NULL OBJECT DESIGN PATTERN

Shani Guttman
Doron Tuchman

INTRODUCTION

The intent of a Null Object is to **encapsulate** the **absence** of an object by providing a substitutable alternative that offers suitable default **do nothing** behavior. In short, a design where *"nothing will come of nothing."*

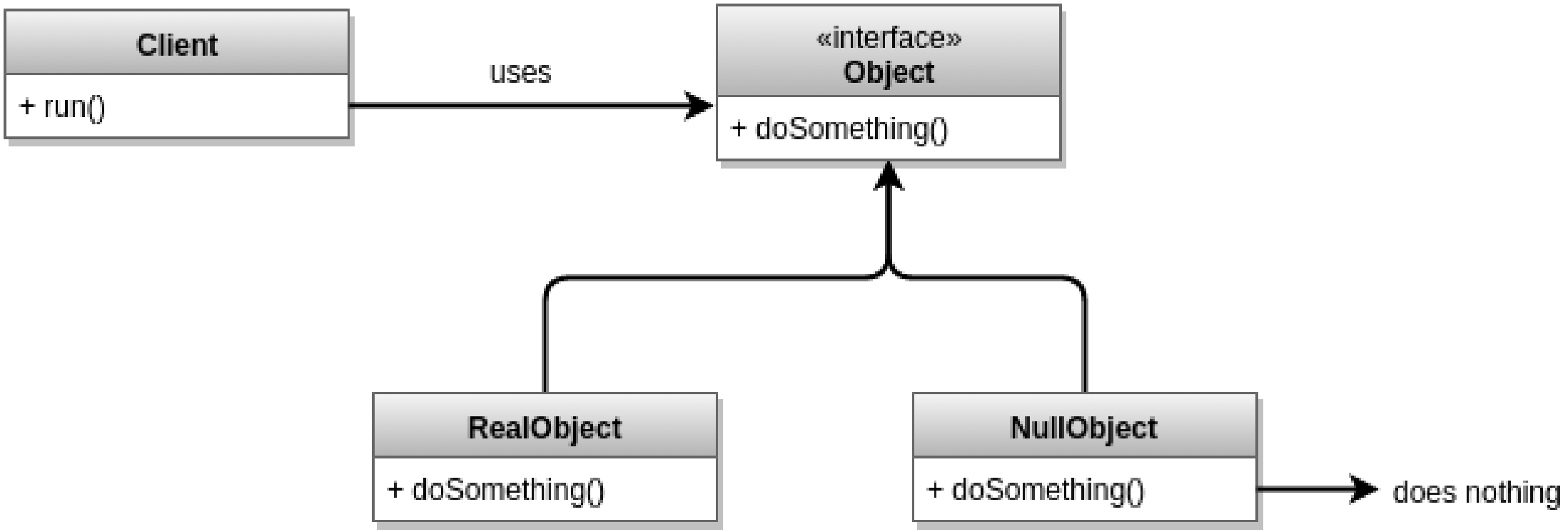
Null Object pattern is **Behavioral** design pattern.

THE PATTERN EXPLAINED

The null object design pattern describes the uses of null objects and their behavior in the system.

- Null object patterns deal with null objects.
- Instead of checking for the null object, we define null behavior or call do-nothing behavior.
- These null objects can also be used to provide default behavior in case data is unavailable.
- The advantage of this approach over a working default implementation is that a null object is very predictable and has no side effects — it does *nothing*.

UML DIAGRAM


Client
+ run()
uses
«interface»
Object
+ doSomething()
RealObject
+ doSomething()
NullObject
+ doSomething()
does nothing

ADVANTAGES

- Client code is simplified and cleaner, because it avoids having to write testing code which handles the null collaborator specially.
- There is no need to be cautious when accessing an object. Client code will always get a reference to a valid object, even if that object does nothing.
- This implementation can speed up the code, by eliminating unnecessary if checks.

