Null Object Pattern to avoid Null checks?

Lately I have come across Null Object design pattern and my colleagues say it can be used to do away with the null pointer checks that are encountered throughout the code.

for e.g suppose a DAO class returns information on Customer (in a value object called CustomerVO). My main class is supposed to extract the firstName and emailId and send email to customer.

```
...
CustomerVO custVO = CustomerDAO.getCustomer(customerID);
if(custVO != null) { // imp, otherwise we may get null ptr exception in next line
     sendEmail(custVO.getFirstName(), custVO.getEmailID());
}
...
```

This is very simple example, but such null checks can quickly spread throughout your code based on the complexity of the value objects.

I have two issues with null check, - they tend tmake the code ugly and hard to read - lesser experienced developers put unnecessary null checks when in fact they are supposed to throw exceptions. for e.g. in above code, it would be better to throw exception from getCustomer() itself because if its not able to find customer info for given CustID, it indicates the CustID was invalid.

okay, coming back to null object pattern, can we use a 'null' CustomerVO object to hide the null check?

```
CustomerVO {
   String firstName = "";
   String emailID = "";
}
```

Don't it would make sense. what do you think?

And what are the things you follow to minimize null checks in your app.

design-patterns null-object-pattern

edited Oct 17 '10 at 6:12 asked Oct 17 '10 at 5:53

 ergosys
33.2k 4 30 61

 Rohit
322 1 7 23

5 Answers

In this case a null object may be inappropriate since the default may infact hide what is in actuality an exception. If you find yourself having to check to see if your have safe null to perform some other activity the null pattern isn't buying you anything.

As you stated many new developers spend time trying to protect their code from exception situations that are worse then a halting program.

answered Oct 17 '10 at 6:07

 rerun
20.4k 4 34 70

While the null object pattern has it's uses, you're still going to need to make a check here, otherwise you're going to try to `sendEmail()` to an email address that's the empty string (or you push the check into `sendEmail()`, which could just as easily check for `null`).

Where the null object pattern would be really helpful here is if the `CustomerVO` class implemented the `sendEmail()` method. then you could simply chain the calls together, since the contract of `getCustomer()` would ensure that a `null` reference wouldn't be returned:

```
CustomerDAO.getCustomer(customerID).sendEmail();
```

In that case the `sendEmail()` method would check that its been asked to act on the special 'null object' and simply do nothing (or whatever's appropriate).

answered Oct 17 '10 at 6:09

Michael Burr
274k 37 415 649

1 +1, though I wouldn't want my customers to have sendEmail methods. That's just giving a customer class too many responsibilities. It's like giving every class a "print" method and now having to implement print logic everywhere or making every class dependent on some print manager, while it could be restricted to a print manager and the forms/classes that actually initiate printing giving the print manager the classes to print. Where the null pattern is really nice, is in unit testing for example to provide a log class that doesn't do anything so you don't have to ammend the code being tested. – Marjan Venema Oct 17 '10 at 7:50

Should your `getCustomer` method throw an exception if the customer is not found, rather than returning null?

The answer, of course, is that it depends: should it *almost never* be the case that a customer ID does not exist? In other words, is it an exceptional circumstance? If so, an exception is appropriate.

However, in a data-access layer it is often quite normal for something to not exist. In that case, it is better not to throw since it isn't an unexpected, exceptional circumstance.

The 'return a non-null object that has empty fields' is probably not any better. How do you know if the returned object is 'valid' or not without now adding some checking code that is probably worse than the null check?

So, if it might be a normal state that something being fetched does not exist, the null check pattern is probably best. If it is something that is unexpected then having the data access method throw a NotFound exception is probably better.

answered Oct 17 '10 at 6:19

mtreit
496 2 6

See my answer, if something can not-exist, have a method that checks whether it exists. That way you only need to throw exceptions where it's exceptional. – nicodemus13 May 15 '12 at 15:55

I have an issue with this type of code, which is a common pattern. You don't need the null checks at all if

you decompose what you're actually doing. The problem here, in my view, is that you're violating SRP.

The method `CustomerVO custVO = CustomerDAO.getCustomer(customerID);` does two things.

Firstly it returns a customer, if the customer exists and secondly returns null if there is no such customer. These are two distinct operations and should be coded as such.

A better approach would be:

```
bool customerExists = CustomerDAO.exists(customerID);

if (customerExists)
{
  CustomerVO custVO = CustomerDAO.getCustomer(customerID);
  sendEmail(custVO.getFirstName(), custVO.getEmailID());
}
else
{
   // Do whatever is appropriate if there is no such customer.
}

}
```

So, split the method into two, one which checks whether the requested object exists and the second which actually retrieves it. No need for any exceptions and the design and semantic is very clear (which, with a does-not-exist-returns-null pattern it isn't, in my view). Moreover, in this approach, the `CustomerDAO.getCustomer(customerID)` method throws an `ArgumentException`, if the requested customer does not exist. After all, you've asked for a `Customer` and there isn't one.

Futhermore, in my view, no method should return `null`. `Null` is explicitly, 'I don't know what the correct answer is, I have no value to return'. `Null` isn't a meaning, it's a *lack* of meaning. Ask yourself, 'why am I returning something that I know shouldn't really happen? Your method `GetCustomer` should return a `Customer` obviously. If you return `null`, you're simply pushing the responsibility of what to do back up the call-chain and breaking the contract. If there's meaningful default, use that, but throwing an exception here may make you think a bit harder about what the correct design is.

answered May 15 '12 at 15:53

nicodemus13

1,507 1 11 27

The name is NULL object design pattern and not NULL check design pattern. Null object signifies absence of a object. This should be used when you have objects working in COLLABORATION. In your case you are checking existence of a object for which NULL check should be fine.

NULL design pattern is not meant to replace NULL exception handling. It's one of the side benefits of NULL design pattern but the intention is to provide a default behavior.

NULL checks should not be replaced with NULL design pattern objects as it can lead to silent defects in application.

Do see the below article which goes in details of DO's and Donts of NULL design pattern.

http://www.codeproject.com/Articles/1042674/NULL-Object-Design-Pattern

answered Oct 30 '15 at 7:50

Shivprasad Koirala
14.2k 5 54 51