Clean Code: Homicidal Maniacs Read Code, Too

An overview of making code readable and maintainable by JeremyBytes.com

Overview

The world is constantly changing; business processes are constantly changing. Our code needs to change along with it. This means that we, as developers, are constantly revisiting our code. If we want our applications to be successful, we need to make sure that they are readable and maintainable. Two years from now, you probably won't remember your original intent in a piece of code. Clean code is a philosophy: We write code in such a way that we don't have to remember the original intent; the intent is in the code itself.

Remember, there is no such thing as write-once code. Our code should be prepared for modification. Today, we'll look at the qualities of clean code as well as some specific techniques to add those qualities. Then we'll spend a bit of time refactoring some existing code to make it easier to understand.

What is Clean Code?

If you ask 10 people what their idea of clean code is, you will get twelve different answers. But you will probably find several things in common.

Readable

All code is readable by machines (that's its ultimate purpose, after all). But clean code is readable by humans. And by humans, I don't mean the gods of programming; I mean mere mortal programmers.

Maintainable

Clean code has good organization and extensibility points. When we need to go back into the code to make updates, we can easily find where the updates need to be made. And ideally, we should be able to make our changes in a single location without needing to make modifications across the entire codebase.

Testable

In addition to the extensibility points, clean code also has good "seams". These are places in the code where we can easily swap out one implementation for another. In our tests, we can swap in fake objects and focus on isolated pieces of code for testing.

Elegant

There's something pleasant about reading clean code. When we run across it, we notice a sort of elegance to the implementation. This doesn't mean complexity – most of the time we find elegance in the simplicity of a solution.

If our code has these properties, then we are prepared for the changes that the world throws at us.

Roadblocks

Clean code sounds like a great thing, right? It's hard to imagine that anyone would not want it. But there are a number of attitudes that keep people from actually writing clean code.

Ignorance

"This is the way I was taught to do it." Some developers just don't know any better. They were taught to write code a certain way, and that's how they do it. This is actually the easiest roadblock to overcome. Generally, once someone sees the benefits of doing things a little bit differently, he is glad to try the new techniques.

Stubbornness

"This is the way I've always done it." This attitude is a little bit harder to change. Some people are reluctant to change. This is especially unfortunate when talking about developers since our industry is all about change. We have to deal with new technologies all the time. We have to deal with changing requirements. Change is part of the job.

Short-Timer Syndrome

"I'm just a contractor. I don't have to deal with this code 6 months from now." If you ever find a contractor with this attitude, run the other way. Good developers are always concerned about their code even after they're gone. Unfortunately, I've seen this attitude in companies that have separate development and support teams. The developers are concerned with slapping the code together as quickly as possible and then toss it over to the support team. Again, we want to avoid such short-sighted views of software.

Arrogance / Job Security

"If you can't understand my code, then you're not a good enough to work on it." Let's face it, some developers have superiority complexes. They like it that they are the only ones who can understand a certain bit of code. It also means that they can't be easily replaced (or so they think). Not a great attitude, but we are bound to run across it in our careers.

Scheduling

"There's no time to make it pretty. Ship it now." This is one of the biggest misconceptions. Some people (often non-technical people who haven't really studied the realities) think that it takes longer to write clean code. Try to avoid taking the short-term view. Yes, I can slap a system together in a weekend, but I'll spend 6 months trying to make it stable and extensible (all while trying to manage production issues and upset customers). Or I can spend 2 weeks building a stable, extensible system that is easy to update when the business changes (meaning fewer production issues and more happy customers).

Procrastination

"I'll clean it up later." This is probably the biggest roadblock to clean code. Developers want to write good code. They recognize ugly code when they see it. But often, a developer will slap something together just to get it working. Then they'll justify releasing it because they will fix it later.

Here's the reality: Later never comes. There's always some other task that takes priority. We need to focus on clean code as we're writing it.

Clean Code Saves Time

The truth is that clean code saves time in the long run. Bugs hide in the dark places of our code. If the code is easy to understand, the bugs will have no place to hide, and we'll be more likely to find them before release. And we all know that finding bugs before release is 100 times better than finding bugs after release.

In addition, automated testing is a big part of clean code strategy. If we find the bugs during the development process – while we're still thinking about a particular piece of functionality – it will be easier to fix. Plus, we save the testing team the trouble of finding and logging those bugs.

Readable code is a huge asset in team environments. When code is understandable, any developer on the team can get quickly up-to-speed on code that he didn't write himself. This is especially desirable in Agile groups where no individual owns a piece of code. The code is owned by the team and all developers are responsible for (and can make changes anywhere in) the entire codebase.

Ultimately, we don't release perfect code (yes, I know it's hard to believe). When we have clean code, it's easier to fix any bugs that are found after we go into production. This leads to a faster response to customers and ultimately a better product.

We can't take a short-term view of software. We need to be thinking about the future.

The Problem with Defining Clean Code

So, we decided that in order to be clean code, our code needs to be readable, maintainable, testable, and elegant. Now we have to face a problem: all of these qualities are subjective. Readable by whom? Testable by which methodologies?

I have a rule of thumb that I follow when writing code:

Imagine that the developer who comes after you is a homicidal maniac who knows where you live.

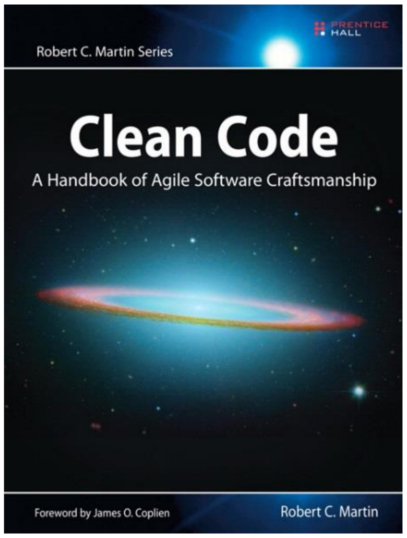
This eliminates the tendency toward the short-timer syndrome that we saw above. Think about the developer who is going to have to maintain and update this code after you're gone. Now, put yourself in his shoes.

If you've been a developer for any length of time, you've probably had to go in to change code that was written by someone else. Did you end up cursing the whole time? ("What was this moron thinking; this code is awful!") Or did you make your updates confidently because the code was easy to understand and had good unit test coverage?

The reality is that the former is more common; but we'd like it to be the latter. Clean code has to start somewhere. If we keep the next developer in mind when we're writing our code, we're more likely to write something that's understandable. Plus, if he really is a homicidal maniac, we want to keep him happy, right?

Some Guidance

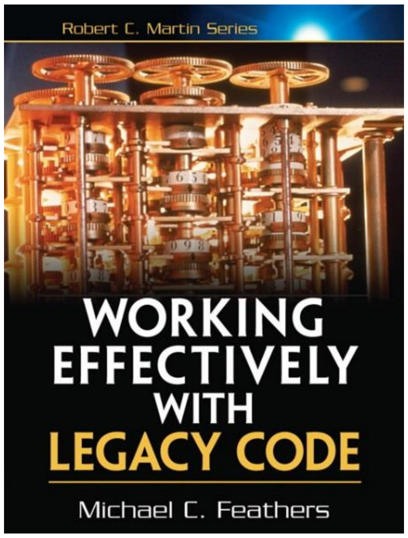
Fortunately for us, there are folks who have written some really good books to give us guidance to writing clean code. The first of these is *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin (Uncle Bob). I posted a review of this book on my blog: http://jeremybytes.blogspot.com/2012/12/book-review-clean-code.html.


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Clean Code gives a number of specific techniques and things to watch out for. This book was a little disconcerting to read. It is full of common sense – things that make you say, "Of course, that's how you should write that code. It's obvious." But then immediately you think, "But that's not how I've been doing it. Why haven't I been doing it that way?"

So, it's filled with tons of practical advice (and we'll be looking at we'll look at a few specific items here). In addition, Martin provides a list of code "smells". A code smell is usually recognized based on your experience. You look at a piece of code and something doesn't "smell" right. There's something wrong with the code that you might not be able to put your finger on, but it's there.

Martin gives some concrete advice to recognizing code smells and what can be done to improve the code.



Another great book is *Working Effectively with Legacy Code* by Michael C. Feathers. I have a review for this book as well: http://jeremybytes.blogspot.com/2013/02/book-review-working-effectively-with.html.

This book is focused on how to make changes to existing (non-clean) code. It gives techniques on how to bring the legacy code under test so that we can make changes confidently and ensure that we aren't breaking any existing features. If we don't have tests, then we don't really know what the code does. And if we don't know what the code does, how can we make changes and know that we didn't break anything?

We can't snap your fingers and make an entire codebase into clean code. But we can slowly transform an existing project by cleaning up the pieces that we need to make changes to.

So, let's look at a few specifics. We can't cover everything here. I've picked out a few things that have helped me.

The DRY Principle

One extremely important principle is DRY: Don't Repeat Yourself. Whenever you are tempted to copy/paste a block of code somewhere else, stop. Copy/Pasta only leads to spaghetti code. Remember:

Don't Repeat Yourself

Duplication is dangerous. If we fix a bug in a piece of code, it's highly unlikely that we will look for copies of that same code in different parts of the application. Another problem is that duplication hinders readability. If we see the same 5 lines of code over and over again, those lines just become "noise" that gets in the way of understanding the rest of the code.

If you find that you need the same block of code somewhere else, don't copy it. Instead, create a new method to encapsulate that code. Then call that method from multiple places in your application.

But what if we need *almost* the same code in multiple places. We can encapsulate the method and provide a parameter. The method would behave differently based on this parameter. We can also look at some more advanced features like using virtual methods to polymorphically change the behavior of a class.