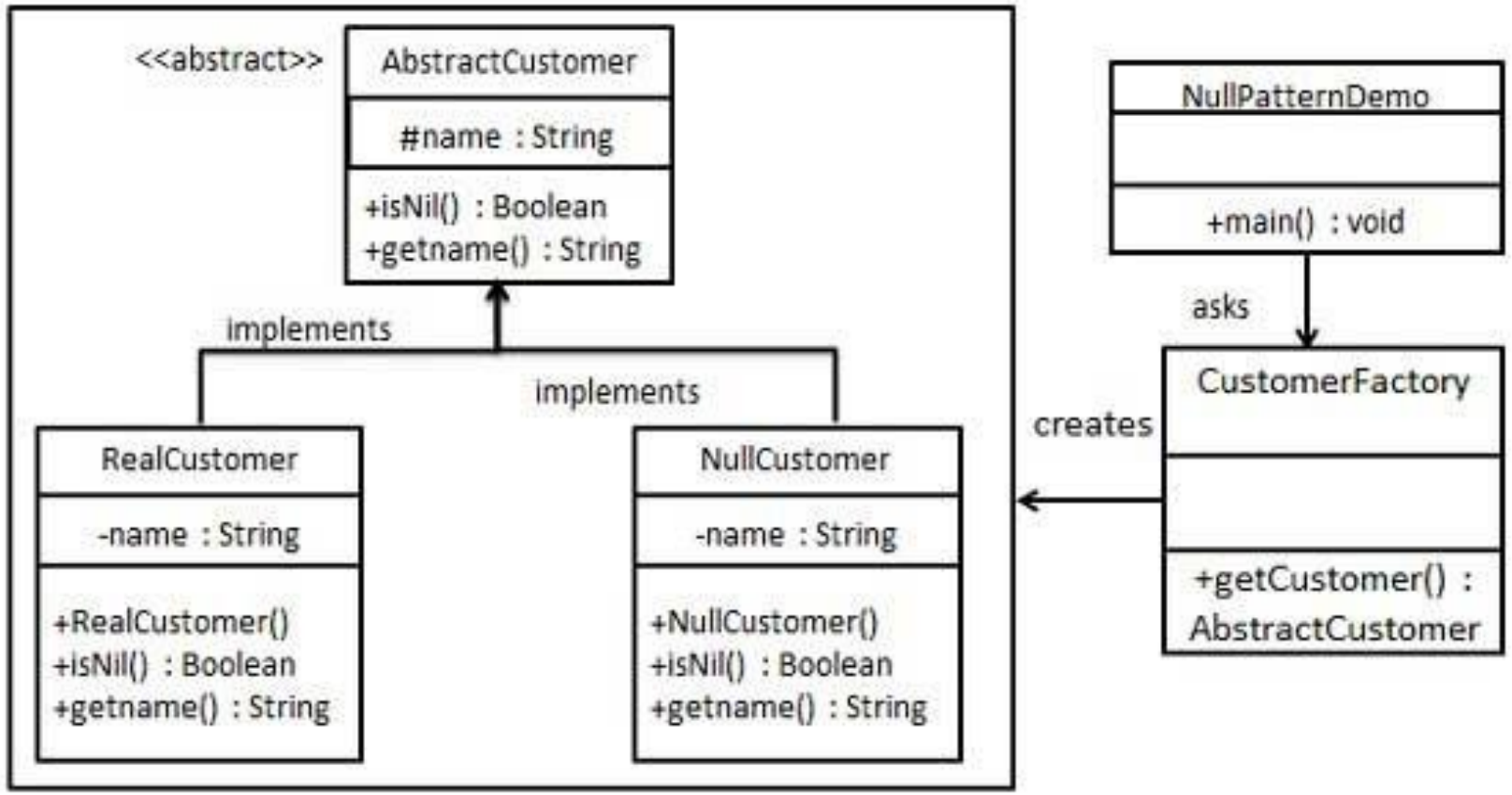
DISADVANTAGES

- Can necessitate creating a new NullObject class for every new AbstractObject class.
- Costs a lot in memory terms.
- The “do-nothing” behavior may not be well defined. If there are multiple clients, they must all “agree” on what a default behavior is, or we may need to implement different null objects.
- Sometimes a null object must be replaced by a real object, which can’t be done using only this design pattern.

