Testing, fun? Really?

Using unit and functional tests in the development process

Jeff CannaRoleModel Software, Inc.

Summary: Testing. Yuck! Puh! Aagh! I've always hated testing. Testing, both unit and functional, is something that gets in the way of the "real" work. Everyone knows that their code is perfect, right? In the unlikely event that the code does need to change, the comments are so well written that anyone could figure it out. Wow, am I in need of growth (maybe some counseling as well.

Date: 01 Mar 2001
Level: Introductory

Over the last few years, unit testing has become central to the way I write software, thanks to a lightweight programming methodology called Extreme Programming (XP) (see Resources). This methodology requires that I write unit tests for every function I add, and that I maintain those tests. I can't integrate any code with failing unit tests. As the code base grows, these tests allow developers to integrate changes with confidence.

Originally I thought the existence of these unit tests would make functional tests unnecessary. Oops, wrong again. Functional tests and unit tests are vastly different. It took me a long time to understand how they are different and how to use them together to enhance the development process.

This article explores the differences between unit testing and functional testing. It also outlines a process for using them in your daily development.

Testing and the development process

As a developer, testing is so important that you should be doing it all of the time. It should not be relegated to a specific stage of the development cycle. It definitely shouldn't be the last thing done before giving your system to a customer. How else are you going to know when you're done? How else are you going to know if your fix for a minor bug broke a major function of the system? How else will the system be able evolve into something more than is currently envisioned? Testing, both unit and functional, needs to be an integrated part of the development process.

Unit tests should become central to how you write code, especially if the project you are working on has tight time constraints and you'd like to keep it under control. Unit tests are so important that you should write your tests before you write the code.

A maintained suite of unit tests:

- Represents the most practical design possible
- Provides the best form of documentation for classes
- Determines when a class is "done"
- Gives a developer confidence in the code
- Is a basis for refactoring quickly

Unit tests constitute design documentation that evolves naturally with a system. Read that again. This is the Holy Grail of software development, documentation that evolves *naturally* with a system. What better way to document a class than to provide a coded set of use cases. That's what these unit tests are: a set of coded use cases that document what a class does, given a controlled set of inputs. As such, this design document is always up-to-date because the unit tests always have to pass.

You should write tests before you write code. Doing so provides a design for the class that the test will exercise, allowing you to focus on small chunks of code. This practice also keeps the design simple. You aren't trying to look into the future, implementing unnecessary functionality. Writing tests first additionally lets you know when the class is complete. When all the tests pass, the task is complete.

Lastly, unit tests provide you with a high degree of confidence, which translates into developer satisfaction. If you run unit tests whenever you make changes to code, you'll find out immediately if your changes broke something.

Functional tests are even more important than unit tests because they verify that your system is ready for release. The functional tests define your working system in a useful manner. A maintained suite of functional tests:

- Captures user requirements in a useful way
- Gives the team (users and developers) confidence that the system meets those requirements

Functional tests capture user requirements in a useful way. Traditional development captures requirements in use cases. Usually, people argue about the use cases and spend a lot of time refining them. When they're finished, all they have is paper. Functional tests are like self-validating use cases. Extreme Programming methodology can illustrate this concept. XP Stories are commitments to a future conversation between the customer and developers. Functional tests are the output of this conversation. Stories without functional tests can't be built very well.

Functional tests fill in the gap left by unit tests and give the team even more confidence in the code. Unit tests miss many bugs. They may give you all the *code* coverage you need, but they might not give you all the *system* coverage you need. The functional tests will expose problems that your unit tests are missing. A maintained, automated suite of functional tests might not catch everything either, but it will catch more than the best suite of unit tests can catch alone.

Unit versus functional tests

Unit tests tell a developer that the code is *doing things right*; functional tests tell a developer that the code is *doing the right things*.

Unit tests
Unit tests are written from a programmer's perspective. They ensure that a particular method of a class successfully performs a set of specific tasks. Each test confirms that a method produces the expected output when given a known input.

Writing a suite of maintainable, automated unit tests without a testing framework is virtually impossible. Before you begin, choose a framework that your team agrees upon. You will be using it constantly, so you better like it. There are several unit-testing frameworks available from the Extreme Programming Web site (see Resources). The one I am most familiar with is JUnit for testing Java code.