Naming

Many times developers don't put enough thought into what they call things. If we have something called a1 or a2, what are we supposed to do with it? This is where we need to start thinking about intentional naming.

HELLO my name is a2
HELLO my name is bFlag
HELLO my name is Inigo Montoya You killed my Father. Prepare to die!
HELLO my name is a1
HELLO my name is Param
HELLO my name is ldtstmgr
HELLO my name is recdptrl
HELLO my name is theList

Intentional Naming

When we name something in our code (whether it is a field, parameter, function, or class) we want the name to show the intent of the item. This will help other developers use the item correctly.

Let's take a look at an example. Say we have a property called theList. This name isn't as bad as a1 because at least it tells us what it is (a collection of something). But it doesn't tell us anything about the intent. What am I supposed to use this collection for? It would also be helpful to know what types of items are in the collection. Is it a list of random objects? A list of employees? This name just doesn't tell us enough.

What if we change the property to ProductList? Well, this is a bit better. It tells us what types of things are in the collection. But we still don't know the intent. Is this a list of products that appear in a price list? Is it a list of products that are included for an order? Is it a list of products that we just received into inventory?

What about ProductCatalog? This is much better. The word "catalog" tells the intent – this is a list of products that we sell that someone can select from. We've got a couple of good things with this name. First, as mentioned, we have the intent. Next, we know that this is a collection of items (implied by the word "catalog"). Third, we know what kind of items are in the catalog (products).

The only thing that could make this a bad name is if ProductCatalog actually contained a list of employees that start their shift on Saturdays (i.e. something other than a catalog of products).

Other Naming Tips

There are some "well, that's obvious" tips for naming that aren't always followed. One is to use nouns for "things" (such as variable, fields, parameters, and classes) and to use action verbs for methods and functions.

Things can get a little more ambiguous depending on your development environment. In the C# world, we commonly see properties called things like `IsEnabled` or `IsReadOnly` (particularly in the XAML world). These are somewhere in between because they are technically verbs, but they are verbs of state rather than verbs of action. Consistency is important. So, if we were to make another property to go alongside these (such as a property that tells whether the current item can be cached), we should follow the convention and name it similarly, such as `IsCacheable`.

Names should also be pronounceable and unambiguous. A name like `recdptrl` isn't a real word and it doesn't have enough vowels to figure out how to pronounce it (which it difficult to use in conversation). Plus, what does it mean? Is it "received patrol"? Or maybe "record department role"? It's okay to spell it out. Most of us are using IDEs that give us some form of code completion – we don't have to worry about having to type long identifiers all the time.

We should also consider using longer names for larger scopes. When we have an application-level object, it's going to be around for a while. We will probably be referencing it in a number of different modules or classes. This means that we want to make sure that the object name makes sense in all of those different areas. We should think about using a very descriptive name like `CurrentUserSecurityRoles`.

With smaller scopes, we can get away with names that are a bit less descriptive. For example, we might have a method parameter named `securityRoles`. Since it's only used within the scope of that method, it's much easier to keep track of. And, depending on what the method does, we might not need the context of "current user". The method may act on any security roles regardless of whether they belong to the current user. It all depends on the context.

It is okay to use `i` as a counter or indexer. So, we don't need to feel like we need to have a longer indexer name for a `for` loop that iterates through an array. We're shooting for readability here. Contrast that with using an `l` (lowercase "L") as a counter. This would be horrible since in many fonts a lowercase L looks exactly like the numeral 1.

Naming Standard

So, we've looked at quite a few tips about how we select meaningful names. But what about format? Do we use camel casing (like `securityRoles`)? Or Pascal casing (i.e. initial caps, like `SecurityRoles`)? Or all lowercase with underscores (`security_roles`)?

The truth is that what format we use really doesn't matter. But there are 2 things that do matter:

1. **Have a Standard**
2. **Be Consistent**

If you don't already have a formatting standard, come up with one (or use one that someone else created). This will go a long way toward making your code readable. You can use something like StyleCop (http://stylecop.codeplex.com/) in Visual Studio to enforce styles and formatting within the code.

A brief word about Hungarian Notation: it's no longer needed. If you aren't old enough to remember Hungarian notation, that's okay. The basic premise is that each identifier has a prefix that tells the type of the object. For example, you might have "bFlag" to denote that this is a byte. With modern IDEs, we don't need this. Our development environment tells us what the types are (assuming we're using a strongly-typed language), and the compiler enforces how we use our objects and will error if we try to use a type incorrectly.

Robert C. Martin takes this a step further and suggests that we should remove prefixes from things like interfaces. It is a common practice to prefix an interface with a capital `I`, so we end up with things like `IList` or `IDisposable`. He argues that since the developer who is consuming the API doesn't need to make a distinction between the interface and a concrete class (which is exactly the purpose of using interfaces), there shouldn't be any distinction in the naming, either.

However, in the .NET world, all of the interfaces in the framework are prefixed with the letter `I`. In order to maintain consistency with the environment, we should follow this convention when we are creating our own interfaces. Again, we're trying to write code that a developer expects to see – that's a key part of maintaining readability.

Comments

Comments can be an asset, but often they are just noise – things that get in the way of looking at the actual code. There are a number of ways that comments are used incorrectly, and we need to get away from these.

First Rule: Comments Lie

The reality of the situation is that comments are often incorrect. We move code around or change its functionality, but we rarely go in to update the comments. This means that many comments are stale and don't apply to the current code. Or even worse, they are orphaned and actually apply to code that has been relocated to a different method or class. This should give us incentive to avoid unnecessary comments.

Second Rule: Comments Do Not Make Up For Bad Code

Many developers use comments to explain their code. Here's an example:

```
        // Determine if End of Day Time for Last Date
        // has been reached
        // If Last Date is null use Converted Date
        // Based on Today's Date > Last Date
        //  And Curr Time >= End of Day Time
```

This is basically pseudo-code (and quite honestly, I don't really understand it). The problem is that the developer felt that the code wasn't understandable, so he added this comment block.

Instead of writing comments to try to make up for bad code, we should focus on making the code readable. If code is not understandable on its own, it needs to be re-written.

Good Comments

Comments should not tell what the code does (the code should do that). But comments may be used to give the intent of the code. For example, we may have a fairly cryptic regex statement (some people would argue that "cryptic" describes pretty much every regex statement). In this situation, a comment that gives a sample input goes a long way toward showing the intent of the regex statement.

Comments can also be used to provide warnings or consequences. I have first-hand experience with this. I took over some code and saw a piece that looked odd to me. The code created deep copies of a collection and assigned them to multiple properties. (A deep copy is one where all of the elements are copied from one collection to another.) This looked wasteful to me, so I changed it to use a reference assignment. This way, all of the properties pointed to the same physical object with no need to go through the overhead of making copies.

This broke the code pretty quickly, and I reverted my changes (hurray for source control!). But before I left that section of code, I left a comment that told why the deep copy of the collection was necessary. This will hopefully act as a warning flag to any future developers who get the desire to "optimize" the code.

Comments can also be used for `TODO`s. Many development environments support `TODO` comments that are automatically parsed into a task list. We should take advantage of this. However, these comments should be temporary. As soon as the task is completed, the `TODO` should be removed. Visual Studio has a habit of replacing `TODO` with `DONE` when you mark a task as completed. Remove the `DONE` comment. Once the task is complete, it is just noise. If we really need the history, it's available in source control.

Bad Comments

We've already mentioned that comments should not describe what the code is doing, but there are a few other ways that comments are misused.

One misuse is "journaling comments". These are comments like: `jjc – 4/18 – Added tax calculation`. They don't add anything to the understandability of the code. If we really need to know what was changed by whom on which date, we just need to check our source control – that's what source control is for. We need to make sure we're using the tools that we have at hand.

A big misuse of comments is to comment out code. Do not comment out code. Ever. Period. (Okay, so it's okay to comment out code while you're making modifications on your machine, but never check it in to source control that way.)

The problem with commented out code is that no one will ever touch it again. When I run across a piece of commented code, my first thought is "Is this really supposed to be commented out?" Maybe the developer was doing some testing and just forgot about this code. Maybe it's important to the application. But I won't change it, because I don't know. So, I just leave the commented out code as-is.

My next thought is "Maybe this code is really important to someone". Again, in that case, I'm not going to touch it. I'll just leave it alone.

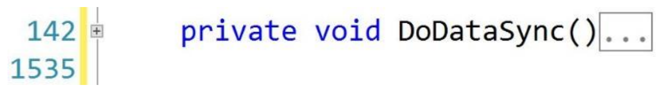
Don't be afraid to delete code. It's okay. If we ever need it back, we can retrieve it from source control – that's what it's for. Use the tools in the way they are intended. Commented out code just creates noise and uncertainty in the codebase.

Functions and Methods

Before talking about how we should write functions and methods, let's talk about what not to do. Here's an experience that I had.

I was tasked with writing "Version 2" of a data transfer service. I had a set of requirements documents, but one of the specifications was a little vague. I didn't have anyone to ask about this function because it was already after hours in their time zone. So, I decided to take a look at the current system to see what I could figure out.

So, I popped open the file and found a bunch of regions in the code. "That's great," I thought, "Things are organized into different sections." Then I opened the region that I thought contained the code I needed. Sure enough, there was a method called `DoDataSync`. That sounded promising. But when I opened the method, it was rather long. I turned on line numbering, and here's what I saw:



```
 142      private void DoDataSync()...
1535
```

Yes, that single method goes from line 142 to line 1535. It was 1393 lines long! After looking through it (just a bit), finding nested if/else statements, comment blocks that were difficult to understand, and a few "gems" like this:

```
        // *** This is copy of the section from above
        // *** with some slight changes
```

I decided that I wasn't going to find the answer I was looking for in the existing code. (BTW, I never did find out what those "slight changes" were.)

