WIKIPEDIA

Null object pattern

In object-oriented computer programming, a **null object** is an object with no referenced value or with defined neutral ("null") behavior. The null object design pattern describes the uses of such objects and their behavior (or lack thereof). It was first published in the *Pattern Languages of Program Design* book series.[1]

Contents

Motivation

Description

Example

Relation to other patterns

Alternatives
- Extension methods and Null coalescing

In various languages
- C++
- C#
- Smalltalk
- Common Lisp
- CLOS
- Scheme
- Ruby
- JavaScript
- Java
- PHP
- Visual Basic .NET

Criticism

See also

References

External links

Motivation

In most object-oriented languages, such as Java or C#, references may be null. These references need to be checked to ensure they are not null before invoking any methods, because methods typically cannot be invoked on null references.

The Objective-C language takes another approach to this problem and does nothing when sending a message to `nil`; if a return value is expected, `nil` (for objects), `0` (for numeric values), `NO` (for `BOOL` values), or a struct (for struct types) with all its members initialised to `null`/`0`/`NO`/zero-initialised struct is returned.[2]

Description

Instead of using a null reference to convey absence of an object (for instance, a non-existent customer), one uses an object which implements the expected interface, but whose method body is empty. The advantage of this approach over a working default implementation is that a null object is very predictable and has no side effects: it does *nothing*.

For example, a function may retrieve a list of files in a folder and perform some action on each. In the case of an empty folder, one response may be to throw an exception or return a null reference rather than a list. Thus, the code which expects a list must verify that it in fact has one before continuing, which can complicate the design.

By returning a null object (i.e. an empty list) instead, there is no need to verify that the return value is in fact a list. The calling function may simply iterate the list as normal, effectively doing nothing. It is, however, still possible to check whether the return value is a null object (an empty list) and react differently if desired.

The null object pattern can also be used to act as a stub for testing, if a certain feature such as a database is not available for testing.

Example

Given a binary tree, with this node structure:

```
class node {
  node left
  node right
}
```

One may implement a tree size procedure recursively:

```
function tree_size(node) {
  return 1 + tree_size(node.left) + tree_size(node.right)
}
```

Since the child nodes may not exist, one must modify the procedure by adding non-existence or null checks:

```
function tree_size(node) {
  set sum = 1
  if node.left exists {
    sum = sum + tree_size(node.left)
  }
  if node.right exists {
    sum = sum + tree_size(node.right)
  }
  return sum
}
```

This however makes the procedure more complicated by mixing boundary checks with normal logic, and it becomes harder to read. Using the null object pattern, one can create a special version of the procedure but only for null nodes:

```
function tree_size(node) {
  return 1 + tree_size(node.left) + tree_size(node.right)
}
```

```
function tree_size(null_node) {
  return 0
}
```

This separates normal logic from special case handling, and makes the code easier to understand.

Relation to other patterns

It can be regarded as a special case of the State pattern and the Strategy pattern.

It is not a pattern from *Design Patterns*, but is mentioned in Martin Fowler's *Refactoring*[3] and Joshua Kerievsky's Refactoring To Patterns[4] as the *Insert Null Object* refactoring.

Chapter 17 of Robert Cecil Martin's *Agile Software Development: Principles, Patterns and Practices*[5] is dedicated to the pattern.

Alternatives

From C# 6.0 it is possible to use the "?." operator (aka null-conditional operator), which will simply evaluate to null if its left operand is null.

```
// compile as Console Application, requires C# 6.0 or higher
using System;

namespace ConsoleApplication2
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            string str = "test";
            Console.WriteLine(str?.Length);
            Console.ReadKey();
        }
    }
}
// The output will be:
// 4
```

