The Null Object Pattern

Bobby Woolf

Knowledge Systems Corp.
4001 Weston Pkwy, Cary, NC 27513-2303
919-677-1119 x541, bwoolf@ksccary.com

NULL OBJECT — Object Structural

Intent

Provide a surrogate for another object that shares the same interface but does nothing. The Null Object encapsulates the implementation decisions of how to “do nothing” and hides those details from its collaborators.

Also Known As

Stub, Active Nothing

Motivation

Sometimes a class that requires a collaborator does not need the collaborator to do anything. However, the class wishes to treat a collaborator that does nothing the same way it treats one that actually provides behavior.

Consider for example the Model-View-Controller paradigm in Smalltalk-80. A View uses its Controller to gather input from the user. This is an example of the Strategy pattern [GHJV95, page 315], since the Controller is the View’s strategy for how it will gather input.

A view can be read-only. Since the view does not gather input from the user, it does not require a controller. Yet View and its subclasses are implemented to expect a controller and they use their controller extensively.

If no instances of the “view” class ever needed a controller, then the class would not need to be a subclass of View. It could implemented as a visual class similar to View that did not require a controller. However, this will not work for a class that has some instances which require a controller and some which do not. In that case, the class needs to be a subclass of View and all of its instances will require a controller. Thus the view class requires a controller but a particular instance does not.

One way to solve this problem would be to set the instance’s controller to nil. This would not work very well though because the view constantly sends its controller messages that only Controller understands (like isControlWanted and startUp). Since UndefinedObject (nil’s class) does not understand these Controller messages, the view would have to check its controller before sending those messages. If the controller were nil, the view would have to decide what to do. All of this conditional code would clutter the view’s implementation. If more than one view class could have nil as its controller, the conditional code for handling nil would be difficult to reuse. Thus using nil as a controller does not work very well.

Another way to solve this problem would be to use a read-only controller. Some controllers can be set in read-only mode so that they ignore input. Such a controller still gathers input, but when in read-only mode, it processes the input by doing nothing. If it were in edit mode, it would process that same input by changing the model’s state. This is overkill for a controller which is always going to be read-only. Such a controller does not need to do any processing depending on its current mode. Its mode is always read-only, so no processing is necessary. Thus a controller which is always read-only should be coded to perform no processing.

Instead, what we need is a controller that is specifically coded to be read-only. This special subclass of Controller is called NoController. It implements all of Controller's interface, but does nothing. When asked isControlWanted, it automatically answers no. When told to startUp, it automatically does nothing and returns self. It does everything a controller does, but it does so by doing nothing.

```
View    ------> Controller
              isControlWanted
                  startUp
          __________|_________  . . .
          |                  |
  TextController       NoController
 isControlWanted     isControlWanted------> ^false
     startUp             startUp  ------>   ^self
```

This diagram illustrates how a view requires a controller and how that controller may be a NoController. The NoController implements all of the behavior that any controller does, but it does so by doing nothing.

NoController is an example of the Null Object pattern. The Null Object pattern describes how to develop a class that encapsulates how a type of object should do nothing. Because the do nothing code is encapsulated, its complexity is hidden from the collaborator and can easily be reused by any collaborator that wants it.

The key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the abstract class' interface, it can be used any place this type of object is needed.

Applicability

Use the Null Object pattern when

- an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists.
- some collaborator instances should do nothing.
- you want clients to be able to ignore the difference between a collaborator which provides real behavior and that which does nothing. This way, the client does not have to explicitly check for nil or some other special value.
- you want to be able to reuse the do nothing behavior so that various clients which need this behavior will consistently work the same way.
- all of the behavior which might need to be do nothing behavior is encapsulated within the collaborator class. If some of the behavior in that class is do nothing behavior, most or all of the class' behavior will be do nothing. [Coplein96]

Structure

```
Client   ------>AbstractObject
                   request
          __________|__________ _ _ _
          |                   |
     RealObject          NullObject
      request             request  ------>"do nothing"
```

Participants

- Client (View)
  – requires a collaborator.
- AbstractObject (Controller)

  - declares the interface for Client's collaborator.
  - implements default behavior for the interface common to all classes, as appropriate.
- RealObject (TextController)
  - defines a concrete subclass of AbstractObject whose instances provide useful behavior that Client expects.
- NullObject (NoController)
  - provides an interface identical to AbstractObject's so that a null object can be substituted for a real object.
  - implements its interface to do nothing. What exactly it means to do nothing is subjective and depends on what sort of behavior Client is expecting. Some requests may be fulfilled by doing something which gives a null result.
  - when there is more than one way to do nothing, more than one NullObject class may be required.

