DOUBLE DUTY

How to repurpose the unit tests you're doing to help create the documentation you're not.

BY BRIAN BUTTON

THE PROBLEM: We're spending a large percentage of our development budget on writing documentation. For instance, I'm working on a project to create a reusable set of infrastructure components that are intended to be used by .Net programmers as a starting point for their own business applications. Our project output is code, so documentation is especially important to us, but still we're spending quite a lot of our time and budget writing about what we're doing rather than actually doing it. Something just seems out of balance.

A big part of this cost is the early and continued maintenance of keeping the needed API documentation in sync with code changes. We're an Agile team, developing software using Test Driven Development (TDD). The advantage to developing code in this manner is that it allows us to change and evolve the software as the requirements change, and change they do! But as the code changes, the API documentation needs to change as well, and this has been hard to make happen at the same rate as our code is changing.

OUR SOLUTION? To solve this problem, we are leveraging the unit tests that we use to design our system and make changes safe. We are changing the way we write our tests so that they serve as API documentation. Now the process of creating our code through TDD is the process of creating the documentation, and evolving the code also evolves the documentation. We call this Agile Documentation.

Documentation Pet Peeves

There are two things that have always really bothered me about traditional documentation. This whining reflects seventeen years of professional experience, and seventeen years of realizing that fundamentally nothing has changed to make things better. Starting with Unix manual pages, into javadoc, and finally into MSDN documentation, the issues that existed in the late eighties into the nineties still exist today.

My first pet peeve is that the documentation I'm reading seldom answers the question I'm asking. I always seem to have this uncontrollable urge to junk the documentation and just go read the source. The source is the only unambiguous definition of what the system does, and all answers are available there. Unfortunately, reading the source is becoming less and less possible in this age of proprietary software. You're left with trying to read between the lines of what documentation you *can* find, and then either Googling for examples or writing your own.

My second source of irritation stems from the static nature of the documentation for the code I'm writing. As our industry has learned more about writing code, and as our tools for writing that code have gotten infinitely better (Eclipse and IntelliJ for Java, ReSharper for .Net), it has become easier and easier to modify our code. We can change the names of methods, add and remove parameters from these methods, split classes into two, extract interfaces, etc., all of which change our public API. We do this because it makes our code easier to read, modify, digest, and understand. But the documentation for these methods is not changed at the same time by the same tools. Changing documentation is always a separate, manual pass that is (maybe) done later. And the cost of doing this post-coding documentation step is frequently a drag on development. There have been many times during my career that my team has really, really wanted to make a change in the software—a change that is clearly dictated for business or software quality reasons but that is disallowed because of the documentation costs associated with that change. That just seems backward to me.

Join Brian Button on StickyMinds.com 7 February through 18 March for a facilitated RoundTable discussion on tests as documentation. Ask questions, share stories, and get more details by logging on to www.stickyminds.com/roundtable.asp.

What I'd really like to see, and what Agile Documentation will do, is something that is going to address both of these problems. Text-based documentation doesn't answer my detailed questions, but examples do. The way we are writing our unit tests makes them great examples of everything that our system is designed to do. And these examples are forced to change along with the code, which solves

UNIT TESTS DEFINED

In this context, unit tests are specific, targeted probes into the behavior of one particular path through the application code. They set up some aspect of the system, perform some action on it, and assert something about the reaction of the system. Unit tests are not performance tests, scalability tests, security tests, or other system-level kinds of tests. They are focused on a single module and its relationship with the system at large.

ANATOMY OF A UNIT TEST

The figure at right shows a typical unit test. This test documents one behavior of the System.String class that is part of the Microsoft .Net Framework. In it, we see that the behavior we are documenting converts the contents of a string object to uppercase and results in the framework creating a new string instance. How do we know that? We know because we read the name of the test method. It is a simple, clear, assertive statement of the behavior being tested. By naming tests like this, we help people find the right test to read and help them understand what the test is for.

The code in the test is very simple and declarative. It breaks down into three separate parts, originally described by Bill Wake as the three "A's" of unit tests. The first part is where we *arrange* the environment so that we can test. In this case, we're defining a string object containing a well-known piece of data. The second piece is where we *act* on the system to force it to change state. Here we are calling a method on the original string object to cause it to return an uppercase version of the original string's contents. And finally, we *assert* something about the behavior of the system, in this case that the original string is unchanged after the method call in the act section, and that a new string was created with an uppercase version of my first name.

