# Chapter 1

# Null Object

*Bobby Woolf*

## Intent

A Null Object provides a surrogate for another object that shares the same interface but does nothing. Thus, the Null Object encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators.

## Also Known As

Active Nothing [Anderson95]

## Motivation

Sometimes an object that requires a collaborator does not need the collaborator to do anything. However, the object 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 framework in Smalltalk-80 and VisualWorks Smalltalk. A view uses its controller to gather input from the user. This is a Strategy, since the controller is the view's strategy for how it will gather input [Gamma+95, p. 315].

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 be implemented as a visual class similar to `View` that did not require a controller. However, this would not work for a class that has some instances that require a controller and some that do not. In that case, the class would need to be a subclass of `View`, and all of its instances would require a controller. Thus the `View` class requires a controller, but a particular instance does not.

A common 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 messages to its controller that only `Controller` understands (like `isControlWanted`, `isControlActive`, and `startUp`). Since `UndefinedObject` (`nil`'s class) does not understand these `Controller` messages, the view would have to check its controller before sending such 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.

For example, this is how `VisualPart` (the topmost class in the `View` hierarchy) would have to implement `objectWantingControl`, one of its key messages

```
VisualPart>>objectWantingControl
      ...
      ^ctrl isNil ifFalse:
            [ctrl isControlWanted
                  ifTrue: [self]
                  ifFalse: [nil]]
```

Note the check that `ctrl` is not `false` before sending it a controller message like `isControlWanted`. This simple check causes a number of problems. First, not only does this check clutter the code, it obscures its behavior. What does this implementation return when the controller is `nil`? (The method returns `nil`, which works, but is it what the author intended?) Second, there are several views and controllers that send `isControlWanted` to an unknown controller. They would all have to check for `nil` and handle that case not only appropriately but also consistently. Third, if a developer implements new code that sends `isControlWanted` to a controller, he will need to add this check. If he forgets, his code may have a very subtle failure built into it. Finally, not only would most senders of `isControlWanted` have to be protected, but so would most senders of other key controller messages, like `isControlActive` and `startUp`.

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` (see Figure 1-1). It

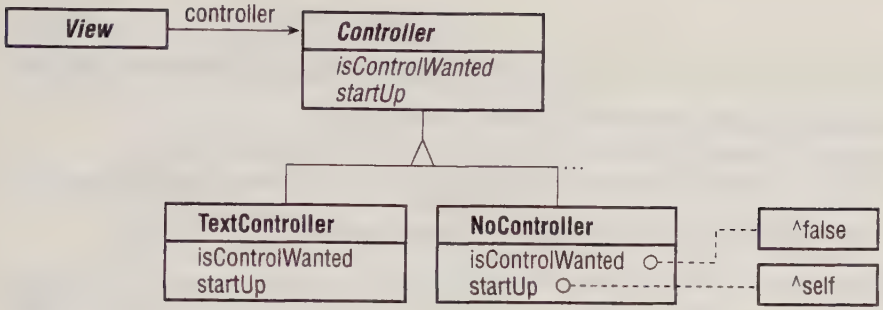


**Figure 1-1** *A null controller*

implements all of `Controller`'s interface, but does nothing. When asked `isControlWanted`, it automatically answers `false`. When told to `startUp`, it automatically does nothing and returns `self`. It does everything a controller does, but it does so by doing nothing.

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.

For example, this is how `VisualPart>>objectWantingControl`, shown in Figure 1-1, is really implemented.

```
VisualPart>>objectWantingControl
      ...
      ^ctrl isControlWanted ifTrue: [self] ifFalse: [nil]
```

If the controller is a real `Controller`, such as a `TextController`, then `isControlWanted` works normally. But if the controller is a `NoController`, `isControlWanted` automatically returns `false`; `objectWantingControl` works correctly without having to know whether its controller is a real `controller` or a null one. Any other sender of `isControlWanted`, or `isControlActive`, or `startUp` can also send these messages without knowing whether the controller is real or null. Whenever the controller is a `NoController`, all senders will behave consistently because `NoController` encapsulates the do-nothing behavior.

`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 be easily 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 interface of the abstract class, it can be used any place this type of object is needed.

## Keys

A framework that incorporates the Null Object pattern has the following features.

- A type whose classes provide the desired behavior
- An object of the same type that fulfills its interface by doing nothing
- A client class that collaborates with this type and can use any of its instances

Usually, such a framework also has the following features.

