Conditionals, Big Balls of Mud, and How to Avoid Them

Reduce and simplify conditionals and maintenance

## Loops and Sets

If you are looping, you are operating on a set of data. But often we don’t think in terms of sets when we begin to write loops. We are often focused on what we want to do with each item of the iteration, but not how we are going to be interacting with the sets and subsets of the data. Example:

if(f.FeatureType == 1)

{

//do some processing

}

else

{

//do other processing

}

This is a pretty common scenario. But what happens when we want to add another feature type to our product? That is reflected in our loop for FeatureType == 3:

if(f.FeatureType == 1)

{

//do some processing

}

else if (f.FeatureType == 3)

{

//do the 3 processing

}

else

{

//do other processing

}

This is pretty common as new features are requested, bug fixes are made, special cases are requested, or new developers come along. How about if we need to add a status indicator into the dataset? Products and Orders go in and out of service, a customization is needed. Maybe now we can’t process the 1’s if they are a status 2, but we can process the 3’s if they are status 1 or 3:

if(f.FeatureType == 1 && f.Status != 2)

{

//do some processing

}

else if(f.FeatureType == 3 && (f.Status == 1 || f.Status == 3))

{

//do the 3 processing

}

else

{

//do default processing

}

Whoa! Is that even right? Now my FeatureType 1’s that are not status 2 drop into the default case. What if FeatureType 3 *is* status 2 or another status comes along? Is it supposed to be processed as the default case or not processed at all? It’s pretty easy to forget what you wrote 4 months ago, let alone intended, and if it’s a new developer he/she probably never knew what was intended, they were just told to add this new status. It can become a maintenance nightmare; a single change can break everything downstream. Now its become a Big Ball of Mud.

Now I know this isn’t anything new. My intent is not to imply you are not aware of this, but to show you a sample of something I have seen before and how I approached it. If nothing else, it’s an interesting exercise in refactoring. We’re not looking to get rid of all conditionals, it isn’t very practical to spend your time there. We just want to mitigate some of the risks that go along with the conditionals we write all the time, and if your sequential conditionals are getting close to 13, well, it may be time to take a look at it.

## Type Tests

We test for types, everywhere we go. In the case of the examples above, f represents a Feature, but f.FeatureType is a data point that represents a more specific type. If you see “type” in your data, or “mode” or similar words that define data points, it’s a candidate for inheritance. Status could be a subtype, for further derived classes, or maybe, as in the cases above it indicates a “has a” relationship and is therefore better suited for composition.

So loops, type tests, especially as they get long and involved, are candidates for a factory pattern, at least. Keep the tests for types, subtypes and needed conditionals like a “has a” status field as close to the source as possible. Each Feature handler generated by the factory encapsulates its own information, and hides it from everyone else. Each has a single responsibility which becomes easier to maintain than an enormous method with lots of conditionals. So in my sample this code, my Mud Ball:

if(f.FeatureType == 1 && f.Status != 2)

{

//do some processing

Console.WriteLine( "Processing FeatureType {0}", f.FeatureType );

}

else if (f.FeatureType == 3 && (f.Status == 1 || f.Status == 3))

{

//do the 3 processing

Console.WriteLine( "Processing FeatureType {0}", f.FeatureType );

}

else

{

//do default processing

Console.WriteLine( "Processing Default FeatureType {0}", f.FeatureType );

}

becomes this:

IFeature myFeature = MyFeatureFactory.GetFeature( f );

myFeature.DoProcess();

## Final Comments

Let’s not beat it to death, but we all know refactoring is an iterative operation. You see new patterns, further improvements you can make. You may notice I have not tested for nulls. Yet we test for nulls everywhere, all the time. If we don’t, we run the “shame” of having a null reference pointer in production. But why not test for nulls in one place? A null feature reference can be viewed as a type test, a null FeatureType. Add the Null Pattern to the Factory, return the null feature handler and never test for nulls again, at least for the feature data point, except inside the feature handler. Also, we ended up with a default case, so instead of an interface it could be an abstract class, and let it provide the default handler, in this example Feature2 became the default.