Admittedly, this unit test is testing a very simple concept, but that's part of what can make a unit test effective as documentation. As a test writer, you have to focus on writing tests that are simple and easily read and understood. Without this property, the tests are no more useful as documentation than the application code.

```
public void ConvertingStringToUpperCaseCreatesNewStringInstance()
{
    // Arrange
    string lowerCaseString = "brian";

    // Act
    string upperCaseString = lowerCaseString.ToUpper();

    // Assert
    Assert.AreEqual("brian", lowerCaseString);
    Assert.AreEqual("BRIAN", upperCaseString);
}
```

Customer Tests as Documentation

PERSPECTIVE

Using tests as documentation is nothing new for Jennitta Andrea, partner and senior consultant at clearStream. She has years of experience creating customer tests that both validate system functionality and act as a core piece of documentation that provides the entire team with a big picture of system workflow and the overall business process description. Although Jennitta uses Agile strategies to set up the customer testing that feeds into documentation, she says one size does not fit all projects.

"There really isn't one standard way to do this. There isn't one tool or framework that will fit every situation," she says. "Luckily we have a lot of different types of tools to choose from, because the best strategy and tool depends on the overall context and the level of experience of the people on the project."

And while she believes that ideally the customer should write the tests, Jennitta has found that this rarely happens.

"There are some tools that are geared toward the customer writing the test himself, but I've been involved in many situations where the customer is only involved in reviewing the tests or where he really just wants to validate the results, to make sure that when the tests run, the results are correct and the system truly does what it is supposed to do," she explains.

Even with all the tools available, Jennitta warns that it's not easy to make tests that can be used effectively as documentation for system requirements. In fact, tools can sometimes get in the way.

"It takes a lot of experience to write effective customer tests, even with tools like FIT, which presents the content in a friendly way for the end-user, or Canoo WebTest, which uses a higher-level language for user interface testing," she says. "You can easily get caught in the trap of simply expressing your tests in terms of user interface interaction—'push this button,' 'enter text in this field.'"

And that, she contends, is much too low level to be effective, because the business meaning contained in the test is obscured.To counteract this, she recommends building the tests on a domain-specific testing language.

"A domain-specific testing language provides a vocabulary for expressing tests in terms of user interaction goals with the system. For example—a billing system. Instead of specifying the tests with low-level statements like 'click here' 'enter there,' the test says 'generate charges.' To support the automated execution of these tests, we often need to extend or enhance the tools we use to translate this specific testing language into a sequence of lower-level concepts. Now customer tests can be expressed in business terminology and still be used to automatically verify system behavior," Jennitta explains. This makes the test more viable as documentation.

According to Jennitta, another key component in using tests as documentation is to focus on the seemingly simple things such as naming and organizing your tests. As easy as organizing the tests sounds, Jennitta has found that most people don't think of it.

"Because tests are very specific and very detailed in how they are expressed, and . . . [because] they can be executed, the expectation is this is the best form of requirements you can be given."

Yet, Jennitta says, "Depending on the project, you could end up with literally hundreds of tests. If these tests are your only form of requirements and if you can't find all of the requirements related to a particular functional area because the tests aren't grouped or named very well, then you don't really have the big picture of what your system requirements are."

On a recent project, this need for big picture insight led Jennitta's team to the wiki, albeit late in the project.

"We hadn't used a wiki before, so it was fairly new to us," she recalls. "We began using it just as a way to put everything together for the support team. When we discovered it was easy to use and how much potential it had, we realized it should have been used at the beginning to help the development team as well."

While Jennitta specializes in Agile methods, she says she doesn't think it's off course to have some amount of "real" textual documentation.

"For all but the simplest system, I think it is still necessary to combine customer tests with some kind of traditional documentation," she explains. "Use diagrams and text to provide the necessary background for the system at an appropriate level for the team. Providing links to the customer tests within this content provides the contextual grouping that is necessary. If possible, tie everything together within a wiki. This is an easy way to combine . . . the textual background with the list of tests that correspond to that area."

my second issue. Agile Documentation could be the first thing in my seventeen years in the industry that fundamentally addresses the two big documentation shortcomings I've seen since day one.

