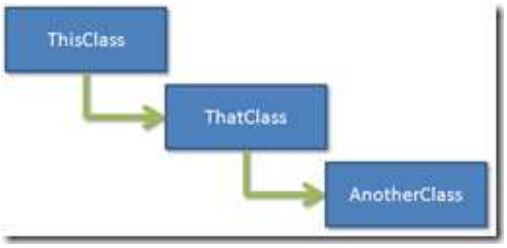
DIP: Creating and Working With A Cloud of Objects by Derick Bailey

A few months ago, I posted some thoughts and questions on the proper use of Inversion of Control (IoC) containers and the Dependency Inversion (DI) principle. Since then, I've had the opportunity to do some additional study and teaching of DI. I've had that light bulb moment for the proper use of an IoC container. I haven't talked about it or tried to present the info to my team(s) yet because I had not verified my thoughts were on the right track, until recently. I got to spend a few hours at the Dovetail office with Chad, Jeremy, Josh, etc, and had the pleasure of being able to pick their brains on some of the questions and thoughts that I've had concerning DI and IoC. In the end, Chad confirmed some of my current thoughts and helped me put them into a metaphor that I find to be very useful in understanding what Dependency Inversion really is. It is a cloud objects that can be strung together into a necessary hierarchy, at runtime.

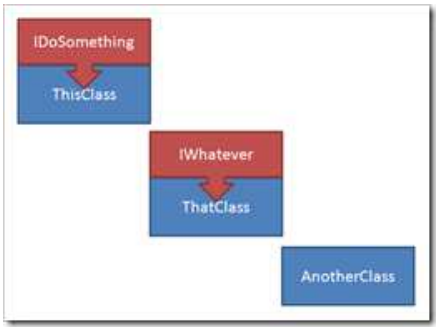
Consider a set of classes that need to be instantiated into the correct hierarchy so that we can get the functionality needed. It's really easy to have the highest level class, the one that we really want to call method on, instantiate the class for next level down. Then have that class instantiate it's next level down, and so-on. For example:



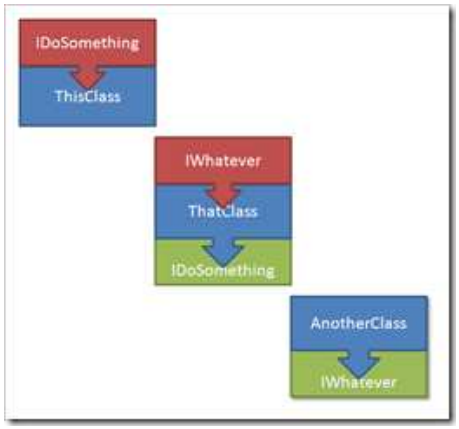
This creates the necessary hierarchy, but breaks the core object oriented principle of loose coupling. We would not be able to use ThisClass without bringing ThatClass along, and we would not be able to use ThatClass without bringing AnotherClass along.

By introducing a better abstraction for each class and putting Dependency Inversion into play, we can break the spaghetti mess apart. We then introduce the ability to use any of these individual classes without requiring the specific implementation of the dependent class.

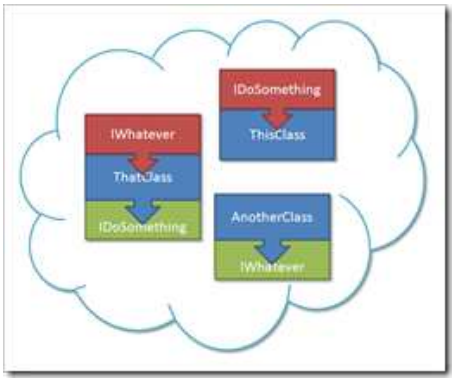
For starters, let's introduce an interface for ThisClass to depend on and an interface for ThatClass to depend on.



Now that that both of these classes can depend on an interface, instead of the explicit implementation of the child object, we need to have the expected child implement the interface in question. For example, we expect ThatClass to be used by ThisClass, so we will want ThatClass to implement the IDoSomething interface. By the same notion, we want AnotherClass to implement the IWhatever interface. This will allow us to provide AnotherClass as the dependency implementation for ThatClass. Our object model now looks like this:



What we have now is not just a set of classes that all depend on each other, but a "cloud" of objects with dependencies and interface implementations that will let us build the hierarchy we need, when we need it.

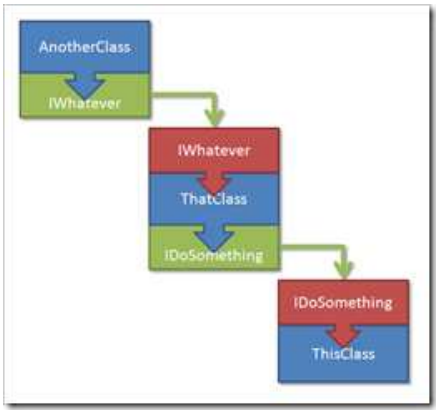


The real beauty of this is we no longer care about the implementation specifics of IDoSomething from ThisClass. ThisClass can focus entirely on doing its duty and calling into IDoSomething when needed. By passing in the dependency as an abstraction, we're able to replace the dependency implementation at any time; runtime, unit test time, etc. This also makes our system much easier to learn, understand and most importantly, easier to change.

Now that we have our cloud of implementations and abstractions in place, we can reconstruct the hierarchy in order to call into ThisClass and have it perform its operations. Here's where Dependency Inversion meets up with Inversion of Control.

- To create ThisClass, we need an implementation of IDoSomething
- ThatClass implements IDoSomething, so we'll instantiate it before ThisClass
- ThatClass needs an instance of IWhatever
- AnotherClass implements IWhatever, so we'll instantiate it before ThatClass
- Once we have AnotherClass instantiated, we can pass it into ThatClass's constructor
- Once we have ThatClass instantiated, we can pass it into ThisClass's constructor

We end up with a hierarchy of objects that is instantiated in reverse order, like this:



We have now successfully inverted our system's construction; each implementation detail is created and passed into the object that depends on it, re-creating our hierarchy from the bottom up. In the end, we have an instance of ThisClass that we can call into with the same basic hierarchy of classes that we started with. The real difference is we can change this hierarchy at any time without worrying about breaking the functionality of the system.

Once we have our Dependency Inversion and Inversion of Control in place, we can start utilizing the existing IoC frameworks to automatically create our hierarchy of objects based on the advertised dependencies (an advertised dependency is a dependency that is specified as a constructor parameter of a class). Tools like StructureMap, Spring.net, Windsor, Ninject and others, all provide auto-magic ways of creating each dependency of the object that is requested all the way up/down the hierarchy. Utilizing one of these IoC containers can greatly simplify our

code base and eliminate the many object instantiations that would start to litter our code. As I said in my previous post, I know all about what not to do with IoC containers. Good IoC usage, though, is another subject for another post.