- A separate class that implements the null object
- An abstract superclass that defines the interface for the regular classes and the null one
- The null class typically has no state and may only have a single instance

## Applicability

Use the Null Object pattern

- When an object requires a collaborator. This collaboration already exists before the Null Object pattern is applied.
- When some collaborator instances should do nothing.
- When you want clients to be able to ignore the difference between a collaborator that provides real behavior and one that does nothing. This way, the client does not have to explicitly check for `nil` or some other special value.
- When you want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work the same way.
- When all of the behavior that 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 behavior of the class will be do-nothing [Coplien96].

The primary alternative to the Null Object pattern is to use `nil`. Use a variable set to `nil` in these instances.

- When very little code actually uses the variable directly.
- When the code that does use the variable is well encapsulated—at least within one class—so that it will hide the `nil` variable. Because the code using the variable is all encapsulated in one place, it is easily consistent with itself, does not have to be consistent with any other code, and will not need to be reused.
- When the code that uses the variable can easily decide how to handle the `nil` case and will always handle it the same way.

## Structure

An example of the Null Object pattern typically has the structure shown in Figure 1-2.

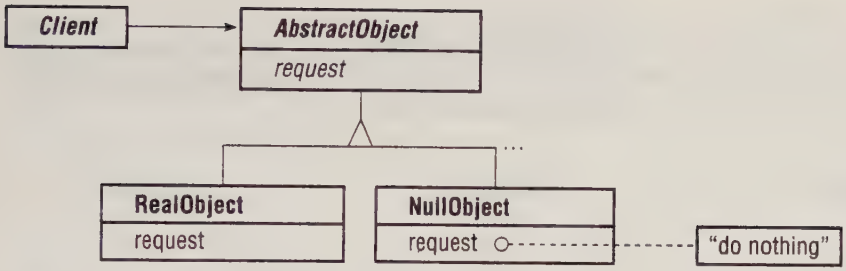


**Figure 1-2** *Structure of the Null Object pattern*

## Participants

- Client (`view`):
  - requires a collaborator with a specific interface.
- AbstractObject (`Controller`):
  - defines a type and 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 `NullObject` can be polymorphically substituted for a `RealObject`.
  - implements its interface to do nothing. Exactly what "do nothing" means is subjective and depends on the sort of behavior the `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.

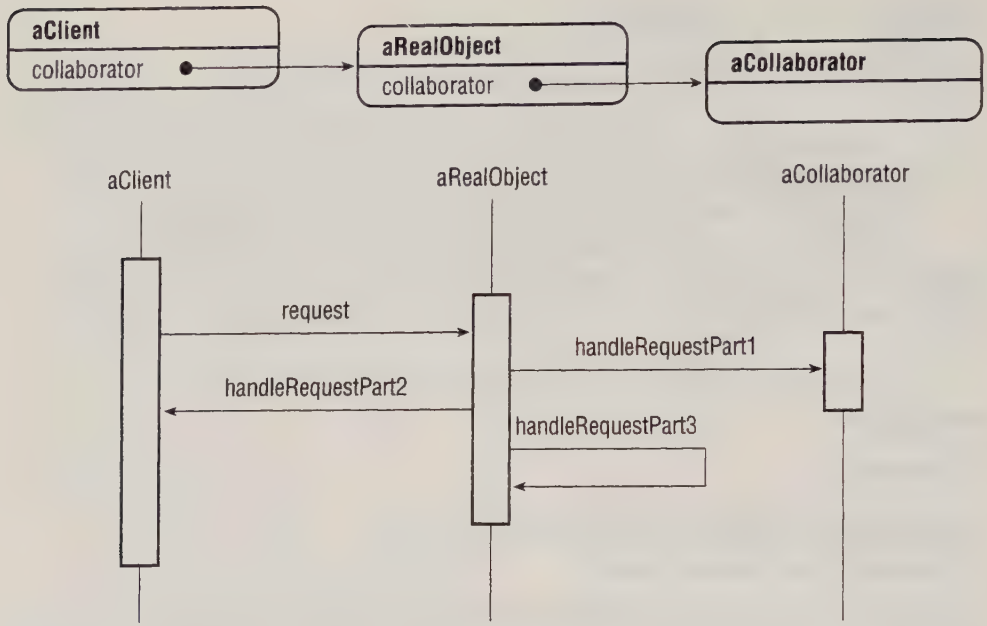


**Figure 1-3** *A client collaborating with a real object*

Figure 1-3 shows how `aRealObject` handles a request. When `aClient`'s collaborator is `aRealObject` and it sends the collaborator a message such as `request`, the real object implements the message with real behavior by sending more messages to other collaborators, even back to the client and to itself. This is a routine collaboration between objects.