Agile Documentation

If you are already writing unit tests, you are a good part of the way toward creating Agile Documentation. We just have to tweak a few things that you are doing to make it easier to find the appropriate unit tests and make those unit tests as understandable as humanly possible for as many different readers as possible.

Traditional documentation for a system provides both an overview of that system and specific, individual usage details for methods and classes. Unit tests, as presented here, do a good job of describing the usage details for pieces of our system at a highly granular level. However, we can expand their scope of description a bit if we organize and write them in such a way that they tell a story. They won't replace the overview aspects of traditional documentation, but they will do a better job of explaining the details of a system.

We do this by creating a Test List. A Test List is a narrative about what the class or subsystem under test is supposed to do. It starts out by specifying basic be-

TEST LIST

- Notifies no one if no one is listening when file changes
- Notifies sole registered listener when underlying file changes
- Notifies all registered listeners when underlying file changes
- Data about changed file is included in callback argument when callbacks occur
- Removing listener prevents that listener from being called back
- No callback will happen after last listener is removed
- Deleting watched file does not cause any callbacks
- Creating watched file after we start watching it will not cause any callbacks
- Creating and changing watched file after we start watching it will cause one callback
- Overwriting existing watched file causes exactly one callback
- Notifications do not accumulate while callbacks are happening
- Disposing of file watcher stops callbacks from happening
- Exception thrown in callback method stops other listeners from being called back

Figure 1: My initial Test List for this class is not complete. And that's not important. What *is* important is that it is completed by the time the class is implemented.

haviors and grows toward more complex situations, including telling stories about error and exception handling. Some of the behaviors being documented will be immediately understandable to anyone looking at the class for the first time, and some will be so advanced that only highly experienced programmers who understand the class intimately will be able to comprehend them. This is OK because we have different kinds of users for our classes. The basic examples will help those using our class for the first time start to get that comprehensive understanding and will demonstrate how to use a feature quickly and explicitly. Over time, as the users' needs from any class grow, the users will find that some of the entries in the Test List will begin to intrigue them, and then they can start using that aspect of that class.

Building a Test List

So let's look at building a Test List. As an example, let's take a class that I implemented recently. This class has the responsibility of raising a notification event whenever a particular configuration file changes. This notification is sent to all objects that have previously registered with this object to be told about these changes. That's the entire problem description.

So where do we start with our first test? I usually start with the most basic state of the system being created: its behavior right after being initialized. This is a simple case to get started with, and, no matter what else you implement, this behavior still has to work. So, the first test in my list would be to document that I can create this class, which I called a ConfigurationChangeFileWatcher, and that the class will do nothing if the file it is watching changes but no one has registered interest in hearing about that change.

Test 1: Notifies no one if no one is listening when file changes

Please notice something about this test description. It reads like a requirement statement of the class being written. It is also entirely concerned with the intent of the class being developed and not concerned with the implementation of the test. There is nothing in the description that describes the implementation steps inside the test, and that's how it should be. Written from the point of view of a user of the class, this description documents the class's externally visible behavior, just like a requirement would.

Our next test should add something valuable to our first test description. The next logical thing to describe would be its behavior after a single interested party has registered as a listener and the file changes. This test would illustrate the most basic feature of the ConfigurationChangeFileWatcher: It will notify a listener when the underlying file changes.

Test 2: Notifies sole registered listener when underlying file changes

The next step could be to document that all listeners are notified when multiple listeners are registered.

Test 3: Notifies all registered listeners when underlying file changes

After this test, we have the basic operation of the class defined to the point that a user of the class could read this list of tests and get an idea of how our system works. He still wouldn't know how to use the ConfigurationChangeFileWatcher in code yet, as we haven't shown any of our examples, but that will come soon enough.

The best part of documenting a class's behavior like this comes in when you start talking about those odd things that can happen during an object's life. What do you suppose the class does if there are many listeners registered to be called back if the configuration file changes, and one of those callback methods throws an exception for some reason? Are the rest of the listeners called back anyway, or do the callbacks stop at that point? I defy you to find even a mention of a detail like this in any written manual for any class. On the other hand, when implementing the class, I have to write some kind of code to handle this situation, so I have to have a unit test for it, which means that a description of this test ends up on my Test List.

Test N: Exception thrown in callback method stops other listeners from being called back

Now that's a pretty specific scenario to be documented, but that's exactly the kind of thing I would previously have had to read the code to find out. Now it is documented for me, explicitly, and I know exactly how the code behaves.

