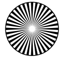
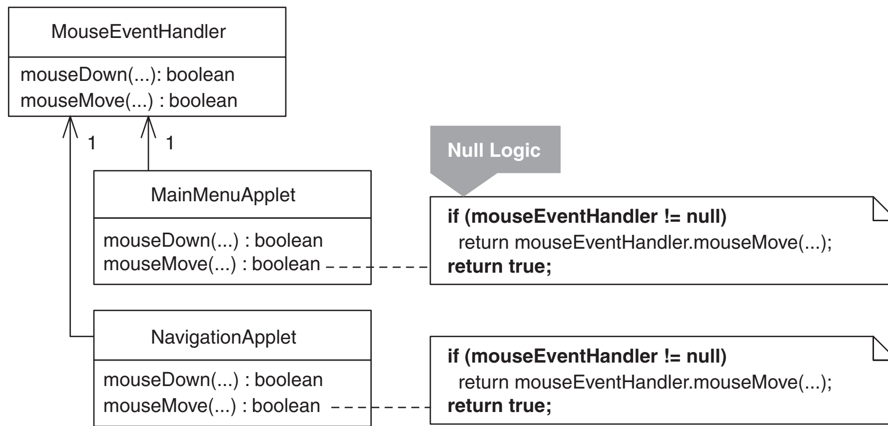
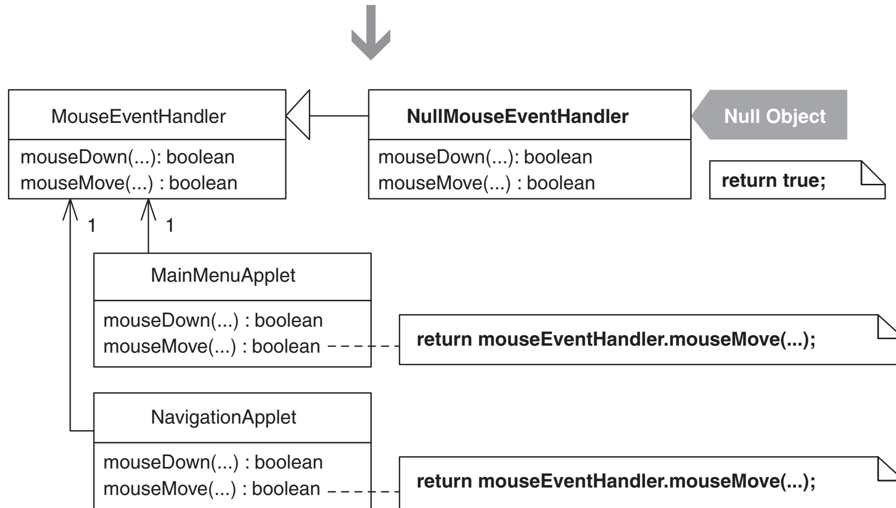


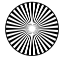
Introduce Null Object

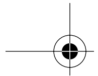
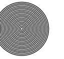
Logic for dealing with a null field or variable
is duplicated throughout your code.

Replace the null logic with a Null Object, an object that provides the appropriate null behavior.







Motivation

If a client calls a method on a field or variable that is null, an exception may be raised, a system may crash, or similar problems may occur. To protect our systems from such unwanted behavior, we write checks to prevent null fields or variables from being called and, if necessary, specify alternative behavior to execute when nulls are encountered:

```
if (someObject != null)
   someObject.doSomething();
else
   performAlternativeBehavior();
```


Repeating such null logic in one or two places in a system isn't a problem, but repeating it in multiple places bloats a system with unnecessary code. Compared with code that is free of null logic, code that is full of it generally takes longer to comprehend and requires more thinking about how to extend. Null logic also fails to provide null protection for new code. So if new code is written and programmers forget to include null logic for it, null errors can begin to occur.

The *Null Object* pattern[1] provides a solution to such problems. It removes the need to check whether a field or variable is null by making it possible to *always* call the field or variable safely. The trick is to assign the field or variable to the right object at the right time. When a field or variable can be null, you can make it refer to an instance of a Null Object, which provides do-nothing, default, or harmless behavior. Later, the field or variable can be assigned to something other than a Null Object. Until that happens, all invocations safely route through the Null Object.

The introduction of a Null Object into a system ought to shrink code size or at least keep it even. If its implementation significantly increases the number of lines of code compared with just using null logic, that's a good sign that you don't need a Null Object. Kent Beck tells a story about this in his book *Test-Driven Development* [Beck, TDD]. He once suggested refactoring to a Null Object to his programming partner, Erich Gamma, who quickly calculated the difference in lines of code and explained how refactoring to a Null Object

1. Bruce Anderson aptly named this pattern *active nothing* [Anderson] because a Null Object actively performs behavior that does nothing. Martin Fowler described how a Null Object is an example of a broader pattern called Special Case [Fowler, PEAA]. Ralph Johnson and Bobby Woolf described how null versions of patterns like Strategy [DP], Proxy [DP], Iterator [DP], and others are often used to eliminate null checks [Woolf].