Functional tests
Functional tests are written from a user's perspective. These tests confirm that the system does what users are expecting it to.

Many times the development of a system is likened to the building of a house. While this analogy isn't quite correct, we can extend it for the purposes of understanding the difference between unit and functional tests. Unit testing is analogous to a building inspector visiting a house's construction site. He is focused on the various internal systems of the house, the foundation, framing, electrical, plumbing, and so on. He ensures (tests) that the parts of the house will work correctly and safely, that is, meet the building code. Functional tests in this scenario are analogous to the homeowner visiting this same construction site. He assumes that the internal systems will behave appropriately, that the building inspector is performing his task. The homeowner is focused on what it will be like to live in this house. He is concerned with how the house looks, are the various rooms a comfortable size, does the house fit the family's needs, are the windows in a good spot to catch the morning sun. The homeowner is performing functional tests on the house. He has the user's perspective. The building inspector is performing unit tests on the house. He has the builder's perspective.

Like unit tests, writing a suite of maintainable, automated functional tests without a testing framework is virtually impossible. JUnit is very good at unit testing; however, it unravels when attempting to write functional tests. There is no equivalent of JUnit for functional testing. There are products available for this purpose, but I have never seen these products used in a production environment. If you can't find a testing framework that meets your needs, you'll have to build one.

No matter how clever we are at building the projects we work on, no matter how flexible the systems are that we build, if what we produce isn't usable, we've wasted our time. As a result, functional testing is the most important part of development.

Because both types of tests are necessary, you'll need guidelines for writing them.

How to write unit tests

It is easy to become overwhelmed when you start writing unit tests. The best way to start is to create unit tests for *new* code. (It is difficult to start by creating unit tests for existing code, but it is possible.) Start with new code, get used to the process, and then revisit the existing code to create a test suite for it.

As mentioned earlier, you should write unit tests before you write the code they will test. How can you write a test for something that doesn't exist yet? Very good question, Grasshopper. Mastering this practice is ninety percent mental and ten percent technical. What I mean is that you simply pretend that the class you are writing the test for exists. Then write the test. Initially you will get a lot of syntax errors, but stay with it. What you are doing through this exercise is defining the interface that the class will implement. The next step is to run your unit tests, fix the syntax errors (that is, implement the class with the interfaces just defined by your test), and run the tests again. Repeat this process, each time writing just enough code to fix the failures. Run the tests until they pass. The code is "done" when all of the unit tests pass.

In general, there should be a unit test for every public method of your class. However, methods with very straightforward functionality, for example, getter and setter methods, don't need unit tests unless they do their getting and setting in some "interesting" way. A good guideline to follow is to write a unit test whenever you feel the need to comment some behavior in the code. If you're like many programmers who aren't fond of commenting code, unit tests are a way of documenting your code behavior.

Put the unit tests in the same package as the associated classes being tested. This type of organization allows each unit test to call methods and reference

variables that have access modifiers of `package` or `protected` in the class being tested.

Avoid using domain objects in unit tests. Domain objects are objects specific to an application. For example, a spreadsheet application might have a register object; this object would be a domain object. If you have a class that already knows about the domain objects, it is fine to use these objects in your tests. But if you have a class that isn't using these objects, do not tie these objects to the class through the tests. The reason this practice should be avoided is all wrapped up with code reuse. Very often the classes created for a project apply to other projects. Reusing these classes may be straightforward. But if the tests for the reused classes use another project's domain objects, getting the tests to work can become a very time-consuming activity. Usually the test will either be dropped or rewritten.

These mechanics will serve you well, but a comprehensive suite of unit tests will not be worth anything if you don't run the tests. Running the tests early and often gives you absolute confidence in your code all the time. As the project proceeds, you will add features. Running the tests will tell you if the new features you've just implemented have broken something.

