Null Object [Woolf]

Intent

Instead of using a `null` object handle, provide an alternative in form of an object whose implemented methods do nothing.

Motivation

Suppose you write a program, and you want to write some output for debugging purpose:

```
// in the body of some method:
   if (cond) {
      ...
      System.out.println("Cond is true.");
   } else {
      ...
      System.out.println("Cond is false.");
   }
   ...
```

While useful during the debugging phase, you certainly do not want the output to appear in your production release of your application. Thus, you could comment it out:

```
// in the body of some method:
   ...
   if (cond) {
      ...
      // System.out.println("Cond is true.");
   } else {
      ...
      // System.out.println("Cond is false.");
   }
   ...
}
```

However, you got a problem now with your version control system and with your software quality process: Every change of the source code requires the test cases to be re-executed!

Another approach could be the provision of an `OutputWriter` object which allows to send the debugging output wherever it wants to send to. If you don’t

want to have debugging output in your production software then provide a `null` handle:

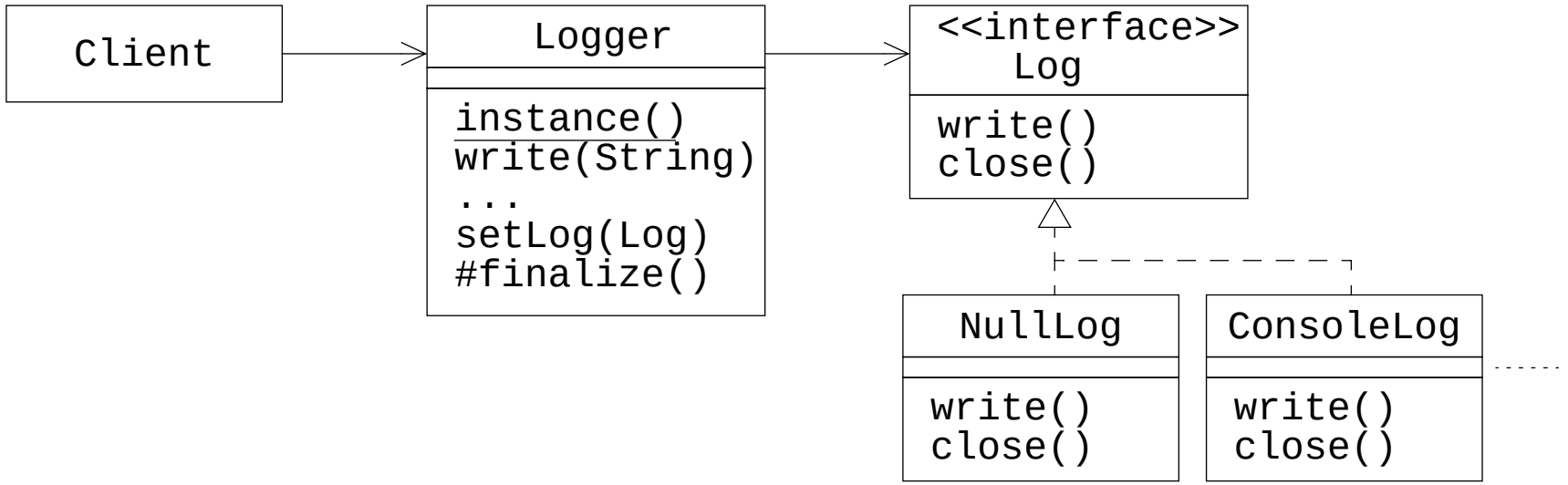
```
// Some class:
public class MyClass {
   private OutputWriter log = null;
   public MyClass() {
   }
   public MyClass(OutputWriter log) {
      this.log = log;
   }

// in the body of some method:
   ...
   if (cond) {
      ...
      if (log != null)
         log.write("Cond is true.\n");
   } else {
      ...
   }
   ...
```

```
            if (log != null)
               log.write("Cond is false.\n");
         }
      }
```

However, notice a few drawbacks of the above solution:

- You *must* ensure that the handle for the `OutputWriter` object is not `null`. You do this using `if`-statements which clutters your code.
- No uniform way is implied in the above solution how the handle for the `StringWriter` object is obtained.
- Avoiding to test the `log` handle can remain unobserved until the production code is produced.

Yet another improved solution is to have `Logger` singleton, which can be configured with actual `Log` objects:



A client acquires the pre-configured `Logger` singleton object, optionally re-configures it with `setLog`, and then uses it by applying the `write` methods. Several overloaded `write` methods could be provided by `Logger`. For example, a method `write(int level, String log)` could write a `log` message only if it has a certain importance.

The `Logger` object writes the `log` message by using one of the concrete `Log` objects. Optionally, it can decorate the `log` string by some standard information pattern such as the date and time. Method `finalize` closes the concrete `Log` object upon destruction of the `Logger` object by the garbage collector.

The concrete `Log` objects finally do the actual writing. `ConsoleLog`, for example, writes the messages to the console using, for example, `System.out` or `System.err`.

The `NullLog`, on the other hand, does *nothing*.Its method bodies are empty. See the details in section “Sample Code”.