On the other hand, as shown in Figure 1-4, when the same client's collaborator is `aNullObject`, not much happens. In this case, when `aClient` sends the collaborator a `request` message, the null object implements the message by doing nothing. This is an unusual collaboration because the null object never does much of anything.

## Consequences

The advantages of the Null Object pattern are that it

- ***Uses polymorphic classes.*** The 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.
- ***Simplifies client code.*** Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know (and shouldn't care) whether they're

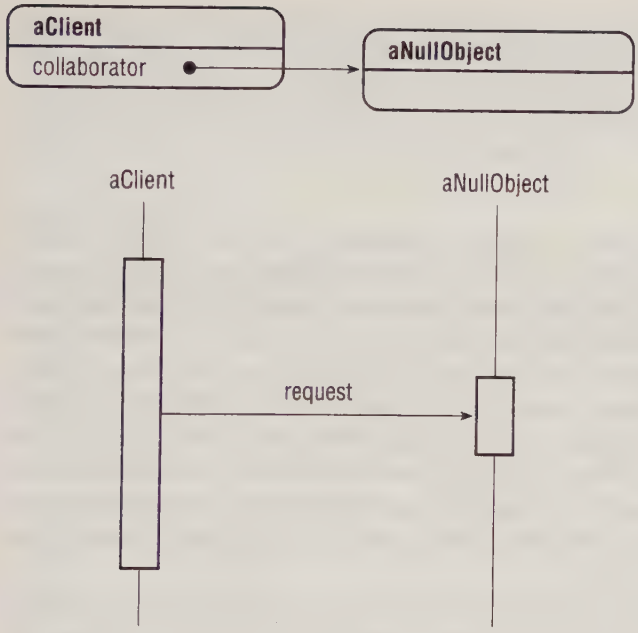


**Figure 1-4** *A client collaborating with a null object*

dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write special testing code to handle the null collaborator.

- ***Encapsulates do nothing behavior.*** The do-nothing code is easy to find. Its variation from 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 do-nothing behavior reusable.*** Multiple clients which all need their collaborators to do nothing can reuse the `NullObject` class so that they all do nothing the same way. If the do-nothing behavior needs to be modified, the code can be changed in one place, and the various clients will still behave consistently.

The disadvantages of the Null Object pattern are that it

- ***Forces encapsulation.*** The pattern 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 collaborator that can be a null object class.
- ***May cause class explosion.*** The pattern can necessitate creating a new `NullObject` class for each new `AbstractObject` class.

- *Forces uniformity.* The pattern can be difficult to implement if various clients do not agree on how the null object should do nothing.
- *Is non-mutable.* A `NullObject` always acts as a do-nothing object. It does not transform into a `RealObject`.

## Implementation

There are several issues to consider when implementing the Null Object pattern. One is the special null instance of `RealObject`. As mentioned in the Consequences, the Null Object pattern can cause a single `RealObject` 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, see `NullTimeZone` in the Known Uses section.

Second, 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 are required. If the do-nothing behavior must, then, 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 [Gamma+95, p. 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 also be a Flyweight [Gamma+95, p. 195]. The behavior that 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.

A third issue is transformation to `RealObject`. A `NullObject` cannot be transformed to become a `RealObject`. 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 [Gamma+95, p. 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.

Finally, the `NullObject` class is not a mixin. `NullObject` 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 [Coplien96].

## Sample Code

For an example implementation of the Null Object pattern, look at the implementation of the `NoController` class in VisualWorks Smalltalk (described in the Motivation section). `NoController` is a special class in the `Controller` hierarchy. A `Controller`, part of the Model-View-Controller architecture in Smalltalk-80 and VisualWorks Smalltalk, handles input for a `View`. Container visuals can pass control to their component visuals' `controllers`, which in turn can pass control to their component visuals' `controllers`. This passing of control from a `controller` to a subcontroller is an example of Chain of Responsibility [Gamma+95, p. 223].