Figure 1 shows my initial Test List for the ConfigurationChangeFileWatcher. It

```
[Test]
public void NotifiesSoleListenerIfFileChanges() {
    string fileToWatch = "MyFile.Test";
    string matchingConfigSection = "MySection";

    using (ConfigurationChangeFileWatcher watcher =
        new ConfigurationChangeFileWatcher(fileToWatch, matchingConfigSection))
    {
        watcher.ConfigurationChanged +=
            new ConfigurationChangedEventHandler(FileChangedCallback);
        watcher.StartWatching();
        Thread.Sleep(100);

        File.SetLastWriteTime(fileToWatch, DateTime.Now + TimeSpan.FromHours(1.0));
        Thread.Sleep(250);
    }

    Assert.AreEqual(1, numberOfConfigurationNotifications);
}
```

Figure 2: Start implementing the tests, one by one, in a traditional TDD manner.

is helpful to note a few different things about this list. First of all, it is not complete. And that isn't important. What is important is that it is complete by the time that class is implemented. When I first implement a class, I'll put down all the behaviors I can think of, but I always think of more situations to try, and that leads me to create new test descriptions, which turn into new tests. The other thing to understand is that each test statement may not make sense initially to the developer using this class. But as the developer gets more experience using the ConfigurationChangeFileWatcher and as different situations arise—such as the one about exceptions and callbacks—additional, more advanced test descriptions will begin to make sense and he'll know where to look to understand those behaviors. A developer's understanding of the Test List for a class grows with his understanding of that class.

Implementing Unit Tests

Once I have created my initial pass at the Test List, I turn that into a test fixture and start implementing the tests, one by one, in a traditional TDD manner, following the same pattern as described in the sidebar "Unit Tests Defined." (See Figure 2.) The biggest difference in writing tests for documentation rather than solely for TDD purposes is that they have to be made extremely understandable on their own. This means that the code must be kept simple, great names must be used for variables and methods called in the test, and the purpose of the test must be obvious.

There are many interesting things documented by this single unit test that can be understood by a knowledgeable .Net programmer seeking to use this class. Much of this is admittedly context dependent, but users of your class should already understand its context, so this shouldn't be much of a problem. Casual readers who don't understand the context may have more trouble understanding some of the unit tests, but this is a situation quickly addressed by experience.

EACH TEST STATEMENT MAY NOT MAKE SENSE INITIALLY TO THE DEVELOPER USING THIS CLASS.

An interesting way to read a unit test is to start backward. We see that the intent of this test is to declare that a single listener is notified when the watched file changes. This is documented through the assertion, *Assert.AreEqual*, at the end of the test. The part of the test in the middle, starting with the *using* statement, declares several interesting things. The first is that it shows a user how to create an instance of this class. The constructor of this class takes the name of the file to watch along with the configuration section that is represented by this file. We also see a very important piece of information to a .Net programmer. The *using* statement documents the fact that ConfigurationChangeFileWatcher implements a particular resource management strategy, called *IDisposable*. In .Net terms, this means that this class must be explicitly handed back to the .Net runtime for resource reclamation rather than just allowed to be garbage collected at a later date. This is a critical piece of information for a .Net programmer. Failing to follow this practice each and every time this class is used will result in failures caused by resource exhaustion. This unit test documents this behavior explicitly, something that is not usually shown in usage examples in the .Net documentation.

Once the object is created, we see how to attach a callback to the ConfigurationChangeFileWatcher's ConfigurationChanged event. This syntax is how .Net allows a programmer to associate a callback method with the event that causes it to be called. We also see that we have to tell the object to start watching the file before it will actively begin watching for changes to that file. The rest of that section just waits for a bit, changes the LastWrite time of the watched file, and waits a bit longer (250 mSec) for the object to notice the change.

To a .Net programmer, this test documents a lot of information in just a few lines of code. It does require the reader to have some knowledge of the underlying technologies, in this case .Net and C#, and also of the domain, to make the test contents relevant to them. But these prerequisites should be easily fulfilled by almost everyone looking at your API documentation. They are, after all, programmers.