Good Functions and Methods

There are a couple of qualities of good functions/methods. The first is that they should be short. A method should fit on a single screen (at reasonable font-size/resolution). Ideally, it should be no longer than 10 lines (although, we don't often hit this ideal).

The reason for short functions is simple: we have limited brains. We can't keep an entire codebase in our heads. This means that we need to context-switch to figure out where we are in the code and what we're doing. Keeping an entire method "in view" helps our cognitive process.

Good functions do one thing. Let's repeat that:

Do One Thing

Now, someone might argue that the 1,300 line method mentioned above does one thing: it transfers data. But it was actually pulling data in multiple formats from multiple sources, transforming it in specific ways (depending on the data format), and then pushing it to a destination system. In the "Version 2" system that I worked on, I actually ended up with about a dozen separate classes to handle the work (in small pieces – more on this later).

Now, software is complex. Often methods need to be complex. But there are a few techniques that can be used to break down the complexity and make our code much more readable. One of these is to have multiple levels of methods in a hierarchy.

Method Hierarchy

Complex methods should be broken down into different levels of functionality. The High-Level methods give an overview of what's going on; they make calls to the Mid-Level methods. The Mid-Level methods give you a bit more detail, but not too much; they make calls to the Detail methods. Detail methods are the "weeds" of the functionality. The number of levels we have depends on the amount of complexity.

We'll look at a full example of this when we get to the refactoring sample below. Here's a small taste.

Original Method

```
public void Initialize()
{
    if (!_container.IsRegistered<IPersonService>())
        throw new MissingFieldException(
            "IPersonService is not available from the DI Container");
    _service = _container.Resolve<IPersonService>();

    if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
        throw new MissingFieldException(
            "CurrentOrder is not available from the DI Container");
    _model = _container.Resolve<CatalogOrder>("CurrentOrder");
    RefreshCatalog();
}
```

Updated High-Level Method

```
public void Initialize()
{
    _service = GetServiceFromContainer();
    _model = GetModelFromContainer();
    RefreshCatalog();
}
```

Updated Detail Methods

```
        private CatalogOrder GetModelFromContainer()
        {
            if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
                throw new MissingFieldException(
                    "CurrentOrder is not available from the DI Container");
            return _container.Resolve<CatalogOrder>("CurrentOrder");
        }

        private IPersonService GetServiceFromContainer()
        {
            if (!_container.IsRegistered<IPersonService>())
                throw new MissingFieldException(
                    "IPersonService is not available from the DI Container");
            return _container.Resolve<IPersonService>();
        }
```

Both the Original and Updated methods have exactly the same code. But when I look at the original, I have to spend time trying to figure out exactly what it's doing. It's hard for my brain to pick out the important parts.

But when I look at the updated High-Level method, I can see that this is doing 3 things: it's assigning the `_service` field, assigning the `_model` field, and then refreshing the catalog. If I need to know exactly where `_service` or `_model` come from, I can drill down into the detail methods. But more importantly, if I don't need that level of detail, I never have to look at it.

We'll see some more examples of this when we do some actual refactoring.

Classes

Object Oriented Programming (OOP) has been around for many years. If we're using OO languages (like C#), we can take advantage of a huge number of best practice resources. One of these is the S.O.L.I.D. principles. This is a set of 5 principles that apply to good OO design. I would highly recommend doing some internet browsing if you haven't heard of these before. (As a side note, I mention 4 of the 5 principles in "Dependency Injection: A Practical Introduction" available on my website: http://www.jeremybytes.com/Demos.aspx#DI.)

Do One Thing

Classes should do one thing (and do it well). This is also referred to as the Single Responsibility Principle (the "S" in S.O.L.I.D.). Your class should have only one reason to change. If there is more than one reason to change, then it's time to start looking at moving some of that functionality to other classes or methods.

One way to accomplish this is to outsource functionality through the use of delegates. A delegate can give your class an extensibility point that can be used to inject non-core functionality. This is an example of the Open/Closed Principle (the "O" in S.O.L.I.D.). Your class should be open for extension but closed for modification (meaning we can inject the functionality without needing to modify the class itself). For more examples of this, be sure to check out "Get Func<>-y: Delegates in .NET" available on my website: http://www.jeremybytes.com/Demos.aspx#GF.

Another way to accomplish this is through Dependency Injection (DI). This is a way to make resources available to a class without making the class responsible for the resources themselves (and generally through abstractions). This is an example of the Dependency Inversion Principle (the "D" in the S.O.L.I.D. principles). For more information on Dependency Injection, refer to the session mentioned above.

Work in Small Chunks

One thing that should be clear by now is that we want to work in small pieces – whether this is small methods or small classes. If we're thinking small, we generally end up with readable, maintainable, and testable code.

One technique that encourages working in small pieces is Test-Driven Development (TDD). If you want more information on TDD, I would highly recommend *Test-Driven Development by Example* by Kent Beck (my review is available here: http://jeremybytes.blogspot.com/2013/03/book-review-test-driven-development-by.html).

TDD is built around the Red – Green – Refactor cycle. In TDD, the first thing we do is write a failing test for the functionality we want to implement (Red). The test fails because it is a test for code that doesn't exist yet. The next step is to write just enough code to make the test pass (Green). We're not building the mansion, we just placing one brick. The last step is Refactor. This is the process of cleaning up our code: removing duplication, combining classes, adding abstractions, etc.

What this encourages is building very small pieces. Beck does a great job of showing just how small those pieces can be. What I found by reading this book is that I was thinking too big. The emphasis is small pieces, built incrementally.

Which leads to another piece of wisdom:

If you aren't writing incremental code,
you are writing excremental code.

(I didn't make this up, but I don't have anyone to attribute it to, either.)

Refactoring and Unit Testing

Refactoring is the process of making code better without changing the functionality. But this leads to a big question: How do we know we aren't changing functionality?

The answer to this is unit testing. If we don't have unit tests, we don't really know what our code is doing. We may have an idea of what it's doing, but we don't have any proof that it's actually doing it. With valid unit tests in place, we know what our code is actually doing.

If we want to refactor safely and confidently, we must have valid unit tests in place. If we have tests in place, we know immediately whether our code changes have impacted the functionality. If all of the tests pass, we're good. If we have failing tests, we need to review our updates to see what we did wrong.

This is what Michael Feathers' book is all about: how to bring existing code under test so that we can safely change it. Feathers covers a number of different scenarios and offers practical advice for how to add valid tests. Changes to the code are made only after a set of valid tests are available.

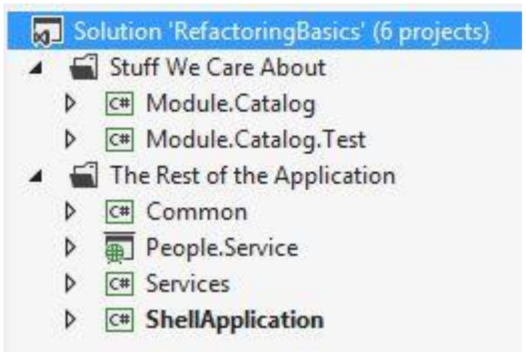
Real-World Refactoring

So, now it's time to refactor some existing code to make it cleaner. The code sample is based on a real-world project. The sample takes the same basic format of the original solution but has been simplified some (the original solution had 40 projects; the sample has 6).

You can download the source code for the application from the website: http://www.jeremybytes.com/Demos.aspx#CC. The sample code we'll be looking at is built using .NET 4 and Visual Studio 2012. The solution is called "RefactoringBasics.sln". Two versions are included: a "starter" solution (if you want to follow along) as well as the "completed" code.

The Project Layout

The solution is broken into 6 projects, but there really only 2 that we care about:



The scenario is that we have an application that is made up of multiple isolated modules (just one module in our sample). The application uses the Model-View-ViewModel (MVVM) design pattern for presentation. (For more information on MVVM, you can refer to this article: http://jeremybytes.blogspot.com/2012/04/overview-of-mvvm-design-pattern.html.)

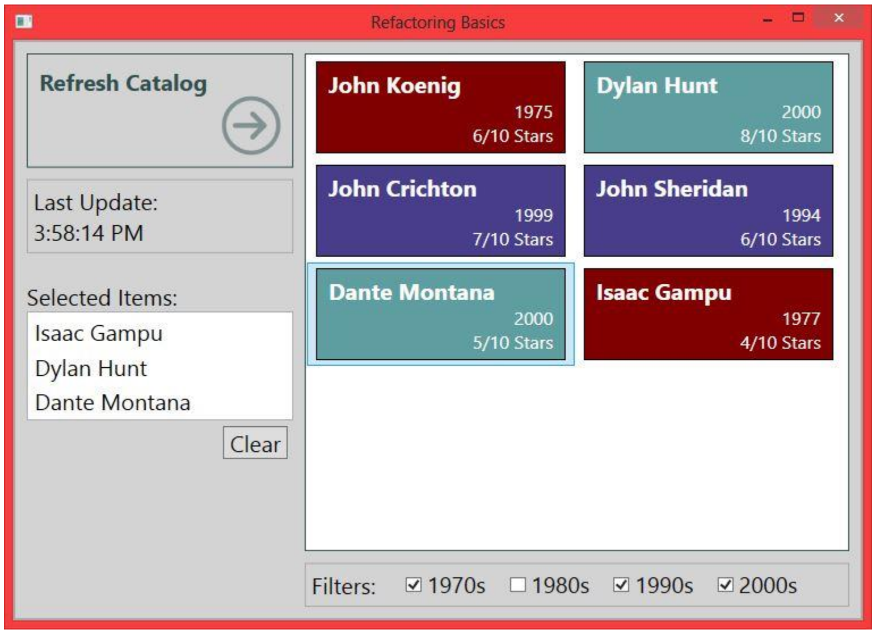
In the "Stuff We Care About" solution folder, we have the projects we'll be working with. The `Module.Catalog` project contains both a view (`CatalogView.xaml`) and a view model

(`CatalogViewModel.cs`). We really only care about the view model for our refactoring. The `Module.Catalog.Test` project contains the unit tests for `CatalogViewModel`.