EXAMPLES OF USE

- **We should use the Null Object Pattern when a *Client* would otherwise check for *null* just to skip execution or perform a default action.** In such cases, we may encapsulate the neutral logic within a null object and return that to the client instead of the *null* value. This way client's code no longer needs to be aware if a given instance is *null* or not.
- Such an approach follows general object-oriented principles, like Tell-Don't-Ask.
- Tell-Don’t-Ask - principle that encourages to move behavior into an object to go with the data.

EXAMPLE


<<abstract>>
AbstractCustomer
#name : String
+isNil() : Boolean
+getname() : String
implements
implements
RealCustomer
-name : String
+RealCustomer()
+isNil() : Boolean
+getname() : String
NullCustomer
-name : String
+NullCustomer()
+isNil() : Boolean
+getname() : String
NullPatternDemo
+main() : void
asks
creates
CustomerFactory
+getCustomer() : AbstractCustomer

ABSTRACT CLASS

```
public abstract class AbstractCustomer {
 protected String name;
 public abstract boolean isNil();
 public abstract String getName();
}
```

CONCRETE CLASSES EXTENDING THE ABSTRACT CLASS

```
public class RealCustomer extends AbstractCustomer {
 public RealCustomer(String name) {
 this.name = name;
  }
 @Override public String getName() {
 return name;
  }
 @Override public boolean isNil() {
 return false;
  }
}
```

NULL OBJECT – NULL CUSTOMER

```
public class NullCustomer extends
AbstractCustomer {
 @Override public String getName() {
 return "Not Available in Customer Database";
  }
 @Override public boolean isNil() {
 return true;
  }
}
```

CUSTOMER FACTORY

```
public class CustomerFactory {

   public static final String[] names = {"Rob", "Joe",
"Julie"};

   public static AbstractCustomer getCustomer(String name){

      for (int i = 0; i < names.length; i++) {
         if (names[i].equalsIgnoreCase(name)){
            return new RealCustomer(name);
         }
      }
      return new NullCustomer();
   }
}
```

DEMONSTRATION

```
public class NullPatternDemo {
   public static void main(String[] args) {

      AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");
      AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");
      AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");
      AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");

      System.out.println("Customers");
      System.out.println(customer1.getName());
      System.out.println(customer2.getName());
      System.out.println(customer3.getName());
      System.out.println(customer4.getName());
   }
}
```

OUTPUT

```
Customers
Rob
Not Available in Customer Database
Julie
Not Available in Customer Database
```

IMPROVEMENTS

- **Problem:** a lot of null objects can waste memory.

  Solution: We can implement the null object as a singleton.

- **Problem:** Sometimes a null object must be replaced by a real object.

  Solution: This can be implemented by using the state design pattern.

IMPROVEMENTS - SINGLETON

- In order to improve performance, it makes sense to create just a **single immutable** instance of the NULL class. Because the NULL class just has default values and default behavior so caching it makes more sense.

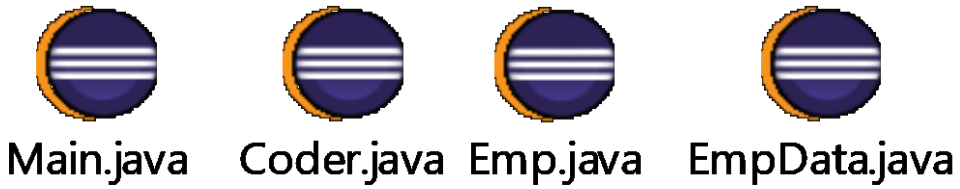
IMPROVEMENTS – STATE DESIGN PATTERN

- In order to replace null object in a real object, it makes sense to use the **state design pattern**. The **state pattern** allows an object to alter its behavior when its internal state changes. In this way we can alter null object to be a real object.

CLASS EXERCISE 1

- Your task is to implement null object patten:

- 1. Create a class and name it "NoClient" this class will Extend the class "Emp".
- 2. Override the functions of "Emp" class.



CLASS EXERCISE 2

Your task is to implement null object patten:

- 1. Create a class and name it "NullShape" this class will implement the class "Shape".
- 2. Override the functions of "Shape" class.