Revisit your existing code after you have mastered the mechanics of writing unit tests. Writing tests for existing code can be a challenge. Don't test for testing sake. Write tests in a "just-in-time" fashion, when you find the need to modify a class that doesn't have good (or any) tests. That is the time to add the tests. As always, the unit tests for that class should capture the functionality for each of its methods. One of the easiest ways to find out what the test should be testing is to look at the comments in the existing code. Any comment should be captured in a unit test. Translate block comments at the beginning of methods describing what the method does into unit tests.

How to write functional tests

Even though functional testing is so important, it has a reputation as the ugly stepchild of development. On most projects, there is a separate group that does functional testing. There is usually an army of people constantly interacting with the system to determine whether it behaves correctly. This attitude and group setup is foolishness.

Functional testing should be approached much like unit testing. Write the tests as soon as there is code to be written that produces something a user will interact with (such as a dialog), but before actually writing the code. Work with a user to write functional tests that capture the user requirements. Whenever you start a new task, describe the task in the functional testing framework. Your development effort then moves forward, unit testing when you add new code. When all of the unit tests pass, run the original functional test to see if it is passing or if it needs modification.

Ideally, the concept of a functional testing group should disappear. Developers should be writing functional tests with users. After there is a suite of functional tests for the system, the members of the development team responsible for functional testing should bombard the system with variations of the initial tests.

The line between unit and functional testing

Often it isn't clear where to draw the line between unit and functional testing. To be honest, it isn't always clear to me where this line is either. While writing unit tests, I have used the following guidelines to determine if the unit test being written is actually a functional test:

- If a unit test crosses class boundaries, it might be a functional test.
- If a unit test is becoming very complicated, it might be a functional test.
- If a unit test is fragile (that is, it is a valid test but it has to change continually to handle different user permutations), it might be a functional test.
- If a unit test is harder to write than the code it is testing, it might be a functional test.

Notice the phrase "it *might* be a functional test." There are no hard and fast rules here. There is a line between unit tests and functional tests, but you have to decide where the line is. The more comfortable you get with unit tests, the clearer it will be when a particular test is crossing the line from unit to functional.

Conclusion

Unit tests are written from the developer's perspective and focus on particular methods of the class under test. Use these guidelines when writing unit tests:

- Write the unit test before writing code for class it tests.
- Capture code comments in unit tests.
- Test all the public methods that perform an "interesting" function (that is, not getters and setters, unless they do their getting and setting in some unique way).
- Put each test case in the same package as the class it's testing to gain access to package and protected members.
- Avoid using domain-specific objects in unit tests.

Functional tests are written from the user's perspective and focus on system behavior that users are interested in. Find a good functional testing framework, or develop one, and use these functional tests to identify what the user really wants. In this way, the functional tester gains an automated tool and has a starting point for using the tool.

Make unit testing and functional testing central to your development process. If you do, you will have confidence that your system works and can grow. If you don't, you can't be sure. Testing may not be fun, but having working unit and functional tests makes development a lot more fun.

Resources

- IBM® Rational®functional testing solutions enable you to automate regression tests to ensure that new changes to your application have not impaired existing functionality.
- Download various unit testing frameworks from the Extreme Programming Web site.

About the author

Jeff Canna has developed software systems since 1982. His experience ranges from mainframes to hand-held platforms. Mr. Canna's primary experience has been in the development of GUIs on UNIX platforms. Recently he has focused on delivering content via server-side Java technology. He has also become very energized by a new programming methodology called Extreme Programming, which is changing his approach to project development. Contact Mr. Canna at jcanna@rolemodelsoft.com.

Close [x]

developerWorks: Sign in

If you don't have an IBM ID and password, register here.

IBM ID:
Forgot your IBM ID?

Password:
Forgot your password?
Change your password

After sign in: Stay on the current page

Keep me signed in.

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

The first time you sign into developerWorks, a profile is created for you. This profile includes the first name, last name, and display name you identified when you registered with developerWorks. **Select information in your developerWorks profile is displayed to the public, but you may edit the information at any time**. Your first name, last name (unless you choose to hide them), and display name will accompany the content that you post.

All information submitted is secure.

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

Average rating (450 votes)

1 star 1 star
2 stars 2 stars
3 stars 3 stars
4 stars 4 stars
5 stars 5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

Notify me when a comment is added1000 characters left

Post

Be the first to add a comment