In "The Rest of the Application" folder, we have several other projects including the service layer, our models, and the shell application that loads up the Catalog module.

Application Functionality

Here is the application in action:


Refactoring Basics
Refresh Catalog
Last Update:
3:58:14 PM
Selected Items:
Isaac Gampu
Dylan Hunt
Dante Montana
Clear
John Koenig 1975 6/10 Stars
Dylan Hunt 2000 8/10 Stars
John Crichton 1999 7/10 Stars
John Sheridan 1994 6/10 Stars
Dante Montana 2000 5/10 Stars
Isaac Gampu 1977 4/10 Stars
Filters: ☑ 1970s ☐ 1980s ☑ 1990s ☑ 2000s

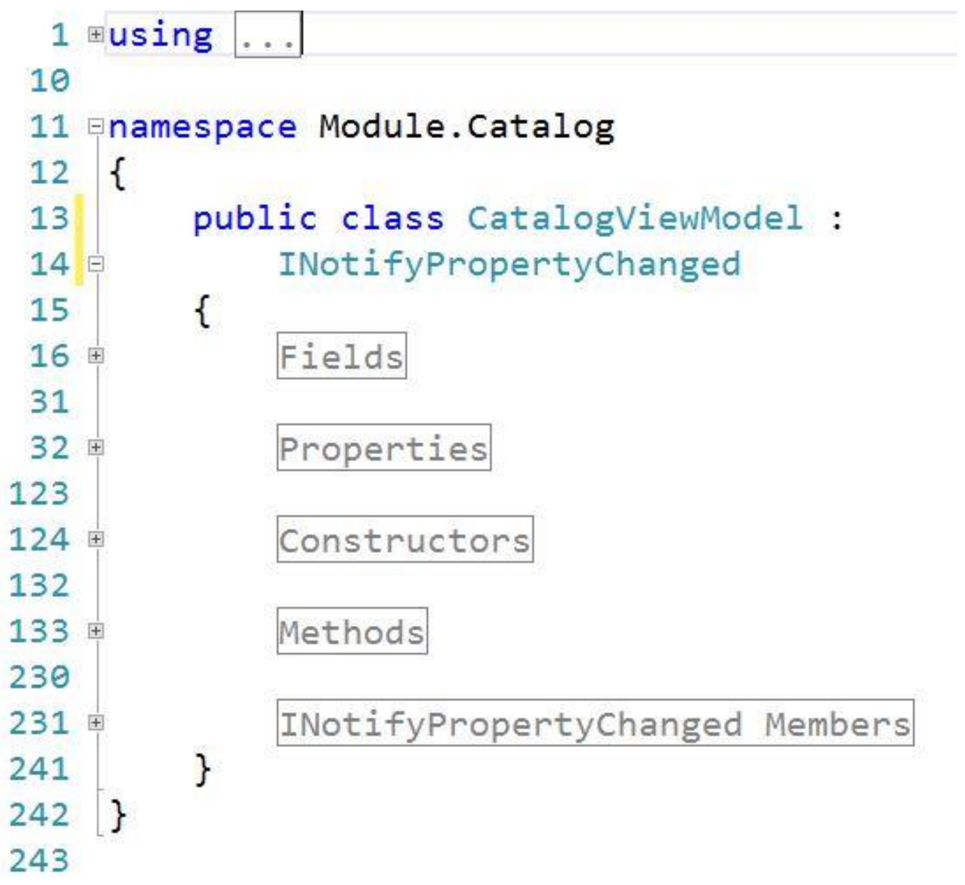
When the application starts up, the data (the Catalog) is automatically refreshed from the service. These are the items that appear in the primary list box on the right. The "Last Update" shows the time that the data was refreshed from the service. There is a 10 second cache built into the system. So, if you click the "Refresh Catalog" button 2 times in a row, you will notice that the Last Update time does not change. However, if you wait for more than 10 seconds and click the button, you will see a brief pause (as the primary list box goes blank), then the list will update and the Last Update time will be different.

At the bottom of the screen are a set of filters based on the decade. When you check/uncheck one of the filters, items from that decade are added/removed from the list. Whenever the "Refresh Catalog" button is clicked, the filters are automatically reset. They are reset whether the cache is used or the data is refreshed from the service.

If you double-click and item in the list box on the right, it will appear under "Selected Items" on the left. You can remove an individual item by double-clicking in the "Selected Items" list or remove all items with the "Clear" button. The "Selected Items" list is not affected by refreshing the catalog.

File Overview: CatalogView.cs

We will be making changes to the CatalogViewModel.cs file. If we look at the regions, we get a general feel for this class:



```
using ...

namespace Module.Catalog
{
    public class CatalogViewModel :
        INotifyPropertyChanged
    {
        Fields

        Properties

        Constructors

        Methods

        INotifyPropertyChanged Members
    }
}

```

The `Fields` region includes some of our internal fields, including `_service` which holds a reference to the `PersonService` (where we get the data), `_model` that holds a reference to an object that holds the Selected Items that we saw in the UI, and also `_container` that holds a reference to our Dependency Injection container (which is a Unity container). The `Fields` region also holds the backing fields for our properties.

The `Properties` region holds the public properties that are data-bound to our View. This includes the `Model` for the Selected Items, `Catalog` for the primary list, `LastUpdateTime`, and the filters used for the checkboxes.

The `Constructors` region holds a single constructor that gets a reference to the Unity DI container. This container is provided by the `ShellApplication` on startup. If you want more information on Unity and Dependency Injection, refer to “Dependency Injection: A Practical Introduction”: http://www.jeremybytes.com/Demos.aspx#DI.

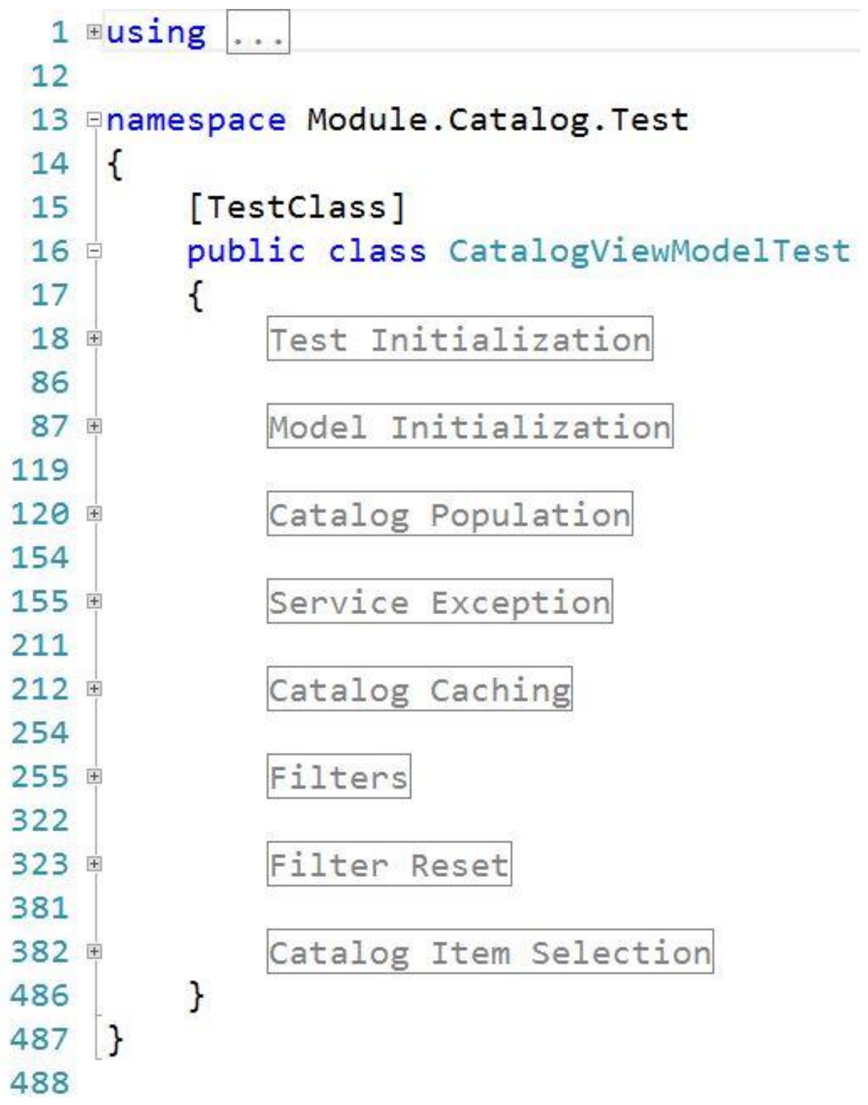
The `Methods` region is where we will be spending most of our time. This holds the public methods for the class, including `Initialize` (which gets called at startup) and `RefreshCatalog` (which is hooked up to the button in the View). `AddToSelection`, `RemoveFromSelection`, and `ClearSelection` are used to manage the Selected Items list. And we have one private member, `RefreshFilter` which is used to filter the primary list.

We'll see more details on the `Methods` region as we start refactoring the code.

The last region is for the `INotifyPropertyChanged` interface members. This is boilerplate code that is used for the interface. The interface itself is implemented so that the View (UI) controls will update properly when the databound objects are changed.