Extension methods and Null coalescing

In some Microsoft .NET languages, Extension methods can be used to perform what is called 'null coalescing'. This is because extension methods can be called on null values as if it concerns an 'instance method invocation' while in fact extension methods are static. Extension methods can be made to check for null values, thereby freeing code that uses them from ever having to do so. Note that the example below uses the C# Null coalescing operator to guarantee error free invocation, where it could also have used a more mundane if...then...else. The following example only works when you do not care the existence of null, or you treat null and empty string the same. The assumption may not hold in other applications.

```
// compile as Console Application, requires C# 3.0 or higher
using System;
using System.Linq;
namespace MyExtensionWithExample {
    public static class StringExtensions {
        public static int SafeGetLength(this string valueOrNull) {
            return (valueOrNull ?? string.Empty).Length;
        }
    }
    public static class Program {
        // define some strings
        static readonly string[] strings = new [] { "Mr X.", "Katrien Duck", null, "Q" };
        // write the total length of all the strings in the array
        public static void Main(string[] args) {
            var query = from text in strings select text.SafeGetLength(); // no need to do any checks here
            Console.WriteLine(query.Sum());
        }
    }
}
// The output will be:
// 18
```

In various languages

C++

A language with statically typed references to objects illustrates how the null object becomes a more complicated pattern:

```
class animal
{
public:
  virtual void make_sound() const = 0;
};

class dog : public animal
{
  virtual void make_sound() const override
  {
      std::cout << "woof!" << std::endl;
  }
};

class null_animal : public animal
{
  virtual void make_sound() const override { }
};
```

Here, the idea is that there are situations where a pointer or reference to an `animal` object is required, but there is no appropriate object available. A null reference is impossible in standard-conforming C++. A null `animal *` pointer is possible, and could be useful as a place-holder, but may not be used for direct dispatch: `a->make_sound()` is undefined behavior if `a` is a null pointer.

The null object pattern solves this problem by providing a special `null_animal` class which can be instantiated bound to an `animal` pointer or reference.

The special null class must be created for each class hierarchy that is to have a null object, since a `null_animal` is of no use when what is needed is a null object with regard to some `widget` base class that is not related to the `animal` hierarchy.

Note, that NOT having a null class at all is an important feature, in contrast to languages where "anything is a reference" (e.g. Java and C#). In C++, the design of a function or method may explicitly state whether null is allowed or not.

```
// function which requires an animal instance,
// and will not accept null
void do_something( const animal& Inst ) {
  // Inst may never be null here
}

// function which may accept an animal instance or null
void do_something( const animal* pInst ) {
  // pInst may be null
}
```

C#

C# is a language in which the null object pattern can be properly implemented. This example shows animal objects that display sounds and a NullAnimal instance used in place of the C# null keyword. The null object provides consistent behaviour and prevents a runtime null reference exception that would occur if the C# null keyword were used instead.

```
/* Null object pattern implementation:
 */
using System;

// Animal interface is the key to compatibility for Animal implementations below.
interface IAnimal
{
    void MakeSound();
}

// Animal is the base case.
abstract class Animal : IAnimal
{
    // A shared instance that can be used for comparisons
    public static readonly IAnimal Null = new NullAnimal();

    // The Null Case: this NullAnimal class should be used in place of C# null keyword.
    private class NullAnimal : Animal
    {
        public override void MakeSound()
        {
            // Purposefully provides no behaviour.
        }
    }
    public abstract void MakeSound();
}

// Dog is a real animal.
class Dog : IAnimal
{
    public void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

/* ========================
 * Simplistic usage example in a Main entry point.
 */
static class Program
{
    static void Main()
    {
        IAnimal dog = new Dog();
        dog.MakeSound(); // outputs "Woof!"

        /* Instead of using C# null, use the Animal.Null instance.
```

```
         * This example is simplistic but conveys the idea that if the Animal.Null instance is used then the program
         * will never experience a .NET System.NullReferenceException at runtime, unlike if C# null were used.
         */
        IAnimal unknown = Animal.Null;  //<< replaces: IAnimal unknown = null;
        unknown.MakeSound(); // outputs nothing, but does not throw a runtime exception
    }
}
```

Smalltalk

Following the Smalltalk principle, *everything is an object*, the absence of an object is itself modeled by an object, called `nil`. In the GNU Smalltalk for example, the class of `nil` is `UndefinedObject`, a direct descendant of `Object`.