Kerievsky_book.fm Page 303 Thursday, July 8, 2004 12:12 PM

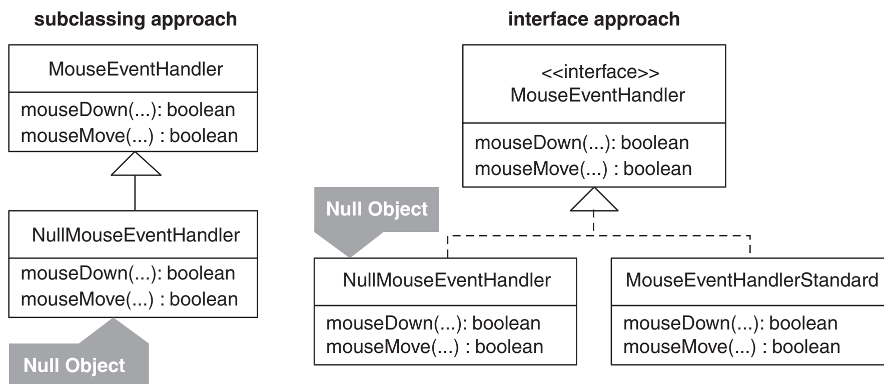
would actually add many more lines of code than they already had with their null logic.

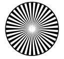
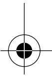
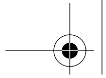
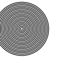
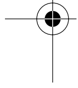
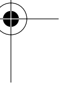
Java's Abstract Window Toolkit (AWT) could have benefited from using a Null Object. Some of its components, like panels or dialog boxes, can contain widgets that get laid out using a layout manager (like `FlowLayout`, `GridLayout`, and so forth). Code that dispatched layout requests to a layout manager was filled with checks for a null layout manager (`if (layoutManager != null)`). A better design would have been to use a `NullLayout` as the default layout manager for all components that used layout managers. If the default layout manager for these components was a `NullLayout`, the code that dispatched requests to the layout manager could have done so without caring whether it was talking to a `NullLayout` manager or a real layout manager.

The existence of a Null Object doesn't guarantee that null logic won't be written. For example, if a programmer isn't aware that a Null Object is already protecting the code from nulls, he or she may write null logic for code that won't ever be null. Finally, if a programmer expects a null to be returned under certain conditions and writes important code to handle that situation, a Null Object implementation could cause unexpected behavior.

My version of this refactoring extends Martin Fowler's *Introduce Null Object* [F] by supplying mechanics to deal with a common situation: a class is sprinkled with null logic for a field because an instance of the class may attempt to use the field before the field has been assigned to a non-null value. Given such code, the mechanics to refactor to a Null Object are different from those defined in Martin's mechanics for this refactoring.

Null Objects are often (though not always) implemented by subclassing or by implementing an interface, as shown in the following diagram.



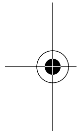
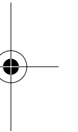







Kerievsky_book.fm Page 304 Thursday, July 8, 2004 12:12 PM

Creating a Null Object by subclassing involves overriding all inherited public methods to provide the appropriate null behavior. A risk associated with this approach is that if a new method gets added to the superclass, programmers must remember to override the method with null behavior in the Null Object. If they forget to do this, the Null Object will inherit implementation logic that could cause unwanted behavior at runtime. Making a Null Object implement an interface rather than being a subclass removes this risk.

Benefits and Liabilities

+ Prevents null errors without duplicating null logic.
+ Simplifies code by minimizing null tests.
– Complicates a design when a system needs few null tests.
– Can yield redundant null tests if programmers are unaware of a Null Object implementation.
– Complicates maintenance. Null Objects that have a superclass must override all newly inherited public methods.

Mechanics

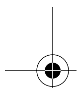
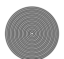
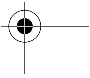
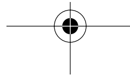
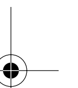
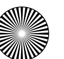
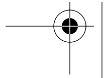
These mechanics assume you have the same null logic scattered throughout your code because a field or local variable may be referenced when it is still null. If your null logic exists for any other reason, consider applying Martin Fowler's mechanics for *Introduce Null Object* [F]. The term *source class* in the following steps refers to the class that you'd like to protect from nulls.

1. Create a *null object* by applying *Extract Subclass* [F] on the source class or by making your new class implement the interface implemented by the source class. If you decide to make your null object implement an interface, but that interface doesn't yet exist, create it by applying *Extract Interface* [F] on the source class.

   ✔ Compile.