File Overview: CatalogViewModelTest.cs

As mentioned earlier, valid unit tests are critical to being able to refactor safely and confidently. These are found in the `CatalogViewModelTest.cs` file:



```
using ...

namespace Module.Catalog.Test
{
    [TestClass]
    public class CatalogViewModelTest
    {
        Test Initialization

        Model Initialization

        Catalog Population

        Service Exception

        Catalog Caching

        Filters

        Filter Reset

        Catalog Item Selection
    }
}

```

We won't be looking at the specifics of unit testing in this sample. What we need to note here is that we have good coverage of all of the functionality of our View Model. We have a total of 19 tests that cover all of the public members in the class. The solution is also configured to run the unit tests on every build, so if you have the Test Explorer window open in Visual Studio (from the menu, select Test -> Windows -> Test Explorer), you'll see the tests update after every build. In a multi-monitor setup, I like to have the Test Explorer open on a separate monitor. That way I can see the details of the test runs (since they are updated constantly with each build).

Lines of Code

One thing you might notice is that the Test file has 488 lines while the ViewModel file only has 243 lines. This means that we have twice as many lines of test as we do actual code. This is one of the things that puts off managers ("Why should I have my developers write 3 times as much code as they need to."). However, this is a short-sighted view. Just because we have 3 times as much code doesn't mean that it took us 3 times longer to write. In fact, it's possible to write code much more quickly with unit tests in place because we can confidently move forward knowing that our code is working.

And ask yourself this question: what developer is limited by typing speed? We spend a lot more time during the day thinking than typing. If we were really limited by typing speed, then we should hire a typist to do all our coding for us.

Not Bad Code

Before we get into the details of refactoring, I want to point out that this is not bad code. It's perfectly reasonable. We have a class that has about 250 lines of code – that's a reasonable size (for ViewModels, I'm comfortable to about 400 lines of code – there's a lot of "lines" in properties).

When we look at the methods, none of them are particularly long. The longest method that we have is about 30 lines – still within the "reasonable" threshold (and extremely reasonable compared to 1,300 lines).

So, the goal here isn't to rip apart code because it's bad. Our goal here is to simply make good code even better. It's the difference between removing graffiti (an eyesore) and planting flowers (something prettier). But as we'll see, the improvements are well worth the effort.

Our First Refactor

So, our goal is to make our View Model more readable and maintainable. (We don't really need to worry about "testable" at this point since it already is well tested.)

We'll start with the `Initialize` method that we saw earlier:

```
        public void Initialize()
        {
            if (!_container.IsRegistered<IPersonService>())
                throw new MissingFieldException(
                    "IPersonService is not available from the DI Container");
            _service = _container.Resolve<IPersonService>();

            if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
                throw new MissingFieldException(
                    "CurrentOrder is not available from the DI Container");
            _model = _container.Resolve<CatalogOrder>("CurrentOrder");
            RefreshCatalog();
        }
```

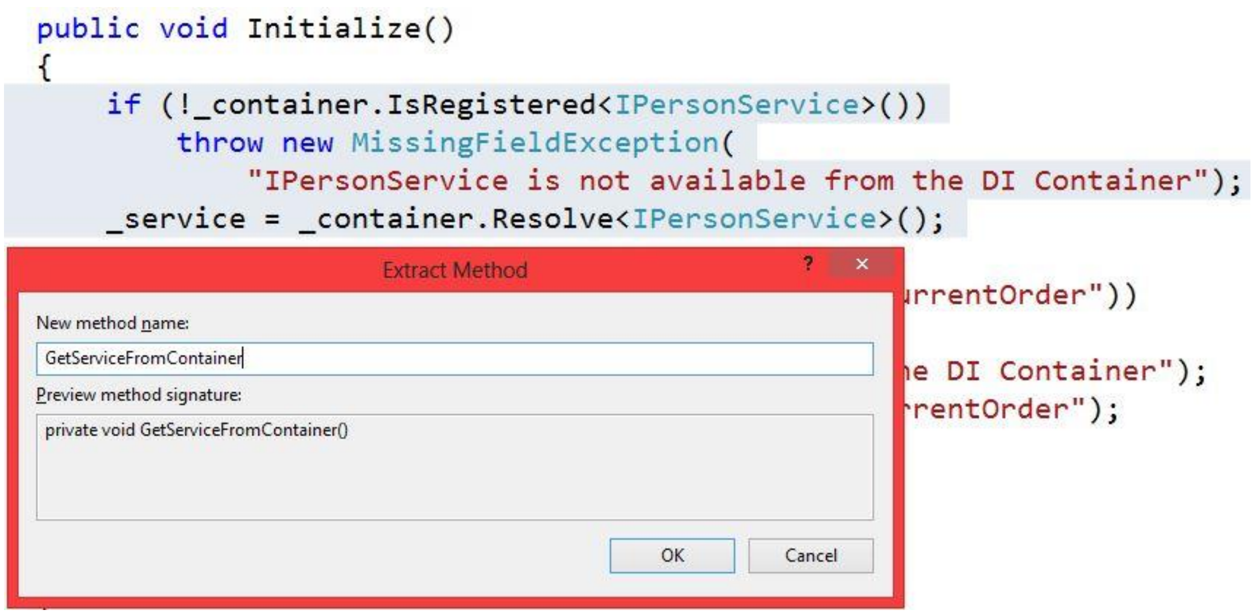
As we looked at briefly earlier, we want to turn this into a hierarchical method – letting `Initialize` hold the high-level functionality and move the details into their own methods.

Now, there are tons of tools that help with refactoring including ReSharper, Just Code, and Code Rush (to name a few). We'll be using the refactoring tools that are built into Visual Studio for this demo (since everyone has access to those). But you will get more functionality with a third-party tool.

Let's look at the first 4 lines of code. We start with an `if` statement. This checks to see if `IPersonService` is registered with the dependency injection container. If it is not registered, then it throws an exception. Otherwise, it will retrieve the item from the container (using the `Resolve` method) and assign it to the `_service` field.

But this is more detail than we really need in the `Initialize` method. These 4 lines are getting the service from the container and assigning it to the `_service` field. So, let's extract a method that says just that.

We'll start by highlighting the first 4 lines in the body of the method. Then we can right-click, select "Refactor", then "Encapsulate Method" (note that the keyboard shortcut is Ctrl+R, Ctrl+M). Then we get a chance to name our new method. Let's choose `GetServiceFromContainer`. This is how things should look:



```
public void Initialize()
{
    if (!_container.IsRegistered<IPersonService>())
        throw new MissingFieldException(
            "IPersonService is not available from the DI Container");
    _service = _container.Resolve<IPersonService>();
```

```
urrentOrder"))
ne DI Container");
rentOrder");
```

Extract Method

New method name:

GetServiceFromContainer

Preview method signature:

private void GetServiceFromContainer()

Here's the result when we click "OK":

```
        public void Initialize()
        {
            GetServiceFromContainer();

            if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
                throw new MissingFieldException(
                    "CurrentOrder is not available from the DI Container");
            _model = _container.Resolve<CatalogOrder>("CurrentOrder");
            RefreshCatalog();
        }

        private void GetServiceFromContainer()
        {
            if (!_container.IsRegistered<IPersonService>())
                throw new MissingFieldException(
                    "IPersonService is not available from the DI Container");
            _service = _container.Resolve<IPersonService>();
        }
```

At this point, we can build successfully, and all of our unit tests will pass as expected.

I'm not completely happy with this, though. I would rather have the assignment of the `_service` field in the high-level method (rather than the details). So, we'll make a few changes to the methods.

```
        public void Initialize()
        {
            _service = GetServiceFromContainer();

            if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
                throw new MissingFieldException(
                    "CurrentOrder is not available from the DI Container");
            _model = _container.Resolve<CatalogOrder>("CurrentOrder");
            RefreshCatalog();
        }

        private IPersonService GetServiceFromContainer()
        {
            if (!_container.IsRegistered<IPersonService>())
                throw new MissingFieldException(
                    "IPersonService is not available from the DI Container");
            return _container.Resolve<IPersonService>();
        }
```

Notice that rather than returning `void`, our detail method now returns `IPersonService`, and the last statement is a `return` rather than an assignment. Then back in our high-level method, we assign the result of the method to the `_service` field.

All of our tests still pass. (Whoo-hoo!)

Let's to the same thing with the next 4 lines in the `Initialize` method. We'll highlight the lines, then press Ctrl+R, Ctrl+M to extract the method. We'll choose `GetCurrentOrderFromContainer` as the method name and click "OK". Here's our updated code:

```
        public void Initialize()
        {
            _service = GetServiceFromContainer();
            GetCurrentOrderFromCatalog();
            RefreshCatalog();
        }

        private IPersonService GetServiceFromContainer() ...

        private void GetCurrentOrderFromCatalog()
        {
            if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
                throw new MissingFieldException(
                    "CurrentOrder is not available from the DI Container");
            _model = _container.Resolve<CatalogOrder>("CurrentOrder");
        }
```

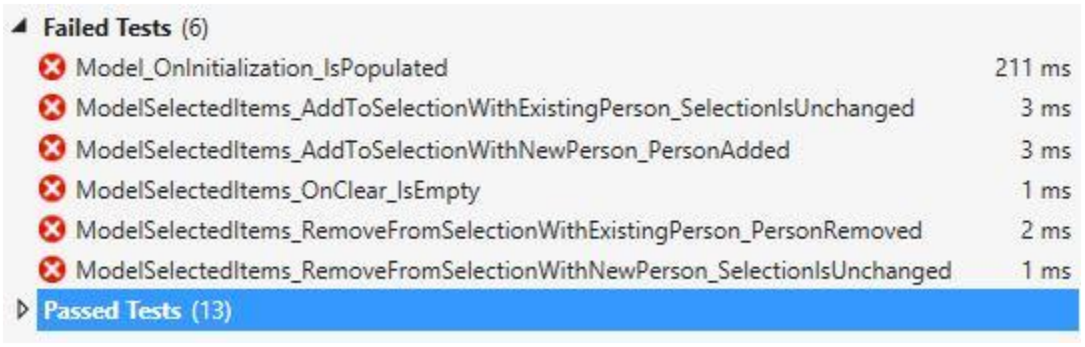
Build and run tests: All tests pass.