Collaborations

- Clients use the AbstractObject class interface to interact with their collaborators. If the receiver is a RealObject, then the request is handled to provide real behavior. If the receiver is a NullObject, the request is handled by doing nothing or at least providing a null result.

Consequences

The Null Object pattern

- defines class hierarchies consisting of real objects and null objects. Null objects can be used in place of real objects when the object is expected to do nothing. Whenever client code expects a real object, it can also take a null object.
- makes client code simple. Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write testing code which handles the null collaborator specially.
- encapsulates the do nothing code into the null object. The do nothing code is easy to find. Its variation with the AbstractObject and RealObject classes is readily apparent. It can be efficiently coded to do nothing, rather than having to go through the motions of doing something, but ultimately doing nothing. It does not require variables that contain null values because those values can be hard-coded as constants or the do nothing code can avoid using those values altogether.
- makes the do nothing code in the null object easy to reuse. Multiple clients which all need their collaborators to do nothing will all do nothing the same way. If the do nothing behavior needs to be modified, the code can be changed in one place. Thereafter, all clients will continue to use the same do nothing behavior, which is now the modified do nothing behavior.
- makes the do nothing behavior difficult to distribute or mix into the real behavior of several collaborating objects. The same do nothing behavior cannot easily be added to several classes unless those classes all delegate the behavior to a class which can be a null object class.
- can necessitate creating a new NullObject class for every new AbstractObject class.
- can be difficult to implement if various clients do not agree on how the null object should do nothing.
- always acts as a do nothing object. The Null Object does not transform into a Real Object.

Implementation

There are several issues to consider when implementing the Null Object pattern:

1. *Null Object as Singleton.* The Null Object class is often implemented as a Singleton [GHJV95, page 127]. Since a null object usually does not have any state, its state can't change, so multiple instances are identical. Rather than use multiple identical instances, the system can just use a single instance repeatedly.
2. *Special null instance of Real Object.* As mentioned in the Consequences, the Null Object pattern can cause a single Real Object class to explode into three classes: AbstractObject, RealObject, and NullObject. Thus even if the entire abstract object hierarchy can be implemented with one RealObject class (and no subclasses), at least one subclass is required to implement the NullObject class. One way to avoid this class explosion is to implement the null object as a special instance of RealObject rather than as a subclass of AbstractObject. The variables in this null instance would have null values. This may be sufficient to cause the null instance to do nothing. For example, a composite object whose children is an empty list acts like a leaf object.
3. *Clients don't agree on null behavior.* If some clients expect the null object to do nothing one way and some another, multiple NullObject classes will be required. If the do nothing behavior must be customized at run time, the NullObject class will require pluggable variables so that the client can specify how the null object should do nothing (see the discussion of pluggable adaptors in the Adapter pattern [GHJV95, page 142]). Again, a way to avoid this explosion of NullObject subclasses of a single AbstractObject class is to make the null objects special instances of RealObject or a single NullObject subclass. If a single NullObject class is used, its implementation can become an example of the Flyweight pattern [GHJV95, page 195]. The behavior which all clients expect of a particular null object becomes the flyweight's intrinsic behavior and that which each client customizes is the flyweight's extrinsic behavior.
4. *Transformation to Real Object.* A Null Object does not transform to become a Real Object. If the object may decide to stop providing do nothing behavior and start providing real behavior, it is not a null object. It may be a real object with a do nothing mode, such as a controller which can switch in and out of read-only mode. If it is a single object which must mutate from a do nothing object to a real one, it should be implemented with the Proxy pattern [GHJV95, page 207]. Perhaps the proxy will start off using a null object, then switch to using a real object, or perhaps the do nothing behavior is implemented in the proxy for when it doesn't have a subject. The proxy is not required if the client is aware that it may be using a null collaborator. In this case, the client can take responsibility for swapping the null object for a real one when necessary.
5. *Null Object is not Proxy.* The use of a null object can be similar to that of a Proxy [GHJV95, page 207], but the two patterns have different purposes. A proxy provides a level of indirection when accessing a real subject, thus controlling access to the subject. A null collaborator does not hide a real object and control access to it, it replaces the real object. A proxy may eventually mutate to start acting like a real subject. A null object will not mutate to start providing real behavior, it will always provide do nothing behavior.
6. *Null Object as special Strategy*. A Null Object can be a special case of the Strategy pattern [GHJV95, page 315]. Strategy specifies several ConcreteStrategy classes as different approaches for accomplishing a task. If one of those approaches is to consistently do nothing, that ConcreteStrategy is a NullObject. For example, a Controller is a View's Strategy for handling input, and NoController is the Strategy that ignores all input.
7. *Null Object as special State*. A Null Object can be a special case of the State pattern [GHJV95, page 305]. Normally, each ConcreteState has some do nothing methods because they're not appropriate for that state. In fact, a given method is often implemented to do something useful in most states but to do nothing in at least one state. If a particular ConcreteState implements most of its methods to do nothing or at least give null results, it becomes a do nothing state and as such

