---

layout: post

title: "The real problem with small classes"

description: "small classes and why they are avoided"

category: Programming

tags: [Programming, Craft]

For a while now I have been an advocate of SOLID principles and trying to keep classes to a level where they have a single responsibility. In applying these principles I have found my classes have become small, very small. So it is no surprise that whenever I pair with a new partner (be it at work or at a Code Retreat) we end up having a discussion on what a class is and how 'small' it should be.

In going through this discussion there tends to agreement at a conceptual level. For instance, in discussing class design my partner will generally agree with me that small classes often increase the cohesive nature of a class while decreasing the level of coupling to the rest of the system. They usually agree with me on the value of applying SOLID principles and how beneficial it is to have just a single responsibility for a class.

But when we move from the theoretical to the practical and write some code suddenly there is tension again. At the point of coding it seems that even after agreement in the theoretical discussion I still want to make the classes smaller and they still want to make them bigger.

I have been questioning why this happens and I believe there are a number of reasons for making classes large.

One major reason why classes are made so large is how we model them. Object orientation at a college or school level is often taught with too tight a parallel to the real world. Students are told to break the system into components and break those components into smaller components. Often you get oversimplified examples used to teach what an object is. For instance, a sausage dog is a subset of a dog, which in turn is a subset of an animal so make a poodle class that inherits from dog class which must then inherit from an animal class and there you go - OOP. This is not an appropriate way to think of an object in OOP.

Objects in OOP are not the same things as objects in the real world. While it can be beneficial to introduce objects to beginners in this way so that they can grasp inheritance – this is a very small part of OOP. It is my experience that inheritance is oversold in OOP and unless people move past this mind set and towards a "responsibility" oriented view of OOP they will battle to conceptualize how to reduce an object to a single responsibility. Unless you make this shift your classes will remain large. For instance, with my dog example how do you identify the single responsibility of a poodle dog?

For me another reason why classes are so large is because of our fear of violating DRY. Violating DRY is almost as bad as using a goto statement in our code. To avoid violating DRY we try and use classes as a grouping mechanism to bunch common functions together.

Sometimes we make this grouping blatant and call them 'helper' classes. Often we will try and mask these ‘module’ classes – for instance a repository class often plays the role of a ‘module’ class. I find ‘module’ classes come from a procedural mind-set. If classes were meant to be used as modules, what is the difference between the concept of a class in OOP and the concept of modules in procedural programming?

I am not saying that being able to locate code for reuse is not important, but if you are using classes as a mechanism for this you are using the practice/tool to fulfil an important principle.

I work predominantly in the .Net space, so my experience is heavily affected by the toolset available in that space - but my experience has been that many developers are not equipped to easily locate classes.

If you are going to have small classes you are going to need to have a way to efficiently find them. For instance, in Visual Studio the IDE typically 'encourages' you to navigate your solution via solution explorer. If you were to make classes have a single responsibility you would see an explosion of classes in your solution. Navigating a solution that has hundreds if not thousands of classes can be both tedious and daunting if you are not effectively equipped to do so.

So, it should be no surprise that there are productivity enhancements to most IDE's that help you get past this barrier. Remember the problem isn't small classes - it is how you work with them. For instance, with Visual Studio, there are a number of tools that help you locate classes without having to use solution explorer. I am a big fan of ReSharper's symbol search feature which is a productity tool that allows you to easily locate classes without having to know exactly where in the solution they are located. It changes the way you view the structure of your solution. By leveraging such a tool it suddenly becomes possible to navigate between small classes very efficiently.

Once you have a toolset that allows you to work with small classes suddenly the argument for using 'larger' classes as a grouping mechanism falls away. When you are empowered to make classes smaller one of two things happen - in becoming smaller they either become more generalized and reuse increases, or they become more specialized to the specific business problem you are solving.

So how small should a class be? It depends... For me I am not too concerned about the internals of a class, I find the interface of the class is important. How many things do I make public on a classes interface? Typically I find that I cannot make things much smaller than a single well-formed public method per interface of a class. Sometimes I might have two or three public methods... but more than that and I feel it is becoming bloated and doing too much.

Is a single public method per class realistic? Yes, it often is. I have managed to write some very effective systems using this approach for the majority of my classes.

Today I would like to challenge this misconception. As I have reduced the scope of classes I see one of two things happen. Either the classes becomes more of an abstraction or it becomes more specific towards that particular problem. These are both good things. Classes that become more abstract become more generalized which often means I have opportunities of reusing them elsewhere. Classes that become more specific may have less reuse (but it is now isolated to a smaller area), so in essence they also increase the reuse of the rest of the code.

believe in classes having high cohesion and low coupling

but into the concept of single responsibility

So then why the resistance to smaller classes? Well going back to the original reason supplied on why not to make a class any smaller my pairing partners are correct, you see an explosion of classes when you make them small and it maybe be harder to identify code blocks. But the problem isn't with the classes being small. It is in how the developer is locating those classes. In Visual Studio I find they typically navigate the solution via solution explorer. This promotes the mindset of remembering where you put things.

If you changed this and instead had a gopher that you told what type of classes you were looking for and the gopher then went through all the available options and returned a few options you would what I believe the root cause of this problem is. And its the hardest problem in computer science. Naming things.

Once you focus on this problem you will use classes as a grouping mechanism to find concepts

change the way we navigate solutions so that it isn't reliant on where we put things

small classes become an easier concept to practically implement

---