Then we'll convert the method to return a `CatalogOrder` rather than `void`:

```
        public void Initialize()
        {
            _service = GetServiceFromContainer();
            GetCurrentOrderFromContainer();
            RefreshCatalog();
        }

        private IPersonService GetServiceFromContainer()...

        private CatalogOrder GetCurrentOrderFromContainer()
        {
            if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
                throw new MissingFieldException(
                    "CurrentOrder is not available from the DI Container");
            return _container.Resolve<CatalogOrder>("CurrentOrder");
        }
```

Build and run tests. Uh, oh. 6 tests failed. It looks like we have a problem.

And this is exactly why it is vital to have good unit tests in place before we start to refactor.



```
Failed Tests (6)
  Model_OnInitialization_IsPopulated                                         211 ms
  ModelSelectedItems_AddToSelectionWithExistingPerson_SelectionIsUnchanged     3 ms
  ModelSelectedItems_AddToSelectionWithNewPerson_PersonAdded                   3 ms
  ModelSelectedItems_OnClear_IsEmpty                                           1 ms
  ModelSelectedItems_RemoveFromSelectionWithExistingPerson_PersonRemoved       2 ms
  ModelSelectedItems_RemoveFromSelectionWithNewPerson_SelectionIsUnchanged     1 ms
Passed Tests (13)
```

This is generally where I'm glad I spent some time naming my unit tests. The format that I use for naming tests is based on a recommendation from Roy Osherove (author of *The Art of Unit Testing with Examples in .NET*). It is a 3-part naming convention with the parts separated by underscores.

The first part is the item under test. For the first failed test, it tells me that `Model` is what I was testing.

The second part is the action performed in the test. For the first failed test, `OnInitialization` tells me that it was during the initialization process – and more specifically, I know that this is a call to the `Initialize` method.

The last part is the expected outcome. For the first failed test, `IsPopulated` tells me that I expect the `Model` property to be populated after the `Initialize` method is run.

So, we're expecting the `Model` property to be populated but it isn't. If we review the code, it looks like we missed a step. We modified the detail method so that it returns the Current Order, but we never assign that to the `_model` field.

This is a quick fix:

```
public void Initialize()
{
    _service = GetServiceFromContainer();
    _model = GetCurrentOrderFromContainer();
    RefreshCatalog();
}

private IPersonService GetServiceFromContainer()...

private CatalogOrder GetCurrentOrderFromContainer()
{
    if (!_container.IsRegistered<CatalogOrder>("CurrentOrder"))
        throw new MissingFieldException(
            "CurrentOrder is not available from the DI Container");
    return _container.Resolve<CatalogOrder>("CurrentOrder");
}
```

Build and run tests: All tests pass. Now we're back to normal.

Now, you might be wondering why we had 6 tests fail instead of just one. The failing test that we looked at (`Model_OnInitialized_IsPopulated`) happens to be the test for the piece of code that we broke. The other 5 tests failed with `NullReferenceException`s because the tests themselves were trying to use the `Model` property (that wasn't being set).

Another side note about testing: in our `Initialize` method, we are assigning to the private `_model` field, but in the unit tests, we are testing the public `Model` property. In my unit tests, I like to test the public members as much as possible. I'm testing the public members because I know that the private implementation is subject to change, and I don't want to constantly change my unit tests when the private parts change.

What I'm really testing is if my class behaves the way that the outside world expects. The outside world can only see the public members – these are the properties that the View uses for data binding as well as the public methods that can be called through button clicks.

There are times when we might want to test private members. But as a general rule, I like to stick with testing the public API.

So, we've seen how the hierarchical refactoring can make our methods easier to understand. Imagine walking up to this for the first time:

```
public void Initialize()
{
    _service = GetServiceFromContainer();
    _model = GetCurrentOrderFromContainer();
    RefreshCatalog();
}
```

At a glance (and without someone needing to explain it to me), we have a good idea of what this method is trying to do. It is assigning the `_service` field (from the DI container), assigning the `_model` field (also from the DI container), and then it is refreshing the Catalog.

If we need details, we can always drill down into the other methods. But we have a very clear roadmap of where we might need to go.

Refactoring the RefreshCatalog Method

We'll use these same techniques to improve the readability of the `RefreshCatalog` method. Let's see if we can find some places to chunk this out:

```
public void RefreshCatalog()
{
    if (DateTime.Now - LastUpdateTime < new TimeSpan(0, 0, 10))
    {
        _include70s = _include80s = _include90s = _include00s = true;
        RefreshFilter();
    }
    else
    {
        Catalog = new List<Person>();

        var asyncBegin = _service.BeginGetPeople(null, null);
        var task = Task<List<Person>>.Factory.FromAsync(
            asyncBegin, _service.EndGetPeople);
        task.ContinueWith(t =>
            {
                _fullPeopleList = t.Result;
                _include70s = _include80s = _include90s = _include00s
                    = true;
                RefreshFilter();
                LastUpdateTime = DateTime.Now;
            }, TaskContinuationOptions.NotOnFaulted);

        Action<Exception> rtxDel = (ex) => { throw ex; };
        var uiDispatcher = Dispatcher.CurrentDispatcher;
        task.ContinueWith(rp =>
        {
            uiDispatcher.Invoke(rtxDel, rp.Exception.InnerException);
        }, TaskContinuationOptions.OnlyOnFaulted);
    }
}
```

Here's the big functionality:

1. The "if" statement determines whether the cache is still valid (by comparing the current time to the Last Updated time and seeing if the difference is more than 10 minutes).
    a. If the cache is valid, then we reset the filters.

2. If the cache is not valid:
    a. We clear the "Catalog" property. This has the effect of blanking out the primary list on the View.
    b. We make an asynchronous call to the service. The asynchronous call makes sure that the UI stays responsive while the data is being retrieved.
    c. When the asynchronous call returns:
        i. We set an internal field (`_fullPeopleList`) with the result of the service. (The internal field is used in the filtering process.)
        ii. We reset the filters.
        iii. We set the `LastUpdated` property.
    d. If the asynchronous call errors, re-throw the exception on the UI thread.

A quick note about the asynchronous service call. We aren't using `async`/`await` here because the code is .NET 4.0 (a limitation because there were still Windows XP machines that needed to run this code).

Step 1: Remove Duplicate Code

Let's start with the duplicated code (remember the DRY principle). Notice that we have "reset the filters" in 2 places. Here's the code:

```
_include70s = _include80s = _include90s = _include00s = true;
RefreshFilter();
```

Since duplicate code is a big red flag, let's refactor this out. Highlight the 2 lines inside the first `if` statement and extract a method (again using the right-click menu or Ctrl+R, Ctrl+M). We'll name the method `ResetFilterToDefaults`. After the method is created, replace the duplicated code with the new method.

Here's what we get:

```
public void RefreshCatalog()
{
    if (DateTime.Now - LastUpdateTime < new TimeSpan(0, 0, 10))
    {
        ResetFilterToDefaults();
    }
    else
    {
        Catalog = new List<Person>();

        var asyncBegin = _service.BeginGetPeople(null, null);
        var task = Task<List<Person>>.Factory.FromAsync(
            asyncBegin, _service.EndGetPeople);
        task.ContinueWith(t =>
            {
                _fullPeopleList = t.Result;
                ResetFilterToDefaults();
                LastUpdateTime = DateTime.Now;
            }, TaskContinuationOptions.NotOnFaulted);
```

```
                Action<Exception> rtxDel = (ex) => { throw ex; };
                var uiDispatcher = Dispatcher.CurrentDispatcher;
                task.ContinueWith(rp =>
                {
                    uiDispatcher.Invoke(rtxDel, rp.Exception.InnerException);
                }, TaskContinuationOptions.OnlyOnFaulted);
            }
        }

        private void ResetFilterToDefaults()
        {
            _include70s = _include80s = _include90s = _include00s = true;
            RefreshFilter();
        }
```

Build and run tests: All tests pass. You can experiment by commenting out the call to `ResetFilterToDefaults` in one place or the other and see that the tests do catch the problem.

Step 2: Clarify the Conditional

For the next part, I want to make the `if` statement more readable.

```
            if (DateTime.Now - LastUpdateTime < new TimeSpan(0, 0, 10))
```

If I'm new to this code, I can figure out what this is doing (checking to see if the `LastUpdateTime` is more than 10 seconds in the past), but it doesn't really tell me why we're checking this. If we extract this as a method or property, we can make the intent clear.

Let's create a calculated property with this code:

```
    public bool IsCacheValid
    {
        get
        {
            return DateTime.Now - LastUpdateTime < new TimeSpan(0, 0, 10);
        }
    }

    public void RefreshCatalog()
    {
        if (IsCacheValid)
        {
            ResetFilterToDefaults();
        }
        else ...
    }
```

Unfortunately, Visual Studio doesn't have any built-in refactoring that makes this easy. I usually end up doing some copy/paste work along with some code snippets to create the property. A third-party tool will probably have some better options.

As an organizational note: I moved the `IsCacheValid` property up to the `Properties` region of the class.