is a Null Object. For example, the state that represents a user who is not logged in allows the user to do nothing but log in, so it is a null state. [Wallingford96]

8. *The Null Object class is not a mixin.* Null Object is a concrete collaborator class that acts as the collaborator for a client which needs one. The null behavior is not designed to be mixed into an object that needs some do nothing behavior. It is designed for a class which delegates to a collaborator all of the behavior that may or may not be do nothing behavior. [Coplein96]

Sample Code

For an example implementation of the Null Object pattern, let's look at the implementation of the NullScope class in VisualWorks Smalltalk (described in the Known Uses).

NullScope is a special class in the NameScope hierarchy. A NameScope knows how to search for a variable with a particular name (variableAt:from:), an undeclared variable (undeclared:from:), and iterate through its variables (namesAndValuesDo:). (The way these messages get passed from one scope to the next is an example of the Chain of Responsibility pattern [GHJV95, page 223].)

```
Object ()
  NameScope (outerScope)

NameScope>>variableAt:from:
      ^self subclassResponsibility

NameScope>>undeclared:from:
      ^outerScope undeclared: name from: varNode

NameScope>>namesAndValuesDo:
      self subclassResponsibility
```

A StaticScope represents the scope for class and global variables. A LocalScope represents the scope for instance variables and method variables. They implement variableAt:from: and namesAndValuesDo: in a pretty straightforward manner that is essentially the same for both classes.

```
Object ()
  NameScope (outerScope)
    LocalScope (...)
    StaticScope (...)

LocalScope>>variableAt:from:
      "find and return the variable, or"
      ^outerScope variableAt: name from: varNode

LocalScope>>namesAndValuesDo:
      "iterate through the receiver's variables, then"
      outerScope namesAndValuesDo: aBlock
```

A NullScope represents the outermost scope. This is either the most global scope's outer scope or the outer scope for a clean or copy block (a block that does not have an outer context). It inherits the instance variable outerScope but never uses it. It never contains the declarations for any variables, so its code is pretty simple.

```
Object ()
  NameScope (outerScope)
    LocalScope (...)
    NullScope ()
    StaticScope (...)

NullScope>>variableAt:from:
      "There are no definitions here."
      ^nil

NullScope>>namesAndValuesDo:
      "Do nothing"
```

What is most interesting about NullScope is how it implements undeclared:from:. NameScope just passes the request to its outer scope. StaticScope and LocalScope inherit this implementation. So none of those classes do anything. But NullScope implements the method to return the variable from the dictionary of undeclared variables.

```
NullScope>>undeclared:from:
      "Find the variable in Undeclared and return it.
      If the variable is not in Undeclared, add it
      and return it."
```

This is how variables become undeclared: If variableAt:from: fails to find the variable in any of the scopes, the client calls undeclared:from: to find it in Undeclared. If it isn't already in Undeclared, it gets added. The hierarchy should encapsulate this decision and hide it from the client by implementing NullScope>>variableAt:from: to send undeclared:from:, but the NullScope can't do that without knowing the scope that the search originally started from.

Notice how NullScope factors the special code out of the real NameScope classes (StaticScope and LocalScope) and encapsulates them into NullScope. This avoids special tests, makes the difference between the general behavior (in NameScope) and the special behavior (in NullScope) easy to see, and makes the special behavior easy to reuse.

