Null Object Design Pattern

The following is a description of the Null Object Design Pattern in the style set forth by Gamma et. al. in their book *Design Patterns: Elements of reusable software*. I feel this is an important pattern which was omitted and that it has not been adequately described in the past. My conception of the Null Object pattern is roughly equivalent of those found in http://exciton.cs.rice.edu/javaresources/DesignPatterns/NullPattern.htm and Woolf “The Null Object Pattern”. My description closely follows that of Woolf.

Null Object Pattern

Intent

Provide an object as a surrogate for the lack of an object of a given type. The Null Object provides intelligent “do nothing” behavior, hiding the details from its collaborators.

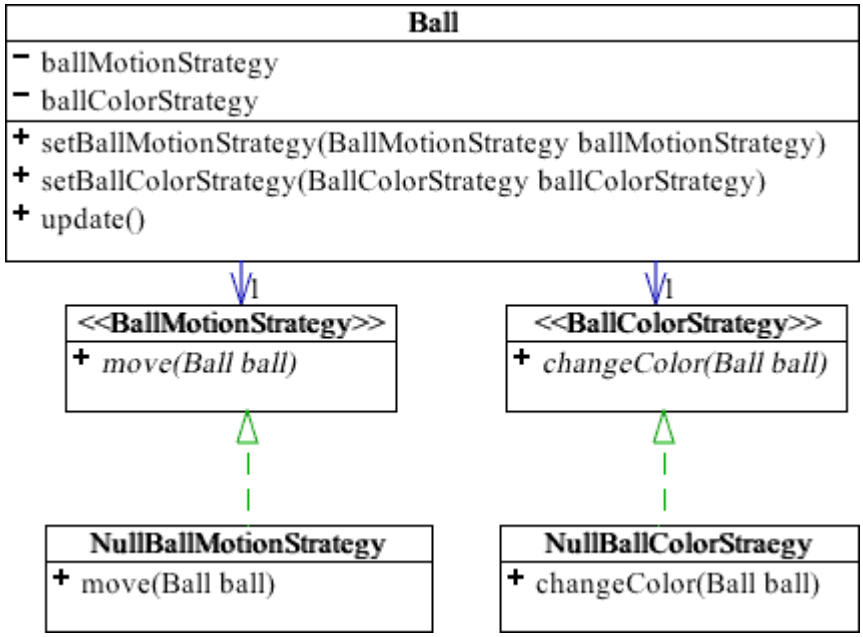
Also Known as

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 a simple screen saver which displays balls that move about the screen and have special color effects. This is easily achieved by creating a Ball class to represent the balls and using a Strategy pattern [GHJV95, page 315] to control the ball's motion and another Strategy pattern to control the ball's color. It would then be trivial to write strategies for many different types of motion and color effects and create balls with any combination of those. However, to start with you want to create the simplest strategies possible to make sure everything is working. And these strategies could also be useful later since you want as strategies as possible strategies.



Now, the simplest strategy would be no strategy. That is do nothing, don't move and don't change color. However, the Strategy pattern requires the ball to have objects which implement the strategy interfaces. This is where the Null Object pattern becomes useful. Simply implement a NullMovementStrategy which doesn't move the ball and a NullColorStrategy which doesn't change the ball's color. Both of these can probably be implemented with essentially no code. All the methods in these classes do "nothing". They are perfect examples of the Null Object Pattern.

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. As compared to using a special "null" value which doesn't actually implement the abstract interface and which must constantly be checked for with special code in any object which uses the abstract interface.

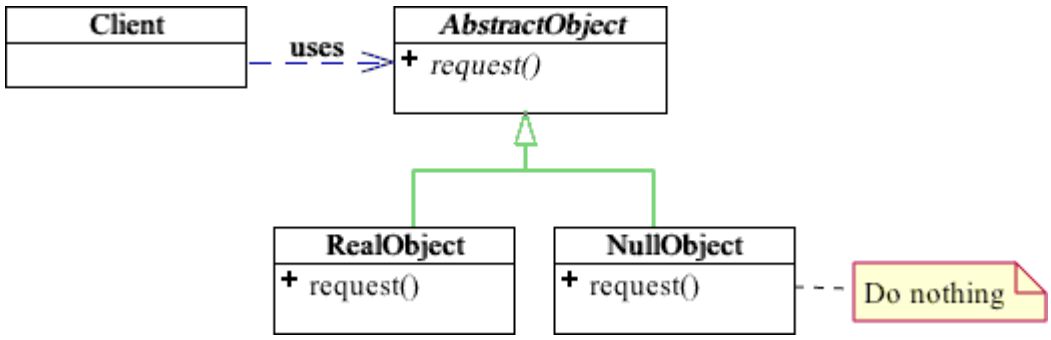
It is sometimes thought that Null Objects are over simple and "stupid" but in truth a Null Object always knows exactly what needs to be done without interacting with any other objects. So in truth it is very "smart."

