Is there a name for this anti-pattern/code smell?

Asked 16 years, 3 months ago Modified 5 years, 7 months ago

Viewed 4k times

19

Let me start by saying that I do not advocate this approach, but I saw it recently and I was wondering if there was a name for it I could use to point the guilty party to. So here goes.

Now you have a method, and you want to return a value. You *also* want to return an error code. Of course, exceptions are a much better choice, but for whatever reason you want an error code instead. Remember, I'm playing devil's advocate here. So you create a generic class, like this:

```
class FunctionResult<T>
{
    public T payload;
    public int result;
}
```

And then declare your functions like this:

```
FunctionResult<string> MyFunction()
{
    FunctionResult<string> result;
    //...
```

```
    return result;
}
```

One variation on this pattern is to use an enum for the error code instead of a string. Now, back to my question: is there a name for this, and if so what is it?

design-patterns anti-patterns

Share

Improve this question

Follow

edited Apr 30, 2019 at 4:08

Bhargav Rao

52k ● 29 ● 126 ● 141

asked Sep 16, 2008 at 14:10

Joel Coehoorn

415k ● 114 ● 577 ● 813

1 The "Somebody who isn't an idiot is going to break my fingers with a hammer when they see this" pattern. But that pretty much applies to all antipatterns. – user1228 Sep 16, 2008 at 14:15

1 If this is an anti-pattern, then what is the recommendation for languages which don't directly support exceptions (I'm thinking of VB/VBA). – Onorio Catenacci Sep 16, 2008 at 14:41

1 If they don't support exceptions, they probably don't support templates/generics either. – Joel Coehoorn Sep 16, 2008 at 15:17

Joel--yep, hadn't thought of that. :-) I guess I tend to think of patterns (and anti-patterns) as language-agnostic but you're

right; given the context I can see how that would look like an anti-pattern. – Onorio Catenacci Sep 16, 2008 at 17:03

13 Answers

Sorted by: Highest score (default)

17

I'd agree that this isn't specifically an antipattern. It might be a smell depending upon the usage. There are reasons why one would actually not want to use exceptions (e.g. the errors being returned are not 'exceptional', for starters).

There are instances where you want to have a service return a common model for its results, including both errors and good values. This might be wrapped by a low level service interaction that translates the result into an exception or other error structure, but at the level of the service, it lets the service return a result and a status code without having to define some exception structure that might have to be translated across a remote boundary.

This code may not necessarily be an error either: consider an HTTP response, which consists of a lot of different data, including a status code, along with the body of the response.

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:29

Dan Fleet

274 ● 1 ● 5

I've just discovered code smells, and I'll grant you smell vs anti-pattern. Not sure how to update the question to reflect that, though. – Joel Coehoorn Sep 26, 2008 at 16:42

13

It is called "Replace Error Code with Exception"

Share Improve this answer

Follow

edited Apr 30, 2019 at 4:08

Bhargav Rao

52k ● 29 ● 126 ● 141

answered Sep 16, 2008 at 14:22

rmaruszewski

2,417 ● 1 ● 21 ● 21

I was thinking more about combining the error code in a template than error codes in general. – Joel Coehoorn Oct 2, 2008 at 17:09

10

Well, it's *not* an antipattern. The C++ standard library makes use of this feature and .NET even offers a special `FunctionResult` class in the .NET framework. It's called `Nullable`. Yes, this isn't restricted to function results but it can be used for such cases and is actually very useful here. If .NET 1.0 had already had the `Nullable` class, it would certainly have been used for the `NumberType.TryParse` methods, instead of the `out` parameter.

Share Improve this answer

answered Sep 16, 2008 at 14:23

Follow

Konrad Rudolph
545k ● 139 ● 956 ● 1.2k

6

I usually pass the payload as (not const) reference and the error code as a return value.

I'm a game developer, we banish exceptions

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:28

ugasoft
3,802 ● 7 ● 29 ● 23

5

Konrad is right, C# uses dual return values all the time. But I kind of like the TryParse, Dictionary.TryGetValue, etc. methods in C#.

```
int value;
if (int.TryParse("123", out value)) {
    // use value
}
```

instead of

```
int? value = int.TryParse("123");
if (value != null) {
    // use value
}
```

...mostly because the Nullable pattern does not scale to non-Value return types (i.e., class instances). This

wouldn't work with Dictionary.TryGetValue(). And TryGetValue is both nicer than a KeyNotFoundException (no "first chance exceptions" constantly in the debugger, arguably more efficient), nicer than Java's practice of get() returning null (what if null values are expected), and more efficient than having to call ContainsKey() first.