Visuals (Views) form a tree. Since most visuals have controllers, controllers form a tree indirectly via the visual tree. One responsibility of this tree of controllers is to determine which one should have control. `VisualPart>>objectWantingControl` looks for a visual whose controller wants control. Then the system gives control to the controller in `VisualPart>>startUp`. Finally, in `Controller>>controlLoop`, the controller keeps control until it no longer wants control.

```
Object ()
  Controller (model view ...)
  VisualPart (container)

VisualPart>>objectWantingControl
      ...
      ^ctrl isControlWanted ifTrue: [self] ifFalse: [nil]

VisualPart>>startUp
      | ctrl |
      ctrl := self getController.
      ^ctrl notNil
            ifTrue: [ctrl startUp]
            ifFalse: [nil]

Controller>>controlLoop
      [...
      self isControlActive]
            whileTrue:
                  [...]
```

So when a controller wants control, `isControlWanted` returns true. To give control to a controller, its view sends it `startUp`. Finally, when a controller has control and wants to keep it, `isControlActive` returns `true`.

How does a controller decide when it wants to get and keep control? Essentially, if a controller's view contains the mouse cursor, the controller wants control. Once it has control, it runs its `controlLoop` until it doesn't want control anymore.

These main controller methods—`isControlWanted`, `startUp`, and `isControlActive`—are implemented in `Controller`. Some subclasses enhance these implementations, but few override them completely.

```
Controller>>isControlActive
      ^self viewHasCursor and: [...]

Controller>>startUp
      self controlInitialize.
      self controlLoop.
      self controlTerminate

Controller>>isControlWanted
      ^self viewHasCursor
```

A `NoController` is a `controller` that never wants control. When asked if it wants control (`isControlWanted`), it responds with `false`. When offered control (`startUp`), it does nothing. And when asked if it wants to keep control (`isControlActive`), it responds with `false` again.

```
Object ()
  Controller (model view ...)
    NoController ()
  VisualPart (container)

NoController>>isControlWanted
      ^false

NoController>>startUp
      ^self

NoController>>isControlActive
      ^false
```

By overriding just these three methods, `NoController` can inherit the two dozen messages in `Controller` and implement them to do nothing. Meanwhile, `NoController` has the interface that a `View` expects, so the `View` does not have to check for a special case like `nil`. Instead, the visual can ask questions like

`isControlWanted` and be assured that the controller will respond successfully. Because the do-nothing code is encapsulated in `NoController`, all views with null controllers will behave the same.

## Known Uses

The Null Object pattern is very commonly used, so there are many examples of it.

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

`NullDragMode` is a class in the `DragMode` hierarchy in VisualWorks Smalltalk. A `DragMode` implements the dragging of widgets in the window painter. Subclasses represent different ways that the dragging can be done. `CornerDragMode` lets the user resize the visual, and `SelectionDragMode` lets him move it. A `NullDragMode` represents an attempt to resize a visual that cannot be resized. It responds to the mouse's drag motions by doing nothing [VW95].

**Null Adapters.** `NullInputManager` is a class in the `InputManager` hierarchy in VisualWorks Smalltalk. An `InputManager` provides a platform-neutral object interface to platform events that affect internationalized input. Subclasses such as `X11InputManager` represent specific platforms. `NullInputManager` represents platforms that don't support internationalization. Its methods do little if anything, whereas their counterparts in `X11InputManager` do real work [VW95].

**Null States.** `Null_Mutex` is a mutual exclusion mechanism in the ASX (ADAPTIVE Service eXecutive) framework for C++. The framework provides several mechanisms for concurrency control. `Mutex` defines a nonrecursive lock for a thread that will not call itself. A `RW_Mutex` lock allows multiple threads to read but only one to write. `Null_Mutex` 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 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 is a type of lock mode in the VERSANT Object Database Management System. The database locks objects so that they can be read and written. A read lock allows other read locks, but blocks write locks. A write lock blocks all other read and write locks. Null Lock does not block other locks and cannot be blocked by other locks. This guarantees the user immediate access to the object, even if another process has already locked it, but it does not guarantee that the object is in a consistent state. Null Lock is not really a lock because it doesn't perform any locking, but it acts like a lock for operations that require one [Versant95].

**Null Proxies.** 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 otherwise causing error conditions [Weibel96].

**Null Iterator.** The Iterator pattern documents a special case called `NullIterator` [Gamma+95, pp. 67–68, 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 finished 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. See z-node under Procedural Nulls below.