One thing to note is that there are certain variables, like numberOfConfigurationChanges, that seem to appear out of nowhere. These extra variables are part of the test class itself. In my opinion, they support the test, but their definitions are

ONCE AND ONLY ONCE MEETS REPLICATED LOGIC

One of the basic precepts of Test Driven Development is something called the principle of Once and Only Once, frequently seen as OAOO. This principle tells you that you cannot permit any duplication at all in your source code because that duplication is going to make your code more complex and harder to maintain. Changes that should only have to be done in one place may have to be done in several, and there is no reliable, automated way to find all of those places. Additionally, duplication in your code may be trying to tell you something. Some abstraction in your system may be trying to find its way out of your code, and it may be talking to you through that duplication. Listen to it, expose it through judicious refactorings, and eliminate the duplication.

The creators of the many testing frameworks for Agile development, including JUnit and NUnit, have taken this principle to heart and given their users a standard way to share code common to individual test cases. Each of these frameworks provides setup and teardown methods that are automatically called before and after each unit test method runs. The purpose of these extra methods is to allow you to factor common test fixture creation and tear down logic into its own place, eliminating any duplication in that logic.

Before I started thinking about tests as documentation, I would aggressively refactor my test code to move any replication into these methods. This would leave only the unique code inside each test method. But once I started thinking about the documentation aspects of unit tests, I began reconsidering this behavior. The problem with it, I was finding, was that it made individual tests harder to read and understand. I could not just read a single unit test and understand each of its three "A's" (arrange/act/assert). I had to go look in the SetUp and TearDown methods to get the context in which the tests were operating and then go look at the unit test, which took away from its understandability. Because of this, I've stopped moving logic that is critical to understanding a unit test out of that test. I make a point to keep code that helps *arrange* or *act* inside each test. Many tests still have extra logic whose purpose is to set up and create incidental, housekeeping data, and I'll still refactor this common logic into SetUp and TearDown, but this doesn't affect a test's readability. And if it does, I'll move this logic back. I find this makes tests more readable on their own, improving their usefulness as documentation.

The other side of the issue is that I do find myself occasionally paying for this replication. At times, if the underlying code changes, I have to change some of this replicated logic in several places. I accept this as a cost of making the tests more understandable, but this is a factor to consider.

not critical to understanding the test. Their names should be clear enough to document their intent. Things like this are frequently factored out of each individual unit test and left to the testing infrastructure to manage and initialize. If it turns out that these extra concepts are essential for understanding a test, then they should be moved back into the body of each test.

Beyond Unit Tests

So far the unit tests that we have talked about creating are an important part of Agile Documentation, but, though some in the Agile world would disagree, they are not sufficient in and of themselves. The tests go a long way toward describing how something is used, but it is difficult to get an overview of a class from them. There are some words that are needed. For example, in the class discussed above, I believe a couple of sentences about the ConfigurationChangeFileWatcher would go a long way to getting an understanding of the context of the class described. Something like:

The ConfigurationChangeFileWatcher is responsible for raising events to its registered listeners when the configuration file it is watching is changed. Clients can register a callback method with an instance of this class, and those callback methods will be invoked when the watched file changes.

The big advantage of an overview like this is that it is at such a high level that there is nothing in it that is likely to change as a class evolves. It gives the necessary overview but does it in such a way that it doesn't add to the maintenance burden of the documentation.

Adoption

As I said before, if you are already writing unit tests as a team, then you're a good part of the way to adopting Agile Documentation techniques. All that is needed is to focus a bit more on telling the story of your system through your tests.

If you are not writing unit tests as an organization, all hope is not lost. You just need to start writing them. Individual developers need to start writing unit tests for their code, preferably in a test-first manner, but any sort of unit test will help. And the organization has to agree to value the tests. Programmers, QA, managers, executives—everyone—must understand and agree that these tests form a valuable investment that is worth preserving over time. Once everyone buys in and you are writing tests, you're ready to go on to the next phase.

The next step should be to work together as a team while creating your first few Test Lists. This will allow everyone to come to a common understanding of the level of detail to which your tests should be described and to develop a common naming style for all tests. As a team, you should be very conscious of creating Test Lists that describe the intent of the test, not the test's implementation. By working together, the entire team can gain experience and each member can receive feedback as he learns how this process works. And as a team, when you need to use a method someone else wrote you should start using the Test Lists devised by your team instead of looking at code. By following your own advice, you'll get great feedback on what you are doing right and what you can improve on in writing your Test Lists.