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 to abstract the handling of null away from the client.

Structure



Participants

- Client
  - requires a collaborator.
- AbstractObject
  - declares the interface for Client's collaborator.
  - implements default behavior for the interface common to all classes, as appropriate.
- RealObject
  - defines a concrete subclass of AbstractObject whose instances provide useful behavior that Client expects.
- NullObject
  - 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 depends on what sort of behavior Client is expecting.
  - 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. 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 as when your AbstractObject interface is not well defined.

- 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. *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]). This may generally be a symptom of the AbstractObject not having a well defined (semantic) interface.

3. *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 State pattern [GHJV95, page 305] or perhaps the Proxy pattern [GHJV95, page 207]. In this case a Null State may be used or the proxy may hold a Null Object.

4. *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.

5. *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.

6. *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 state. [Woolf96]

7. *Null Object as Visitor host*. A Null Object can be used to allow a Visitor [GHJV95, page 331] to safely visit a hierarchy and handle the null situation.

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. [Woolf96]

Sample Code (Java)

For an example of the Null Object pattern lets look at a very simple immutable linked list.

A Linked list is either a head element and a tail which is a list or empty (i.e. Null). Thus is makes sense to model an abstract linked list with a class `List` which has two methods `getTail` and `accept` as per the Visitor pattern [GHJV95, page 331].

```
public abstract class List
{
    public abstract List getTail();

    public abstract Object accept(ListVisitor visitor, Object param);
}
```

Now we will need a class for non-empty lists, we'll call it `NonNullList`. Its implementation is fairly straight forward.

```
public class NonNullList extends List
{
    private Object head;
    private Object tail;
```

```
    /**
     * Creates a list from a head and tail. Acts as "cons"
     */
    public NonNullList(Object head, Object tail)
    {
        this.head = head;
        this.tail = tail;
    }

    // for convenience we could add a constructor taking only the head to make 1 element
    lists.

    public Object getHead()
    {
        return head;
    }

    public List getTail()
    {
        return tail;
    }

    public Object accept(ListVisitor visitor, Object param)
    {
        return visitor.whenNonNullList(this, param);
    }
}
```

Now we get to the Null Object pattern's role. Rather than using **`null`** to represent an empty list, we will create a `NullList` class to represent the empty list. Notice that it knows exactly what to do. Access to the head was intentionally left out of the abstract list because the `NullList` would be unable to fulfill that interface. However, the empty list can provide a tail (you may view it differently but I believe the rest of nothing is more nothing) and it can certainly accept a visitor. Since all empty lists are identical, we will use the Singleton design pattern [GHJV95, page 127].

```
public class NullList extends List
{
    private static final NullList instance = new NullList();

    private NullList() { }

    public static NullList Singleton()
    {
        return instance;
    }

    public List getTail()
    {
        return this;
    }

    public Object accept(ListVisitor visitor, Object param)
    {
        return visitor.whenNullList(this, param);
    }
}
```

The entire system is completed by the visitor interface.

```
public interface ListVisitor
{
```

```
    public Object whenNonNullList(NonNullList host, Object param);

    public Object whenNullList(NullList host, Object param);
}
```

Notice that only the use of the Null Object pattern makes it safe to use a visitor. Otherwise a null pointer exception would occur when one tried to call accept on an empty list.

Known Uses

See [Woolf96]

Related Patterns

Singleton [GHJV95, page 127] is often used to implement a Null Object since multiple instances would act exactly the same and have no internal state that could change.

Flyweight [GHJV95, page 195] can be used when multiple null objects are implemented as instances of a single NullObject class.

Strategy [GHJV95, page 315] patterns often have one Null Object representing the strategy of doing nothing.

State [GHJV95, page 305] often uses Null Object to represent the state in which the client should do nothing.

Iterators [GHJV95, page 257] may have Null Object as a special case which doesn't iterate over anything.

Adapters [GHJV95, page 142] may have Null Object as a special case which pretend to adapt another object without actually adapting 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.

[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.

[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.

[Woolf96] Woolf, Bobby. "The Null Object Pattern"

---

jwalker@cs.oberlin.edu