Known Uses

NoController

NoController, the Null Object class in the motivating example, is a class in the Controller hierarchy of VisualWorks Smalltalk. [VW95]

NullDragMode

NullDragMode is a class in the DragMode hierarchy in VisualWorks Smalltalk. A DragMode is used to implement placement and dragging of visuals in the window painter. Subclasses represent different ways that dragging that can be done. (The DragMode hierarchy is an example of the Strategy pattern [GHJV95, page 315].) For example, an instance of CornerDragMode represents that one of the visual's resize handles is being dragged, so the visual should stay in the same place but its size should change. Alternatively, a SelectionDragMode means that the entire visual is being dragged, so its size should remain fixed but its position should follow the mouse.

A NullDragMode is a counterpart to CornerDragMode that represents an attempt to resize a visual that cannot be resized (such as a text label, whose fixed size is determined by the characters it contains and their font size). The various drag modes implement a method, dragObject:startingAt:inController:, which processes the dragging motion of the mouse. It uses a block to control how the dragging is done. In NullDragMode, this method uses an empty block that does nothing. Thus a NullDragMode responds to the mouse's drag motions by doing nothing. [VW95]

NullInputManager

NullInputManager is a class in the InputManager hierarchy is VisualWorks Smalltalk. An InputManager provides a platform-neutral, object interface to platform events that may affect the handling of internationalized (foreign language) input. (Since it wraps the platform resources to give them a standard object interface, this is an example of the Adapter pattern [GHJV95, page 142].) Subclasses such as X11InputManager represent specific platforms. NullInputManager represents platforms which don't support internationalization. The methods it implements do little if anything whereas their counterparts is X11InputManager do real work. [VW95]

NullScope

NullScope is a class in the NameScope hierarchy is VisualWorks Smalltalk. A NameScope represents the scope of a particular set of variables. What kind of variable it is (global, class level, or method level) defines what kind of NameScope it will use. For example, a StaticScope is assigned to global and class variables and a LocalScope is assigned to instance and temporary variables. Every scope has an outer scope. This is used to access variables whose scope is greater than the current level. It allows the compiler to warn the programmer if he is declaring a variable with the same name as another variable which has already been declared (usually in an outer scope). Thus NameScopes form a tree, with the global scope at the root and branches for class scopes that contain branches for method scopes.

However, since all scopes have an outer scope, what is the global scope's outer scope? It is a NullScope, a scope which never contains any variables. When looking for a variable declaration, each scope keeps looking in its outer scope until either it finds the declaration or until it hits a NullScope. NullScope knows to stop the search and answer that the variable apparently has not been declared (within the scope of the code that initiated the search). This could be handled as a special case in StaticScope, that if it is the global scope, then it should expect its outer scope to be nil, but the special case is coded more cleanly in the special class NullScope. This allows NullScope to be reused by clean and copy blocks, ones which are so simple that they have no outer scope. NullScope is implemented as a Singleton because the system never needs more than one instance [GHJV95, page 127]. [VW95]

NullLayoutManager

The LayoutManager hierarchy in the Java AWT toolkit does not have a null object class but could use one such as NullLayout. A Container can be assigned a LayoutManager (an example of the Strategy pattern [GHJV95, page 315]). If a particular Container does not require a LayoutManager, the variable can be set to nil. Unfortunately, this means that Container's code is cluttered with lots of checks for a nil LayoutManager. Container's code would be simpler if it used a null object like NullLayoutManager instead of nil. [Gamma96]

Null_Mutex

The Null_Mutex class is a mutual exclusion mechanism in the ASX (ADAPTIVE Service eXecutive) framework implemented in C++. The framework provides several mechanisms (e.g., Strategies [GHJV95, page 315]) for concurrency control. The Mutex class defines a non-recursive lock for a thread that will not call itself while the lock is established. The RW_Mutex class defines a lock that allows multiple simultaneous threads for reading but only one thread during a write. The Null_Mutex class defines a lock for a service that is always run in a single thread and does not contend with other threads. Since locking is not really necessary, Null_Mutex doesn't really lock anything; its acquire and release methods do nothing. This avoids the overhead of acquiring locks when they're not really needed. [Schmidt94]

Null Lock

Null Lock is a type of lock mode (e.g., State [GHJV95, page 305]) in VERSANT Object Database Management System. Three of the lock modes VERSANT uses are write lock, read lock, and null lock. Write lock blocks other write locks and read locks on the same object so that no one else can read or change the object while you're changing it. Read lock blocks write locks but allows other read locks so that other people can read the object while you're reading it but they can't change it.