2. Look for a *null check* (client code that invokes a method on an instance of the source class if it is not null, or performs alternative behavior if it is null). Overide the invoked method in the null object so it implements the alternative behavior.

   ✔ Compile.

Kerievsky_book.fm Page 305 Thursday, July 8, 2004 12:12 PM

3. Repeat step 2 for other null checks associated with the source class.
4. Find a class that contains one or more occurrences of the null check and initialize the field or local variable that is referenced in the null check to an instance of the null object. Perform this initialization at the earliest possible time during the lifetime of an instance of the class (e.g., upon instantiation).

   This code should not affect pre-existing code that assigns the field or local variable to an instance of the source class. The new code simply performs an assignment to a null object prior to any other assignments.

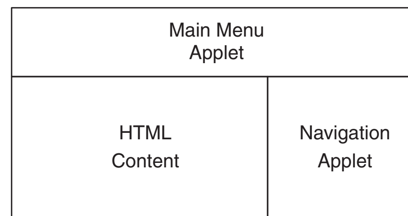
   ✔ Compile.
5. In the class you selected in step 4, remove every occurrence of the null check.

   ✔ Compile and test.
6. Repeat steps 4 and 5 for every class with one or more occurrences of the null check.

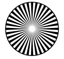
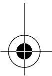
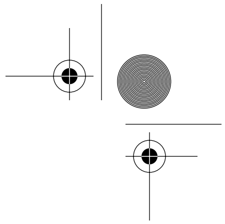
If your system creates many instances of the null object, you may want to use a profiler to determine whether it would make sense to apply *Limit Instantiation with Singleton (296)*.

Example

At a time when most of the world used either Netscape 2 or 3 or Internet Explorer 3 (all of which contained Java version 1.0), my company won a bid to create the Java version of a well-known music and television Web site. The site featured applets with many clickable menus and submenus, animated promotions, music news, and lots of cool graphics. The main Web page featured a frame that was divided into three sections, two of which contained applets.



The company's staff needed to easily control how the applets behaved when users interacted with them. Staff members wanted to control applet behavior

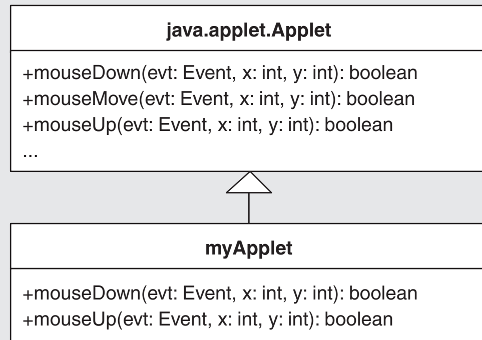




Kerievsky_book.fm Page 306 Thursday, July 8, 2004 12:12 PM

without having to call programmers every time they needed to change a piece of functionality. We were able to accommodate their needs by using the Command pattern [DP] and by creating and using a custom mouse event handler class called MouseEventHandler. Instances of MouseEventHandler could be configured (via a script) to execute Commands whenever users moved their mouse over or clicked on regions within image maps.

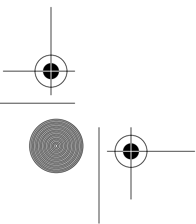
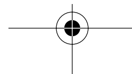
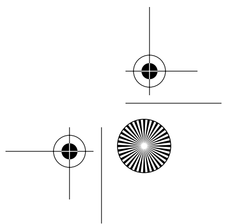
Background for the Example: Mouse Events in Java 1.0

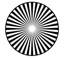
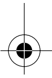
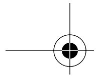
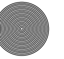
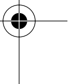
In Java 1.0, the mouse event model for Java applets relied on inheritance. If you wanted an applet to receive and handle mouse events, you would subclass java.applet.Applet and override the mouse event methods you needed, as shown in the following diagram.



Once your applet was instantiated and running on a Web page, its mouse event methods would be called in response to mouse movements or clicks by a user.

The code worked perfectly except for one problem. During start-up, our applets would load into a browser window and initialize themselves. Part of the initialization process included getting MouseEventHandler objects instantiated and configured. To inform each MouseEventHandler instance about which areas of an applet were clickable and what Commands to run when those areas were clicked, we needed to read data and pass it to each instance. While loading that data didn't take a lot of time, it did leave a window of time in which our Mouse-

Kerievsky_book.fm Page 307 Thursday, July 8, 2004 12:12 PM

EventHandler instances weren't ready to receive mouse events. If a user happened to move or click the mouse on an applet before our custom mouse event handlers were fully instantiated and configured, the browser would bark errors onto the console and the applets would become unstable.

