Chapter 27

Null Object

27.1 The Problem

Consider a class that contains one or several methods whose execution takes a long time, say 1–15 minutes, and whose execution is commonly made while a user is expected to wait for it to complete. One example could be a software installation procedure, or a compilation. To assure the user that the system is not hanging or broken it is common to show progress. I could code a progress indicator by inserting reporting checkpoints in my code, something like

```java
public void lengthyExecution() {
  ...
  progress.report(10);
  ..
  progress.report(50);
  ...
  progress.report(100);
  progress.end();
}
```

Here the parameter is a percentage of completion. The object progress can be implemented as a visual indicator, as we are used to in modern installation procedures.

Now, consider the situation where I want to do automated testing of the above method. It makes no sense to bring up dialogs while executing test suites. The traditional programmer's way of saying "I do not need this behavior" is to set the object reference to null. After all, null means absence of an object and therefore absence of behavior. However, the cost of this is a lot of tests in the code, as invoking a method on null leads to a null pointer exception.

```java
public void lengthyExecution() {
  ...
  if (progress != null )
    progress.report(10);
  ..
  if (progress != null )
```

```
    progress.report(50);
  ...
  if (progress != null ) {
    progress.report(100);
    progress.end();
  }
}
```

This is tedious. Even worse, it is quite easy to forget one of the checks, especially if in some code that is only rarely executed.

27.2 A Solution

Looking over the problem it is not really "absence of object" that is required in the case above. It is "absence of behavior". So, instead of representing behavioral absence by a null reference, I can represent it with absence of behavior in a special object, the **null object**, whose methods all simply do nothing. When I do not want any progress reporting behavior, it set the `progress` object reference to this null object, after which all calls of report simply do nothing. This way I do not need to guard all my method invocations by checks for null.

27.3 The Null Object Pattern

This is the NULL OBJECT pattern (design pattern box 27.1, page 58). Its intent is

> *Define a no-operation object to represent null.*

The central roles are **Service** that defines the interface for some abstraction, **ConcreteService** that implements the service, and finally the **Null Object** that contains empty implementations of all methods in the service interface.

This pattern generally aids in increasing both reliability and maintainability. As absence of behavior is represented by the null object, the object reference is always assigned to a valid object and the use of null avoided. Thereby the `if (obj != null )` checks are not needed and the risk of having forgotten it in a single or few places is eliminated. The result is less risk of null pointer exceptions. If there is a lot of such checks that are not needed it also makes the production code easier to read (compare the two code fragments in the first section), increasing analyzability and thus maintainability.

The only liability is if the design was made without much attention to the ① *program to an interface* principle and the **Client** is directly coupled to **ConcreteService**. Then there is an overhead in introducing the **Service** interface in order to allow replacing the concrete service with a null object.

The NULL OBJECT pattern was first described by Wolf (1997).

27.4 Review Questions

Describe the NULL OBJECT pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

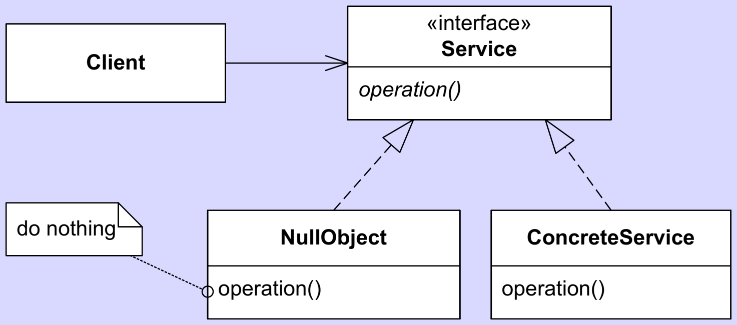
[27.1] Design Pattern: Null Object

Intent Define a no-operation object to represent null.

Problem The absence of an object, or the absence of behavior, is often represented by a reference being null but it leads to numerous checks to ensure that no method is invoked on null. It is easy to forget such checks.

Solution You create a Null Object class whose methods have no behavior, and use an instance of this class instead of using the null type. Thereby there is no need for null checking before invoking methods.

Structure:



Roles **Service** defines the interface of some abstraction while **ConcreteService** is an implementation of it. **NullObject** is an implementation whose methods do nothing.

Cost - Benefit It reduces code size and increases reliability because a lot of *testing for null is avoided.* If an interface is not already used it *requires additional refactoring to use.*