Any operation that fails to return a sensible object for its purpose may return `nil` instead, thus avoiding the special case of returning "no object". This method has the advantage of simplicity (no need for a special case) over the classical "null" or "no object" or "null reference" approach. Especially useful messages to be used with `nil` are `isNil` or `ifNil:`, which make it practical and safe to deal with possible references to `nil` in Smalltalk programs.

Common Lisp

In Lisp, functions can gracefully accept the special object `nil`, which reduces the amount of special case testing in application code. For instance, although `nil` is an atom and does not have any fields, the functions `car` and `cdr` accept `nil` and just return it, which is very useful and results in shorter code.

Since `nil` **is** the empty list in Lisp, the situation described in the introduction above doesn't exist. Code which returns `nil` is returning what is in fact the empty list (and not anything resembling a null reference to a list type), so the caller does not need to test the value to see whether or not it has a list.

The null object pattern is also supported in multiple value processing. If the program attempts to extract a value from an expression which returns no values, the behavior is that the null object `nil` is substituted. Thus `(list (values))` returns `(nil)` (a one-element list containing nil). The `(values)` expression returns no values at all, but since the function call to `list` needs to reduce its argument expression to a value, the null object is automatically substituted.

CLOS

In Common Lisp, the object `nil` is the one and only instance of the special class `null`. What this means is that a method can be specialized to the `null` class, thereby implementing the null design pattern. Which is to say, it is essentially built into the object system:

```
;; empty dog class

(defclass dog () ())

;; a dog object makes a sound by barking: woof! is printed on standard output
;; when (make-sound x) is called, if x is an instance of the dog class.
```

```
(defmethod make-sound ((obj dog))
  (format t "woof!~%"))

;; allow (make-sound nil) to work via specialization to null class.
;; innocuous empty body: nil makes no sound.
(defmethod make-sound ((obj null)))
```

The class `null` is a subclass of the `symbol` class, because `nil` is a symbol. Since `nil` also represents the empty list, `null` is a subclass of the `list` class, too. Methods parameters specialized to `symbol` or `list` will thus take a `nil` argument. Of course, a `null` specialization can still be defined which is a more specific match for `nil`.

Scheme

Unlike Common Lisp, and many dialects of Lisp, the Scheme dialect does not have a nil value which works this way; the functions `car` and `cdr` may not be applied to an empty list; Scheme application code therefore has to use the `empty?` or `pair?` predicate functions to sidestep this situation, even in situations where very similar Lisp would not need to distinguish the empty and non-empty cases thanks to the behavior of `nil`.

Ruby

In duck-typed languages like Ruby, language inheritance is not necessary to provide expected behavior.

```
class Dog
  def sound
    "bark"
  end
end

class NilAnimal
  def sound(*); end
end

def get_animal(animal=NilAnimal.new)
  animal
end

get_animal(Dog.new).sound
 => "bark"
get_animal.sound
 => nil
```

Attempts to directly monkey-patch NilClass instead of providing explicit implementations give more unexpected side effects than benefits.

JavaScript

In duck-typed languages like JavaScript, language inheritance is not necessary to provide expected behavior.

```
class Dog {
  sound() {
    return 'bark';
  }
}

class NullAnimal {
  sound() {
    return null;
  }
}

function getAnimal(type) {
  return type === 'dog' ? new Dog() : new NullAnimal();
}

['dog', null].map((animal) => getAnimal(animal).sound());
// Returns ["bark", null]
```

Java

```
public interface Animal {
    void makeSound() ;
}

public class Dog implements Animal {
    public void makeSound() {
        System.out.println("woof!");
    }
}

public class NullAnimal implements Animal {
    public void makeSound() {
                // silence...
    }
}
```

This code illustrates a variation of the C++ example, above, using the Java language. As with C++, a null class can be instantiated in situations where a reference to an `Animal` object is required, but there is no appropriate object available. A null `Animal` object is possible (`Animal myAnimal = null;`) and could be useful as a place-holder, but may not be used for calling a method. In this example, `myAnimal.makeSound();` will throw a NullPointerException. Therefore, additional code may be necessary to test for null objects.