There was an easy fix: find every place where MouseEventHandler instances could be called when they were still null (i.e., not yet instantiated) and write code to insulate them from such calls. This solved the start-up problem, yet we were unhappy with the new design. Now, numerous classes in our system featured an abundance of null checks:

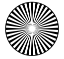
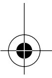
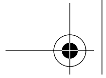
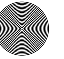
```
public class NavigationApplet extends Applet...
  public boolean mouseMove(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseMove(graphicsContext, event, x, y );
    return true;
  }

  public boolean mouseDown(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseDown(graphicsContext, event, x, y );
    return true;
  }

  public boolean mouseUp(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseUp(graphicsContext, event, x, y );
    return true;
  }

  public boolean mouseExit(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseExit(graphicsContext, event, x, y );
    return true;
  }
```

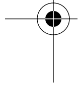

To remove the null checks, we refactored the applets so they used a NullMouse-EventHandler at start-up and then switched to using a MouseEventHandler instance when one was ready. Here are the steps we followed to make this change.

1. We applied *Extract Subclass* [F] to define NullMouseEventHandler, a subclass of our own mouse event handler:

```
public class NullMouseEventHandler extends MouseEventHandler {
  public NullMouseEventHandler(Context context) {
    super(context);
  }
}
```

   That code compiled just fine, so we moved on.





Kerievsky_book.fm Page 308 Thursday, July 8, 2004 12:12 PM



2. Next, we found a null check, like this one:

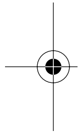
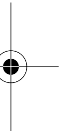
```
public class NavigationApplet extends Applet...
  public boolean mouseMove(Event event, int x, int y) {
    if (mouseEventHandler != null)  // null check
      return mouseEventHandler.mouseMove(graphicsContext, event, x, y);
    return true;
  }
```

The method invoked in the above null check is `mouseEventHandler.mouseMove(…)`. The code invoked if `mouseEventHandler` equals null is the code that the mechanics direct us to implement in an overridden `mouseMove(…)` method on `NullMouseEventHandler`. That was easily implemented:


```
public class NullMouseEventHandler...
  public boolean mouseMove(MetaGraphicsContext mgc, Event event, int x, int y) {
    return true;
  }
```

The new method compiled with no problems.

3. We repeated step 2 for all other occurrences of the null check in our code. We found the null check in numerous methods on three different classes. When we completed this step, `NullMouseEventHandler` had many new methods. Here are a few of them:

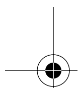
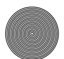
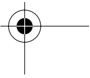
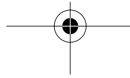
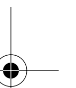
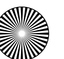
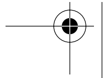



```
public class NullMouseEventHandler...
  public boolean mouseDown(MetaGraphicsContext mgc, Event event, int x, int y) {
    return true;
  }

  public boolean mouseUp(MetaGraphicsContext mgc, Event event, int x, int y) {
    return true;
  }

  public boolean mouseEnter(MetaGraphicsContext mgc, Event event, int x, int y) {
    return true;
  }

  public void doMouseClick(String imageMapName, String APID) {
  }
```

The above code compiled with no difficulties.

4. Then we initialized `mouseEventHandler`, the field referenced in the null check within the `NavigationApplet` class, to an instance of the `NullMouseEventHandler`:

```
public class NavigationApplet extends Applet...
  private MouseEventHandler mouseEventHandler = new NullMouseEventHandler(null);
```

Kerievsky_book.fm Page 309 Thursday, July 8, 2004 12:12 PM

The null that was passed to the NullMouseEventHandler's constructor forwarded to the constructor of its superclass, MouseEventHandler. Because we didn't like passing such nulls around, we altered NullMouseEventHandler's constructor to do this work:

```
public class NullMouseEventHandler extends MouseEventHandler {
  public NullMouseEventHandler(Context context) {
    super(null);
  }
}

public class NavigationApplet extends Applet...
  private MouseEventHandler mouseEventHandler = new NullMouseEventHandler();
```

5. Next came the fun part. We deleted all occurrences of the null check in such classes as NavigationApplet:

```
public class NavigationApplet extends Applet...
  public boolean mouseMove(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseMove(graphicsContext, event, x, y );
    return true;
  }

  public boolean mouseDown(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseDown(graphicsContext, event, x, y);
    return true;
  }

  public boolean mouseUp(Event event, int x, int y) {
    if (mouseEventHandler != null)
      return mouseEventHandler.mouseUp(graphicsContext, event, x, y);
    return true;
  }

  // etc.
```

After doing that, we compiled and tested whether the changes worked. In this case, we had no automated tests, so we had to run the Web site in a browser and try repeatedly to cause problems with our mouse as the NavigationApplet applet started up and began running. Everything worked well.

6. We repeated steps 4 and 5 for other classes that featured the same null check until it had been completely eliminated from all classes that originally contained it.

   Because our system used only two instances of the NullMouseEventHandler, we did not make it a Singleton [DP].