http://structuremap.github.io/interception-and-decorators/

https://stackoverflow.com/questions/19526474/aop-implemented-with-the-decorator-pattern-over-a-generic-repository

http://blog.ploeh.dk/2018/08/27/on-constructor-over-injection/

Dependency Injection Over Use

At some point in your career, you discover dependency injection. Suddenly unit testing gets easier, your code become less brittle, and you’re using interfaces like never before. Everything is wonderful…

Then you begin to realize you’re using interfaces in places where you know you’ll only ever have one implementation. Your classes have constructors with parameter counts in the double digit range. Eventually you realize that although you aren’t using any anti-patterns, your code doesn’t seem right. It’s acquired a certain code smell. This is a clear indication that you’re over using your IoC container.

There are a few things you can do to help clean up this situation. Some only require minor refactoring. Others will force you to make some significant changes to your architecture. This post will address three possible solutions:

1. Using Aspect Oriented Programming to manage cross-cutting concerns
2. Leveraging a Message Broker for loosely coupled communication between sub-systems
3. Using Façade Services to group related arguments

Using Aspect Oriented Programming to manage cross-cutting concerns

Cross-cutting concerns (or aspects) are parts of your infrastructure that do not fit neatly into a single layer of your architecture. Examples include authentication, logging, error / exception handling, caching, and monitoring. In this post we will focus on exception logging using AOP via the decorator pattern.

It is fairly common to see an implementation of `ILogger` injected into a class via its constructor. This instance is then used to handle logging in the `try` / `catch` blocks that appear throughout the implementation of the class. This practice has two drawbacks. First, it adds an additional parameter to the class constructor that does not help the class do its job. This violates the Single Responsibility Principle. Second, `try` / `catch` blocks are repeated over and over in your code. This violates the DRY principle.

We can solve both problems using AOP. There are two main approaches to implementing AOP: compile-time with aspects and run-time with decorators. As previously mentioned, we will use a decorator to help clean our code.

Decorator example

Using a Message Broker

Looking at your architecture from a SRP perspective, it doesn’t take long to realize your class representing a business process should not be handling side effects generated by that process. If you have a class that is responsible for verifying transactions through a bank account, that same class should not also handle the overdraft process.

With dependency injection this scenario is frequently handled by injecting an implementation of `IOverdraftService` into the implementation of `ITransactionService`. This helps separate concerns; however, it results in `ITransactionService` being coupled to `IOverdraftService`. In the end, does `ITransactionService` really care how or if an overdraft gets handled? The bank certainly does, but `ITransactionService` should not.

The Message Broker provides a simple design pattern that helps further decouple classes. In short, messages (events or commands) are generated by one class and received by any number (including zero) listening classes via the message broker. An implementation of `ITransactionService` can generate `AccountOverWithdrawn` event. An implementation of `IOverdraftService` can then handle that message without either class knowing about the other. This can result in fewer constructer arguments and a cleaner separation of concerns.

Example of Message Broker

Using Façade Services to group related arguments

This is my least favorite approach to cleaning up dependency over injection. However, it does have one side affect that bears mentioning. As you begin to group related arguments implicit elements of your domain may become apparent.

http://blog.ploeh.dk/2010/02/02/RefactoringtoAggregateServices/