Build and run tests: All tests pass. And now our `if` statement shows the intent: we want to know if the cache is still valid.

Step 3: Extract the Bulk

Now let's move on to the `else` statement, and let's think of this `RefreshCatalog` method from a high level. If the cache is valid, then we reset the filters. If the cache is not valid, we repopulate the catalog from the service. So, let's go ahead and make our code say that.

Highlight everything inside the `else` statement, extract the method, and name it `PopulateCatalogFromService`.

```
        public void RefreshCatalog()
        {
            if (IsCacheValid)
            {
                ResetFilterToDefaults();
            }
            else
            {
                PopulateCatalogFromService();
            }
        }

        private void PopulateCatalogFromService()
        {
            Catalog = new List<Person>();

            var asyncBegin = _service.BeginGetPeople(null, null);
            var task = Task<List<Person>>.Factory.FromAsync(
                asyncBegin, _service.EndGetPeople);
            task.ContinueWith(t =>
            {
                _fullPeopleList = t.Result;
                ResetFilterToDefaults();
                LastUpdateTime = DateTime.Now;
            }, TaskContinuationOptions.NotOnFaulted);

            Action<Exception> rtxDel = (ex) => { throw ex; };
            var uiDispatcher = Dispatcher.CurrentDispatcher;
            task.ContinueWith(rp =>
            {
                uiDispatcher.Invoke(rtxDel, rp.Exception.InnerException);
            }, TaskContinuationOptions.OnlyOnFaulted);
        }
```

Build and run tests: All tests pass.

Now if we remove some of the curly braces from the `if`/`else`, we end up with a very compact, readable method:

```
public void RefreshCatalog()
{
    if (IsCacheValid)
        ResetFilterToDefaults();
    else
        PopulateCatalogFromService();
}
```

And as always, we can drill down if we need to. If we need to see how long the cache is, we can check the `IsCacheValid` property. If we need details on populating the catalog from the service, we just drill into that method.

Note: I won't get into the debate of whether the curly braces should be left in. There are different schools of thought. Some people always leave them in just in case another statement is added. Personally, if both the "if" and "else" have only a single, short statement, I'll pull them out.

Step 4: Extracting Error Handling

When we look at the `PopulateCatalogFromService` method that we just created, we see that the last part is for error handling.

```
    Action<Exception> rtxDel = (ex) => { throw ex; };
    var uiDispatcher = Dispatcher.CurrentDispatcher;
    task.ContinueWith(rp =>
    {
        uiDispatcher.Invoke(rtxDel, rp.Exception.InnerException);
    }, TaskContinuationOptions.OnlyOnFaulted);
```

Since that is "exceptional" behavior, let's move it out of this method into a lower-level method. We'll extract it into `RethrowServiceExceptionOnUIThread`. Sometimes method names may need to be long in order to tell what they really do. We don't want our method names to be overwhelming, but there are times that we can use some extra information.

One thing to note about the Extract Method refactoring in Visual Studio: When we extract these 6 lines of code, Visual Studio sees that we need a parameter for our new method (which is a `Task`). The refactoring tool automatically adds this for us. Here are the results:

```
private void PopulateCatalogFromService()
{
    Catalog = new List<Person>();

    var asyncBegin = _service.BeginGetPeople(null, null);
    var task = Task<List<Person>>.Factory.FromAsync(
        asyncBegin, _service.EndGetPeople);
    task.ContinueWith(t =>
    {
        _fullPeopleList = t.Result;
        ResetFilterToDefaults();
        LastUpdateTime = DateTime.Now;
    }, TaskContinuationOptions.NotOnFaulted);

    RethrowServiceExceptionOnUIThread(task);
}
```

```
        private static void RethrowServiceExceptionOnUIThread(
            Task<List<Person>> task)
        {
            Action<Exception> rtxDel = (ex) => { throw ex; };
            var uiDispatcher = Dispatcher.CurrentDispatcher;
            task.ContinueWith(rp =>
            {
                uiDispatcher.Invoke(rtxDel, rp.Exception.InnerException);
            }, TaskContinuationOptions.OnlyOnFaulted);
        }
```

Build and run tests: Everything's still good.

Step 5: Some Useful Comments

We could extract some more code out of the `PopulateCatalogFromService` method, but I'm satisfied with the level of detail that we have here. But there is one line of code that could use some explanation:

```
            Catalog = new List<Person>();
```

Why are we doing this? I gave a bit of a hint in the functionality outline. During the process, we make an asynchronous call to the service. The service itself has an artificial 1 second delay built in (to simulate a normal web service call). Since we've clicked the "Refresh Catalog" button on the UI, we want to see some UI changes to know that it's working. So, we clear the primary list box and keep it empty until the data comes back from the service.

We do this through databinding by clearing the `Catalog` property in the ViewModel. When `Catalog` is cleared (by assigning a blank list), the list box in the UI is notified and updates its binding – this clears the list box. Later on, when the `Catalog` property is repopulated, the list box will show data (we'll look at where `Catalog` gets populated in just a bit).

So, let's add a comment to clarify the intent of the code:

```
        private void PopulateCatalogFromService()
        {
            // Clearing the Catalog will clear the screen
            // while we wait for the async service call.
            Catalog = new List<Person>();

            var asyncBegin = _service.BeginGetPeople(null, null);
            var task = Task<List<Person>>.Factory.FromAsync(
                asyncBegin, _service.EndGetPeople);
            task.ContinueWith(t =>
            {
                _fullPeopleList = t.Result;
                ResetFilterToDefaults();
                LastUpdateTime = DateTime.Now;
            }, TaskContinuationOptions.NotOnFaulted);

            RethrowServiceExceptionOnUIThread(task);
        }
```

Step 6: How Does Catalog Get Set?

Let's look at what happens when our asynchronous service call returns:

```
                _fullPeopleList = t.Result;
                ResetFilterToDefaults();
                LastUpdateTime = DateTime.Now;
```

Something's not quite clear. Ultimately, we need to set the `Catalog` property in order for it to be updated. But we're not doing that here. Let's take a look at what happens in `ResetFilterToDefaults`:

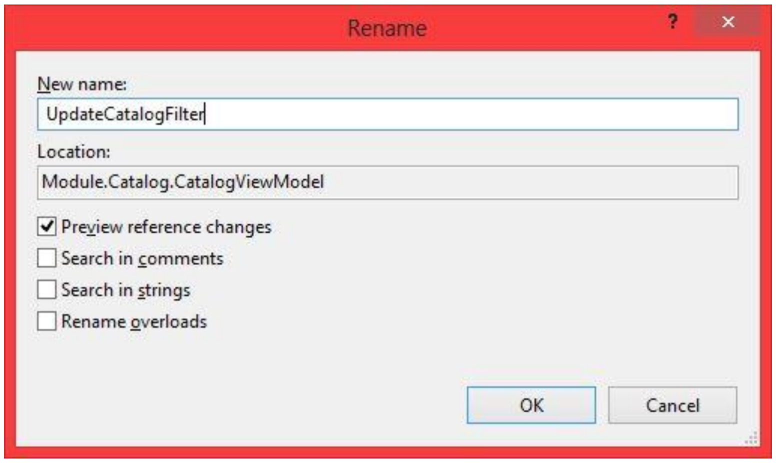
```
        private void ResetFilterToDefaults()
        {
            _include70s = _include80s = _include90s = _include00s = true;
            RefreshFilter();
        }
```

Nope, nothing here. Let's drill down to `RefreshFilter`:

```
        private void RefreshFilter()
        {
            RaisePropertyChanged("Include70s");
            RaisePropertyChanged("Include80s");
            RaisePropertyChanged("Include90s");
            RaisePropertyChanged("Include00s");

            IEnumerable<Person> people = _fullPeopleList;
            if (!Include70s)
                people = people.Where(p => p.StartDate.Year / 10 != 197);
            if (!Include80s)
                people = people.Where(p => p.StartDate.Year / 10 != 198);
            if (!Include90s)
                people = people.Where(p => p.StartDate.Year / 10 != 199);
            if (!Include00s)
                people = people.Where(p => p.StartDate.Year / 10 != 200);

            Catalog = people.ToList();
        }
```

There it is. We're setting the `Catalog` property when we refresh the filter. That gives me a hint that this method might not have the best name, so let's give it a new one.

Right-click on `RefreshFilter`, select "Refactor", then "Rename" (or press Ctrl+R, Ctrl+R). Let's give it a new name of `UpdateCatalogFilter`.

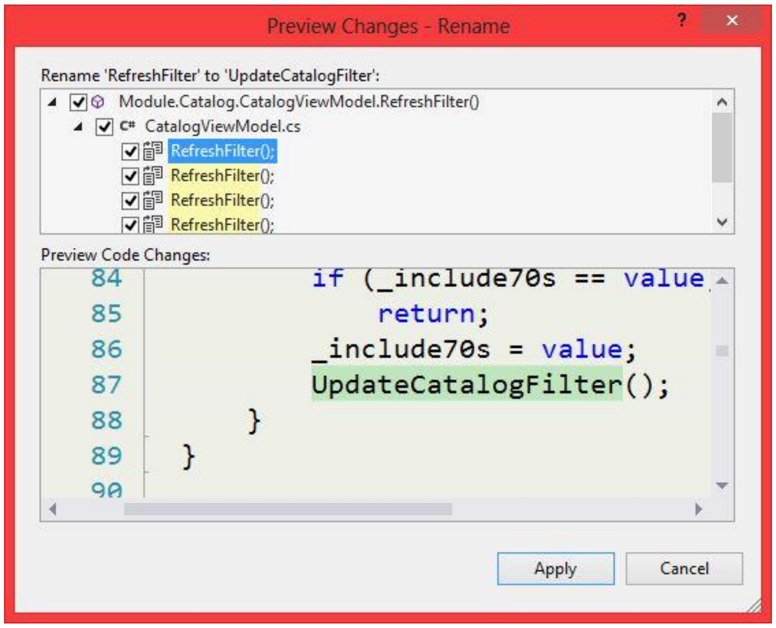

Rename

New name:
UpdateCatalogFilter

Location:
Module.Catalog.CatalogViewModel

☑ Preview reference changes
☐ Search in comments
☐ Search in strings
☐ Rename overloads

Notice the options that we have: we can update strings and comments in addition to any references. Let's leave "Preview reference changes" checked and click "OK".