The null object pattern solves this problem by providing a special `NullAnimal` class which can be instantiated as an object of type `Animal`. As with C++ and related languages, that special null class must be created for each class hierarchy that needs a null object, since a `NullAnimal` is of no use when what is needed is a null object that does not implement the `Animal` interface.

PHP

```php
interface Animal {
    public function makeSound();
}

class Dog implements Animal {
    public function makeSound() {
        echo "Woof..";
    }
}

class Cat implements Animal {
    public function makeSound() {
        echo "Meowww..";
    }
}

class NullAnimal implements Animal {
    public function makeSound() {
        // silence...
    }
}

$animalType = 'elephant';
switch($animalType) {
    case 'dog':
        $animal = new Dog();
        break;
    case 'cat':
        $animal = new Cat();
        break;
    default:
        $animal = new NullAnimal();
        break;
}
$animal->makeSound(); // ..the null animal makes no sound
```

Visual Basic .NET

The following null object pattern implementation demonstrates the concrete class providing its corresponding null object in a static field `Empty`. This approach is frequently used in the .NET Framework (`String.Empty`, `EventArgs.Empty`, `Guid.Empty`, etc.).

```
Public Class Animal
    Public Shared ReadOnly Empty As Animal = New AnimalEmpty()

    Public Overridable Sub MakeSound()
        Console.WriteLine("Woof!")
    End Sub
End Class

Friend NotInheritable Class AnimalEmpty
    Inherits Animal

    Public Overrides Sub MakeSound()
        '
    End Sub
End Class
```

Criticism

This pattern should be used carefully as it can make errors/bugs appear as normal program execution.[6]

Care should be taken not to implement this pattern just to avoid null checks and make code more readable, since the harder to read code may just move to another place and be less standard – such as when different logic must execute in case the object provided is indeed the null object. The common pattern in most languages with reference types is to compare a reference to a single value referred to as null or nil. Also, there is additional need for testing that no code anywhere ever assigns null instead of the null object, because in most cases and languages with static typing, this is not a compiler error if the null object is of a reference type, although it would certainly lead to errors at run time in parts of the code where the pattern was used to avoid null checks. On top of that, in most languages and assuming there can be many null objects (i.e. the null object is a reference type but doesn't implement the singleton pattern in one or another way), checking for the null object instead of for the null or nil value introduces overhead, as does the singleton pattern likely itself upon obtaining the singleton reference.

See also

- Nullable type
- Option type

References

1. Woolf, Bobby (1998). "Null Object". In Martin, Robert; Riehle, Dirk; Buschmann, Frank. *Pattern Languages of Program Design 3*. Addison-Wesley.
2. "Working with Objects (Working with nil)" (https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW22). *iOS Developer Library*. Apple, Inc. 2012-12-13. Retrieved 2014-05-19.
3. Fowler, Martin (1999). *Refactoring. Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.
4. Kerievsky, Joshua (2004). *Refactoring To Patterns*. Addison-Wesley. ISBN 0-321-21335-1.

5. Martin, Robert (2002). *Agile Software Development: Principles, Patterns and Practices*. Pearson Education. ISBN 0-13-597444-5.
6. Fowler, Martin (1999). Refactoring pp. 216

External links

- Jeffrey Walker's account of the Null Object Pattern (http://www.cs.oberlin.edu/~jwalker/nullObjPattern/)
- Martin Fowler's description of Special Case, a slightly more general pattern (http://martinfowler.com/eaaCatalog/specialCase.html)
- Null Object Pattern Revisited (http://www.owlnet.rice.edu/~comp212/00-spring/handouts/week06/null_object_revisited.htm)
- Introduce Null Object refactoring (http://refactoring.com/catalog/introduceNullObject.html)
- SourceMaking Tutorial (http://sourcemaking.com/design_patterns/null_object)
- Null Object Pattern in Swift (https://medium.com/@eofster/null-object-pattern-in-swift-1b96e03b2756)