Null lock does not block other locks and cannot be blocked by other locks. Thus it guarantees you immediate access to the object, even if someone else has already locked it, but it does not guarantee you that the object will be in a consistent state when you access it. Null lock is not really a lock because it doesn't perform any locking, but it acts like a lock for operations that require some type of lock. [Versant95]

NullIterator

The Iterator pattern documents a special case called NullIterator [GHJV95, pages 67-68 and 262]. Each node in a tree might have an iterator for its children. Composite nodes would return a concrete iterator, but leaf nodes would return an instance of NullIterator. A NullIterator is always done with traversal; when asked isDone, it always returns true. In this way a client can always use an iterator to iterate over the nodes in a structure even when there are no more nodes.

Z-Node

Procedural languages have null data types that are like null objects. Sedgewick's *z*-node is a dummy node that is used as the last node in a linked list. When a tree node requires a fixed number of child nodes but does not have enough children, he uses *z*-nodes as substitutes for the missing children. In a list, the *z*-node protects the delete procedure from needing a special test for deleting an item from an empty list. In a binary tree, a node without two children would need one or two null links, but the null *z*-node is used instead. This way a search algorithm can simply skip *z*-node branches; when it has run out of non- *z*-node branches, it knows the search did not find the item. In this way, *z*-nodes are used to avoid special tests the way null objects are. [Sedge88]

NULL Handler

The Decoupled Reference pattern shows how to access objects via Handlers so that their true location is hidden from the client. When a client requests an object that is no longer available, rather than let the program crash, the framework returns a NULL Handler. This Handler acts like other Handlers but fulfills requests by raising exceptions or causing error conditions. [Weibel96]

Related Patterns

The NullObject class can usually be implemented as a Singleton [GHJV95, page 127] since multiple instances would act exactly the same and have no internal state that could change.

When multiple null objects are implemented as instances of a single NullObject class, they can be implemented as Flyweights [GHJV95, page 195].

NullObject is often used as one class in a hierarchy of Strategy classes [GHJV95, page 315]. It represents the strategy to do nothing.

NullObject is often used as one class in a hierarchy of State classes [GHJV95, page 305]. It represents the state in which the client should do nothing.

NullObject can be a special kind of Iterator [GHJV95, page 257] which doesn't iterate over anything.

NullObject may be a special class in a hierarchy of Adapters [GHJV95, page 142]. Whereas an adapter normally wraps another object and converts its interface, a null adapter would pretend to wrap another object without actually wrapping anything.

Bruce Anderson has also written about the Null Object pattern, which he also refers to as "Active Nothing." [Anderson95]

NullObject is a special case of the Exceptional Value pattern in The CHECKS Pattern Language [Cunningham95]. An Exceptional Value is a special Whole Value (another pattern) used to represent exceptional circumstances. It will either absorb all messages or produce Meaningless Behavior (another pattern). A NullObject is one such Exceptional Value.

References

[Anderson95] Anderson, Bruce. "Null Object." UIUC patterns discussion mailing list (patterns@cs.uiuc.edu), January 1995.

[Coplein96] Coplein, James. E-mail correspondence.

[Cunningham95] Ward Cunningham, "The CHECKS Pattern Language of Information Integrity" in [PLoP95].

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995; http://www.aw.com/cp/Gamma.html.

[Gamma96] Gamma, Erich. E-mail correspondence.

[PLoP95] Coplien, James and Douglas Schmidt (editors). *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995; http://heg-school.aw.com/cseng/authors/coplien/patternlang/patternlang.html.

[Schmidt94] Schmidt, Douglas. “Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application,” *C++ Report*. SIGS Publications, Vol. 6, No. 3, July 1994.

[Sedge88] Sedgewick, Robert. *Algorithms*. Addison-Wesley, Reading, MA, 1988.

[Versant95] *VERSANT Concepts and Usage Manual*. Versant Object Technology, Menlo Park, CA, 1995.

[Wallingford96] Eugene Wallingford. E-mail correspondence.

[Weibel96] Weibel, Peter. “The Decoupled Reference Pattern.” Submitted to EuroPLoP ’96.

[VW95] VisualWorks Release 2.5, ParcPlace-Digitalk, Inc., Sunnyvale, CA, 1995; http://www.parcplace.com.