Preview Changes - Rename

Rename 'RefreshFilter' to 'UpdateCatalogFilter':

☑ Module.Catalog.CatalogViewModel.RefreshFilter()
☑ CatalogViewModel.cs
☑ RefreshFilter();
☑ RefreshFilter();
☑ RefreshFilter();
☑ RefreshFilter();

Preview Code Changes:

```
                if (_include70s == value
                    return;
                _include70s = value;
                UpdateCatalogFilter();
            }
        }
```

This shows us all the places where `RefreshFilter` will be updated. We can click each item and see a preview of the updated code. We can also uncheck items selectively if we find something we don't actually want to rename (although I've never actually done that).

Clicking "OK" will complete the renaming.

Build and run tests: Still good.

Note: We could continue the renaming up a level to make `ResetFilterToDefaults` more descriptive, but we won't undertake that now.

One Last Change

Now that I've had a chance to look at this `UpdateCatalogFilter` method, I think we can improve the readability:

```
private void UpdateCatalogFilter()
{
    RaisePropertyChanged("Include70s");
    RaisePropertyChanged("Include80s");
    RaisePropertyChanged("Include90s");
    RaisePropertyChanged("Include00s");

    IEnumerable<Person> people = _fullPeopleList;
    if (!Include70s)
        people = people.Where(p => p.StartDate.Year / 10 != 197);
    if (!Include80s)
        people = people.Where(p => p.StartDate.Year / 10 != 198);
    if (!Include90s)
        people = people.Where(p => p.StartDate.Year / 10 != 199);
    if (!Include00s)
        people = people.Where(p => p.StartDate.Year / 10 != 200);

    Catalog = people.ToList();
}
```

This uses LINQ to perform the actual filtering of the lists. The `if` statements are all negatives because the filters are additive. There are a couple different approaches to this, but this seems to be the cleanest. The part that I want to fix is what we have in the lambda expressions of the `Where` statement:

```
p => p.StartDate.Year / 10 != 197
```

The intent of this is to figure out whether the `StartDate` property is in a particular decade (the 1970s for this particular one). We take the `Year` (such as 1975) and use integer division to divide it by 10. This truncates the last digit (ex., 197). Then we use that for comparison. But comparing a date to 197 doesn't really make sense. What if we multiply everything by 10?

```
if (!Include70s)
    people = people.Where(p => p.StartDate.Year / 10 * 10 != 1970);
if (!Include80s)
    people = people.Where(p => p.StartDate.Year / 10 * 10 != 1980);
if (!Include90s)
    people = people.Where(p => p.StartDate.Year / 10 * 10 != 1990);
if (!Include00s)
    people = people.Where(p => p.StartDate.Year / 10 * 10 != 2000);
```

Honestly, I'm not sure if that's better or worse. I like that I'm comparing to an actual decade (1970), but taking the year and dividing by 10 and then multiplying by 10 – that's ugly. And I can see someone wanting to optimize it.

So, let's hide the ugliness behind a calculated property. This will take us to the `Person` class (in the `Person.cs` file). The easiest way to navigate there is to right-click on `StartDate` and select "GoToDefinition" (or press F12).

Let's create a new calculated property called `StartDecade`:

```
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime StartDate { get; set; }
        public int Rating { get; set; }

        public int StartDecade
        {
            get { return StartDate.Year / 10 * 10; }
        }

        public override string ToString()
        {
            return string.Format("{0} {1}", FirstName, LastName);
        }
    }
```

When we update our `UpdateCatalogFilter` method, the intent is much clearer:

```
        private void UpdateCatalogFilter()
        {
            RaisePropertyChanged("Include70s");
            RaisePropertyChanged("Include80s");
            RaisePropertyChanged("Include90s");
            RaisePropertyChanged("Include00s");

            IEnumerable<Person> people = _fullPeopleList;
            if (!Include70s)
                people = people.Where(p => p.StartDecade != 1970);
            if (!Include80s)
                people = people.Where(p => p.StartDecade != 1980);
            if (!Include90s)
                people = people.Where(p => p.StartDecade != 1990);
            if (!Include00s)
                people = people.Where(p => p.StartDecade != 2000);

            Catalog = people.ToList();
        }
```

File Organization, Part 2

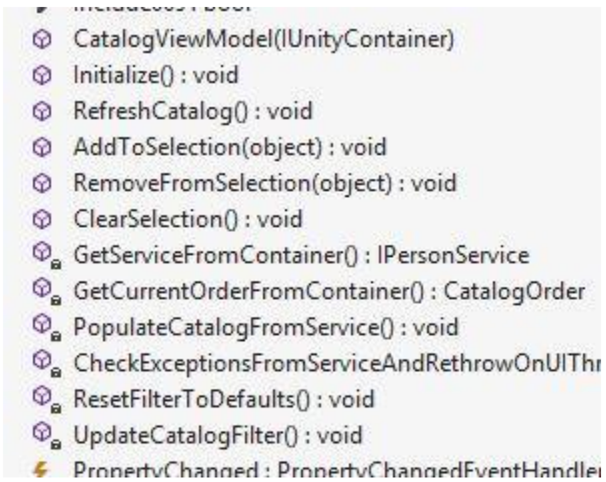
As noted when we first cracked open these files, I have regions set up for the various pieces of the class. I'll emphasize that this organization is a personal preference based on a coding standard that I worked with for quite a while.

Because of this, I like to separate my public methods from my private methods. So, I'll create a separate regions for them.

Here is the organization of those regions (with some white space removed):

```
#region Public Methods

public void Initialize()...
public void RefreshCatalog()...
public void AddToSelection(object person)...
public void RemoveFromSelection(object person)...
public void ClearSelection()...

#endregion

#region Private Methods

private IPersonService GetServiceFromContainer()...
private CatalogOrder GetCurrentOrderFromContainer()...
private void PopulateCatalogFromService()...
private static void CheckExceptionsFromServiceAndRethrowOnUIThread(
    Task<List<Person>> task)...
private void ResetFilterToDefaults()...
private void UpdateCatalogFilter()...

#endregion
```

I like to keep my public API as visible as possible. This is why I like to have the `Properties`, `Constructors`, and `Public Methods` regions. Also, when we look at the class view in the Visual Studio 2012 enhanced Solution Explorer, the members appear in the order they appear in the code (not alphabetically):



CatalogViewModel(IUnityContainer)
Initialize() : void
RefreshCatalog() : void
AddToSelection(object) : void
RemoveFromSelection(object) : void
ClearSelection() : void
GetServiceFromContainer() : IPersonService
GetCurrentOrderFromContainer() : CatalogOrder
PopulateCatalogFromService() : void
CheckExceptionsFromServiceAndRethrowOnUIThr
ResetFilterToDefaults() : void
UpdateCatalogFilter() : void
PropertyChanged : PropertyChangedEventHandle

I like that the public methods (the cube icons) are all grouped together and the private methods (the cubes with padlocks) are all grouped together.

Again, just a personal preference (but there is some logic behind it).

Be sure to build and run the tests again. We want to ensure that we didn't break anything.

Refactoring Review

So, we just spent 20 pages refactoring code. Was it worth it? You be the judge. Here are the high-level methods we ended up with:

```
public void Initialize()
{
    _service = GetServiceFromContainer();
    _model = GetCurrentOrderFromContainer();
    RefreshCatalog();
}

public void RefreshCatalog()
{
    if (IsCacheValid)
        ResetFilterToDefaults();
    else
        PopulateCatalogFromService();
}
```

I don't think we have anything to fear from the homicidal maniac.

Our `CatalogViewModel.cs` file is now 275 lines long (compared to the 243 lines we started with). And remember, we didn't really add new code – we primarily just wrapped existing code in methods that describe the functionality.

And all of our unit tests still run, so we know that we haven't altered the functionality. If we run the application, we see that it works exactly as it did before. But we already knew that since we have good unit test coverage.

There may be some folks who say we didn't go far enough. Perhaps we should split out some of the functionality into separate classes (such as the service calls). Perhaps we need a few helper classes. Perhaps we need to parameterize the time for our cache. Code is never "perfect". And that's okay.

We do have to be careful, though. We could spend all day refactoring. Often it is necessary, but eventually, it gets to the point where it's not adding much benefit for the amount of effort involved. Like everything else in development, we need to keep things in balance.

Be a Clean Code Advocate

The Boy Scout rule is "Always leave the campground cleaner than you found it." If we come across a mess, we should take responsibility to clean it up, even though we didn't make that mess in the first place.

We can bring this rule into our world of clean code:

Always leave the code cleaner than you found it.

Whenever we need to go into a piece of code to make a change (whether it's a bug fix or a feature update), we should clean up the section of code that we're working on. This may mean renaming ambiguous properties or methods. It may mean adding unit tests that weren't there before. It may mean breaking complex methods into a high- / mid- / detail-level hierarchy.

Think about how you wish you had found things when you opened the code. Now, try to leave it that way for the next developer. (And sometimes that next developer is you – six months down the road.)

And if you need more encouragement, just think about that homicidal maniac: he's got your home address…

Happy coding!