But this *is* still a little bit screwy -- since this looks like C#, then it should be using an out parameter. All efficiency gains are probably lost by instantiating the class.

(Could be Java except for the "string" type being in lowercase. In Java of course you have to use a class to emulate dual return values.)

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:36

Peter Davis

51 • 1

Well, non-value types already have `null` values. `Nullable` was basically an effort to replicate this for value types. Of course, as soon as `null` is an acceptable value, this method would fail. – Konrad Rudolph Sep 16, 2008 at 14:42

4

I am not sure this is an anti-pattern. I have commonly seen this used instead of exceptions for performance reasons, or perhaps to make the fact that the method can fail more explicit. To me, it seems to be a personal preference rather than an anti-pattern.

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:19

Gilligan

1,515 ● 2 ● 15 ● 16

3

I agree with those that say this is not an anti-pattern. Its a perfectly valid pattern in certain contexts. Exceptions are for *exceptional* situations, return values (like in your example) should be used in expected situations. Some domains expect valid and invalid results from classes, and neither of those should be modeled as exceptions.

For example, given X amount of gas, can a car get from A to B, and if so how much gas is left over? That kind of question is ideal for the data structure you provided. Not being able to make the trip from A to B is expected, hence an exception should not be used.

Share Improve this answer

Follow

answered Sep 16, 2008 at 15:39

ARKBAN

3,477 ● 4 ● 26 ● 23

2

This approach is actually much better than some others that I have seen. Some functions in C, for example, when they encounter an error they return and seem to succeed. The only way to tell that they failed is to call a function that will get the latest error.

I spent hours trying to debug semaphore code on my MacBook before I finally found out that sem_init doesn't work on OSX! It compiled without error and ran without

causing any errors - yet the semaphore didn't work and I couldn't figure out why. I pity the people that port an application that uses POSIX semaphores to OSX and must deal with resource contention issues that have already been debugged.

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:26

Kyle Cronin

79k ● 45 ● 151 ● 167

1

How about the "Can't decide whether this is an error or not" pattern. Seems like if you really had an exception but wanted to return a partial result, you'd wrap the result in the exception.

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:13

noah

21.5k ● 17 ● 67 ● 88

1

If you don't want to use exceptions, the cleanest way to do it is to have the function return the error/success code and take either a reference or pointer argument that gets filled in with the result.

I would not call it an anti-pattern. It's a very well proven workable method that's often preferable to using exceptions.

Share Improve this answer

answered Sep 16, 2008 at 14:34

Follow

17 of 26
27.4k • 13 • 68 • 85

1

If you expect your method to fail occasionally, but don't consider that exceptional, I prefer this pattern as used in the .NET Framework:

```
bool TryMyFunction(out FunctionResult result){

     //...
     result = new FunctionResult();
}
```

Share Improve this answer

Follow

answered Sep 16, 2008 at 14:36

Paul van Brenk
7,540 • 2 • 35 • 39

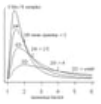
1

Debates about smells and anti-patterns remind me the "Survivor" TV shows, where you have various programming constructs trying to vote each other off the island. I'd prefer to see "construct X has such-and-so pros and cons", rather than a continually evolving list of edicts of what should and should not be done.

Share Improve this answer

Follow

answered Nov 19, 2008 at 0:31



Mike Dunlavey
40.6k • 15 • 94 • 138

In defense of the anti-pattern designation, this code lends itself to being used in a few ways:

0

1. Object x = MyFunction().payload; *(ignoring the return result - very bad)*
2. int code = MyFunction().result; *(throwing away the payload - may be okay if that's the intended use.)*
3. FunctionResult x = MyFunction(); //... *(a bunch of extra FunctionResult objects and extra code to check them all over the place)*

If you need to use return codes, that's fine. But then use return codes. Don't try to pack an extra payload in with it. That's what *ref* and *out* parameters (C#) are for. Nullable types might be an exception, but only because there's extra sugar baked in the language to support it.

If you still disagree with this assessment, DOWNVOTE this answer (not the whole question). If you do think it's an anti-pattern, then UPVOTE it. We'll use this answer to see what the community thinks.

Share Improve this answer Follow

edited Sep 16, 2008 at 17:26

answered Sep 16, 2008 at 17:03

Joel Coehoorn

415k ● 114 ● 577 ● 813

I'd prefer something like TryParse, one out param for the payload, with the function returning the result. – Meff Sep 24, 2008 at 21:43