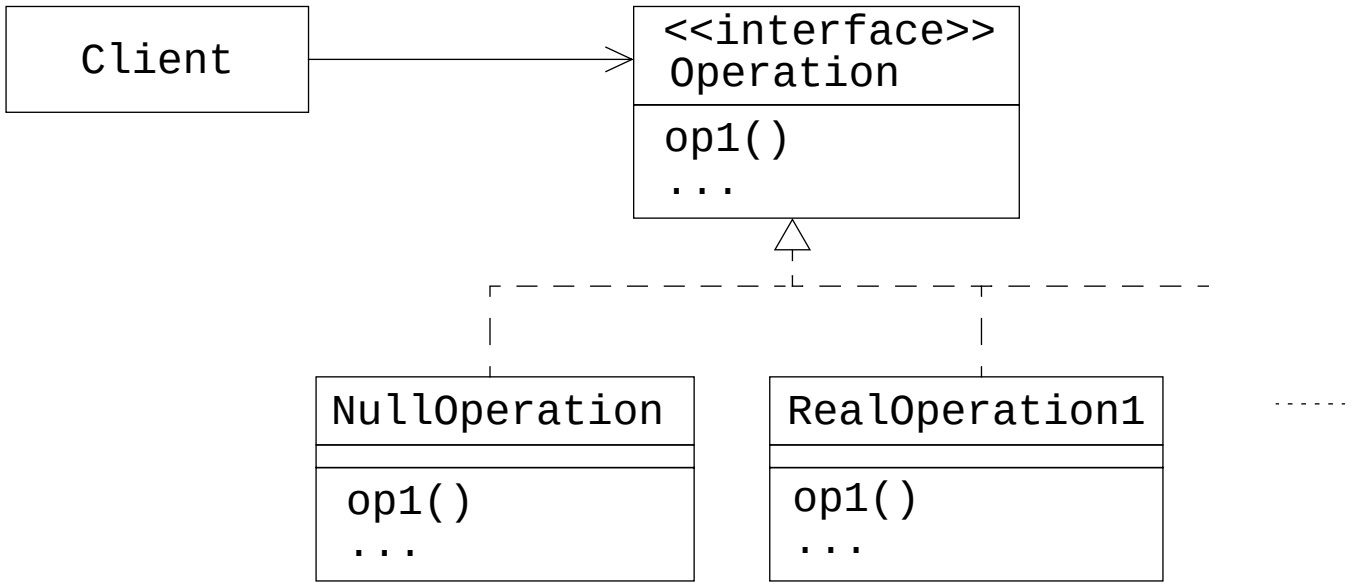
Applicability

Use this pattern when you want uniformly apply an operation on a kind of object, but the object’s behavior depends on its configuration with other objects.

Null objects can sometimes be used to reduce the number of branches in methods: Instead of testing for the `null` value in the body of the method, the call is delegated to the null object, which in turn does “nothing”.[1] As a side effect, the null object pattern reduces the number of test cases (using code coverage).

1. In some situations, the Null object does some kind of “compensation”.

Structure


Client
<<interface>>
Operation
op1()
...
NullOperation
op1()
...
RealOperation1
op1()
...

Participants

- `Operation (Log)`:
  - declares the interface for the concrete `Operation` objects
- `NullOperation (NullLog)`:
  - implements the “null” operations of the operations enforced by `Operation`
- `RealOperation (ConsoleLog, ...)`:
  - implements the “real” operations

Collaborations

- A client uses a concrete `Operation` object. The concrete `Operation` object can be changed during the execution of the application.
- The client acts sometimes as a singleton for an application.
- The client delegates the concrete methods to the concrete `Operation` object.

Consequences

- A uniform way is introduced into your application.
- The concrete `Operation` can be defined at build time (conditional compilation with C++), in Java at load time (by loading the appropriate class, e.g., via a properties file), or at run-time.

Implementation

Since in most cases, the `NullOperation` class contains no instance-specific information, it can then be implemented as singleton. You could also use an abstract class instead of an interface for class `Log`.

Sample Code

We complete in part the example introduced in the “Motivation” section. The `Log` interface might look like:

```
public interface Log {
   public void write(String msg);
   public void close();
}
```

Class `ConsoleLog` implements interface `Log`:

```
public class ConsoleLog implements Log {
   public void write(String msg) {
      System.out.println(msg);
   }

   public void close() {
      // do nothing special here...
   }
}
```

Note that method `close` is void in this class. Next, we present the `NullLog` class:

```
public class NullLog implements Log {
   public void write(String msg) {/* do nothing */}
   public void close() {/* do nothing */}
}
```

An excerpt of an `FileLog` class might look like:

```
public class FileLog implements Log {
   private PrintWriter out;
   public FileLog(String name) {
      out = new PrintWriter(new BufferedWriter
            (new FileWriter
               (new File(name))));
    }
   public void write(String msg) {/* left as an exercise */}
   public void close() {/* left as an exercise */}
   public void finalize() {
      if (out != null)
         out.close();
}  }
```

A client may act as a singleton and provide a log service to the application:

```
public class Logger {
   private static Logger instance = null;
   private Log log = null;
   public static Logger instance() {
      if (instance == null) {
         instance = new Logger();
         instance.log = new ConsoleLog();
               // default configuration
      }
      return instance;
   }
   public void setLog(Log log) {
      // omitted for brevity...
   }
   public void write(String msg) {
      try {
         log.write(msg);
      } catch (Exception ex) {}
   }
}
```

Related Patterns

- Singleton pattern: the `NullOperation` class can often be written as a singleton.
- Decorator pattern: The null object pattern can be combined with the decoration pattern to decorate the messages.
- Composite pattern: Use the composite pattern to perform more than one operation at a time, for example, writing logs simultaneously to one or more log receivers.