**Null Instances.** `NullTimeZone` is a special instance of the `TimeZone` class in VisualWorks Smalltalk. A `TimeZone` converts clock times between Greenwich Mean Time (GMT) and local time and accounts for daylight savings time (DST). This is useful for UNIX machines whose clocks are set to GMT, but unnecessary in Windows and Macintosh machines because their clocks are already set to local time. On these machines, VisualWorks uses `NullTimeZone`, a special instance (not subclass) of `TimeZone` that converts the clock by doing no conversion. All of its variables are set to zero so that all of the offsets it adds to the local clock don't actually change the clock value. This is simpler than checking for a `TimeZone` which is `nil` or testing the platform to determine if it needs a `TimeZone` [VW95].

**Reusable Nulls.** `NullScope` is a class in the `NameScope` hierarchy in VisualWorks Smalltalk. A `NameScope` represents the scope of a particular set of variables. `StaticScopes` hold global and class variables whereas `LocalScopes` hold instance and temporary variables. Every scope has an outer scope. They form a tree that defines all variables in a system. Even the global scope has an outer scope, a `NullScope` that never contains any variables. When the lookup for a variable reaches a `NullScope`, it automatically answers that the variable is not defined within the code's scope. This could be handled as a special case in the `StaticScope` that holds global variables, but `NullScope` handles it more cleanly. This allows `NullScope` to be reused by clean and copy blocks, simple blocks that have no outer scope. `NullScope` is also a Singleton [Gamma+95, p. 127] as well as a Null Object [VW95].

**Procedural Nulls.** 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, Sedgewick 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 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. (This is similar to a null iterator, discussed above.) In this way, z-nodes are used to avoid special tests the way null objects are [Sedge88].

**Null Object Anti-Example.** The `LayoutManager` hierarchy in the Java AWT toolkit does *not* have a null object class but could use a class such as `NullLayoutManager`. A `Container` can be assigned a `LayoutManager` (a Strategy [Gamma+95, p. 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].

## Related Patterns

When a null object does not require any internal state, the `NullObject` class can be implemented as a Singleton [Gamma+95, p. 127], since multiple instances would act exactly the same and their state cannot change.

When multiple null objects are implemented as instances of a single `NullObject` class, they can be implemented as Flyweights [Gamma+95, p. 195].

`NullObject` is often used as one class in a hierarchy of Strategy classes [Gamma+95, p. 315]. It represents the strategy to do nothing. It is also often used as one class in a hierarchy of State classes [Gamma+95, p. 305]. There, it represents the state in which the client should do nothing, such as when the state is unknown. `NullObject` can also be a special kind of `Iterator` [Gamma+95, p. 257], which doesn't iterate over anything.

`NullObject` may be a special class in a hierarchy of Adapters [Gamma+95, p. 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.

Null Object can be similar to Proxy [Gamma+95, p. 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 and begin acting like a real subject. A null object will not mutate to provide real behavior, it will always provide do-nothing behavior. However, a Proxy hierarchy can contain a Null Proxy subclass that pretends to be a proxy but will never have a subject. Bruce Anderson has written about the Null Object pattern, which he also refers to as "Active Nothing" [Anderson95].

Null Object is a special case of the Exceptional Value pattern in The CHECKS Pattern Language. 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 Null Object is one such Exceptional Value [Cunningham95].

## ACKNOWLEDGMENTS

I would like to thank my coworkers at KSC for their help in developing this pattern, Ward Cunningham for his shepherding help, and everyone at PLoP '96 who made suggestions for improving this paper.

## REFERENCES

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

[Coplien96] J. Coplien. E-mail correspondence.

[Cunningham95] Ward Cunningham, "The CHECKS Pattern Language of Information Integrity" in J. O. Coplien and D. C. Schmidt (eds.), *Pattern Languages of Program Design.* Reading, MA: Addison-Wesley, 1995, pp. 145–155.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, (URL: http://www.aw.com/cp/Gamma.html), 1995.

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

[PLoPD95] J. Coplien and D. Schmidt (editors). *Pattern Languages of Program Design.* Reading, MA: Addison-Wesley (URL: http://hegschool.aw.com/cseng/authors/coplien/patternlang/patternlang.html), 1995.

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

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

[Versant95] *VERSANT Concepts and Usage Manual.* Menlo Park, CA: Versant Object Technology (URL: http://www.versant.com), 1995.

[Wallingford96] E. Wallingford. E-mail correspondence.

[Weibel96] P. Weibel. "The Decoupled Reference Pattern." Submitted to EuroPLoP '96.

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

**Bobby Woolf can be reached at** bwoolf@ksccary.com.