special case, such as an EXCEPTIONAL VALUE or a NULL OBJECT, each of which can be considered special cases of SPECIAL CASE.

The implementation of *do nothing* behavior is simple for many methods: empty method bodies. However, methods that return results can only avoid the inevitable `return` statement by throwing an exception — behavior more appropriate for an EXCEPTIONAL VALUE than for a NULL OBJECT. Different parameter passing modes require different responses:

- *in* arguments: Arguments passed to the method to provide it with suitable information can be safely ignored, requiring no handling code in the method body. Pass by copy is Java's sole mechanism for passing arguments. In CORBA the *in* mode corresponds to `in` arguments. In C++ pass by copy and pass by `const` reference play this role. In C# the *in* mode is the default for method arguments.
- `void` result: No method code is required to deal with a `void` return type.
- *out* arguments: An *out* argument must, by definition, be set and so the method body must set it to an appropriate default. Java does not support these directly, but the HOLDER idiom is often used to emulate it. This can is used in the mapping of CORBA's `out` arguments. In C# an `out` argument must also be set.
- Non-`void` result: As with *out* arguments, an appropriate default must be returned.
- *in–out* arguments: An *in–out* argument may be either ignored, as with an *in* argument, or set to an appropriate default, as with an *out* argument. Java only supports *in–out* arguments via the HOLDER idiom, as found in the mapping of CORBA's `inout` arguments. The corresponding feature in C++ is pass by reference. In C# `ref` arguments fulfil this role.

However, what precisely is meant by "appropriate default"?

- *Success defaults*: A success value is often an identity value of some kind and does not indicate any form of failure, and will typically not affect the caller. For numeric types, `0` is often such a value. For result objects that represent values, a `null` reference is sometimes suitable, but more often a default constructed object is a better result, e.g. `""` as opposed to `null` for a `String`. For result objects that represent behavior, either a `null` or a NULL OBJECT of the result type should be returned.
- *Failure defaults*: A failure value shows that a method call was in some way unsuccessful, but in the context of NULL OBJECT it should probably not be fatal. For integer types, `-1` is often used to signal failure. For floating point types, *NaN* or infinity are the common out-of-band results. For objects, either a `null` reference or an EXCEPTIONAL VALUE should be returned.

If no defaults are required — e.g. a method taking no arguments or *in* arguments only and returning `void` — or results can be default constructed — e.g. a return value of `0` for a numeric or a `null` for a reference — a *dynamic proxy* can be used as a generalized NULL OBJECT. Dynamic proxies are a JDK 1.3 feature (`java.lang.reflect.Proxy`) and, based on reflection, they can interpret arbitrary messages. A NULL OBJECT dynamic proxy would simply discard all method requests, returning default values where necessary.

Taking an existing piece of code that does not use NULL OBJECT and modifying it to do so may also open the door to bugs. By following carefully the INTRODUCE NULL OBJECT refactoring [Fowler1999] programmers 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 an EXTRACT INTERFACE refactoring [Fowler1999] should be applied to introduce one or the NULL OBJECT class must subclass the concrete class referred to by the reference. The second option may be unavoidable if the hierarchy code cannot be rearranged, but it is less desirable than the first option: It implies that any representation in the superclass 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, failing the *is a* or *is a kind of* litmus test for appropriate use of inheritance. The inclusion of redundant implementation that cannot be disinherited suggests a tradeoff with weaker cohesion. However, inheritance from an existing concrete class cannot be used if it or any of its methods are declared `final`.

Note that, for the same reasons of substitutability, a NULL OBJECT class should not be used as a superclass except for other NULL OBJECT classes. The assumption is that any inheritance of a NULL OBJECT class will preserve that classification — the subclass of a NULL OBJECT class is also a NULL OBJECT class, and its instances can be used where any NULL OBJECT is expected.

Acknowledgments

An earlier version of this paper was published in *Java Report* (RIP) [Henney1999]. The current version has also benefited from earlier papers by Bruce Anderson [Anderson1996] and Bobby Woolf [Woolf1998].

I would like to thank the following people: Joe Bergin for his shepherding and humor; Hubert Matthews for his additional comments on the preshepherded version of this paper; Frank Buschmann and the Siemens patterns retreat in St Martin, Austria, in June 2000 for their company and comments on a previous draft of this paper; Jim Coplien, Pascal Costanza, Jutta Eckstein, Ian Graham, Klaus Marquardt, Alan O'Callaghan, Andreas Rüping, Titos Saridakis, Didi Schütz, Christa Schwanninger, Markus Völter, and Uwe Zdun for the comments they gave on the paper at EuroPLoP 2002.

References

[Anderson1996] Bruce Anderson, "Null Object", *PloP '96*.

[Beck1997] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[Cunningham1995] Ward Cunningham, "The CHECKS Pattern Language of Information Integrity", *Pattern Languages of Program Design*, edited by James O Coplien and Douglas C Schmidt, Addison-Wesley, 1995.

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Fowler2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 1999.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney1997] Kevlin Henney, "Java Patterns and Implementations", presented at *BCS OOPS Patterns Day*, October 1997, `http://www.curbralan.com`.

[Henney1999] Kevlin Henney, "Patterns in Java: Something for Nothing", *Java Report* 4(12), December 1999, `http://www.curbralan.com`.

[Hunt+2000] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley 2000.

[Schmidt+2000] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[Shakespeare1605] William Shakespeare, *King Lear*, 1605.

[Woolf1998] Bobby Woolf, "Null Object", *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.