Similarly, when implementing the tests, you should focus on creating tests

that are self-documenting—they should be complete unto themselves, they should be easily readable, and they should describe an entire usage example in a way that a reader later will find useful. The entire team should review all the tests over the first few days, so all involved can come to an understanding about a common style in writing these tests. Again, whenever possible, the team should use these tests instead of looking at source code. The idea here is that you should live and breathe in the same environment as your users, so you can understand what problems and successes they may experience.

The final step of the transition, and probably the hardest, is to evangelize. You've been using Test Lists and unit tests to help you understand the code of others. Start to tell others about it. Blog about it. When a user asks you a question about your code, don't tell him where to look in the code—tell him which test cases to look at. If you have the opportunity, grab a few of your users and walk them through an example of learning about your code through the unit tests. The first steps in this transition may not be easy or comfortable for you or your users, but the payoff comes later. That's when both you and your users can start to get detailed, explicit questions answered about the code without ever having to look at the code.

Next Steps

For larger systems, it will be difficult for users to find the tests they need. Larger systems can have thousands of tests, so we need to provide some kind of organization to let people find what they're looking for. There hasn't been much progress made in this area yet, but it is coming. We're still fairly early in the unit test adoption process, let alone the tests-as-documentation revolution to follow, so little thought to larger scale organization of tests has happened. But I can see where we'd really like to be in a few years.

The next logical step would be for manual pages, like those created through Javadoc, to have links from application methods to those tests that exercise them. The manual pages would still have the overview documentation in them, but the details would be exposed as links to unit tests. This would all be autogenerated through our documentation tools. But it's going to take a while to get there. In the meantime, there are some things that we can do to provide help to our users.

The first, and most obvious step, is to manually create Test Maps. Test Maps are just documents that list the tests associated with each application method. Manual creation is obviously not the optimal solution, as it suffers from the same maintenance issues as written documentation—there is nothing automated to help you find what needs to be changed, so it must be kept up to date through manual inspection. But giving users this Test Map is the first step toward helping them begin to rely on the unit tests.

THE IDEA HERE IS THAT YOU SHOULD LIVE AND BREATHE IN THE SAME ENVIRONMENT AS YOUR USERS.

The next step along the way is to automate the process of creating the Test Map. I'm writing a tool that will use markups of test code to create this Test Map for me. This will automate the process of creating the map, but it still relies on the programmer's manually maintaining the markup on the test methods, so we're still not all the way there.

The final step along the way is to figure out how to do all of this automatically. This program would need to inspect all the test code and find all application methods used in it and create the Test Map for you without relying on manual markups as before. The hard part about this is that application methods can be mentioned in a unit test without really being part of the test. (For instance, methods can be part of the Arrange section, where the context for a test is set up.) We're really concerned about methods that are exercised by a test, not just all methods mentioned. This is a harder problem.

The other part of the issue is that documentation generation tools will need to be updated to make use of our Test Maps to create manual pages with embedded links to our unit tests. All of this can, and will, happen, but it will take several more years until we start to see these tools.

Conclusion

My original premise for writing this was that our documentation costs on our project were a large percentage of the total budget, and I wanted to find a way to spend more on features and less on writing about the features. Our TDD project had thousands of unit tests lying around waiting to be used for something. This seemed to be a marriage just waiting to happen.

So I started looking at what it would take to combine the two. I had to change my test-creation process a bit by really focusing more on simplicity and naming. I had to look at my tests in a different way and make sure that the tests told the story of my class or subsystem—and that they told the whole story. And I had to get others on my project to think the same way. This involved a cultural change, which we were willing to make. Other organizations may not be willing to change in this way, and this technique may not be applicable to them.

The one piece we're missing, and the extra piece I'm spending some time investigating, is how to organize these tests and make them part of our published documentation. We're still working on this, but we'll get there eventually. **{end}**

Brian Button (agile@agilesolutionsgroup.com, http://www.agilesolutionsgroup.com) is a serious believer in the Agile way of development. He teaches, consults, mentors, and develops code this way, every day.

Sticky Notes

For more on the following topics go to www.stickyminds.com/bettersoftware

- Source code and documentation for TestMap
- Reference to TestMap source code
- Microsoft patterns and practices source code developed through Test Driven Development
- Test Driven Development links and resources