测试有趣？真的吗？

在开发过程中怎样利用单元和功能测试

作者：Jeff Canna (jcanna@rolemodelsoft.com)
RoleModel Software, Inc.
March 2001
翻译：lyre[AKA]

> 测试。呸！令人厌恶的东西！我一直憎恨它。单元测试和功能测试都是一些所谓“真正”的工作。因为每个人都认为他编写的代码是完美的，不是吗？然而事实未必如此，代码需要变化，注释也同样需要明晰可见。哇喔，我也一样需要成长。（也许律师也需要这些。）

在过去的几年中，单元测试逐渐成为我编写软件的核心内容，在这里要感谢一种叫做极端编程-XP（注 1）(见“资源”一节)的简便程序设计方法。这种方法要求我为新加入的每个函数都编写单元测试，并且维护这些测试。没有通过单元测试，我就不能将任何一个的代码加到模块中。在代码基数增长的同时，这些测试允许开发者有依据地将改变集成起来。起初，我认为这些单元测试就足以应付全局，没有必要涉及到功能测试。噢，又错了。功能测试和单元测试完全不同的两者。我花费了很长的时间才理解到两者的区别，以及如何将它们结合起来，用以改进开发进程。

本文探讨了单元测试和功能测试之间的差别，同时介绍在你的日常开发的过程中如何来利用它们。

测试和开发过程

作为一个开发人员，测试如此之重要，以至于你甚至应该花费几乎所有的时间来完成它。它不仅需要只被划分为开发过程中的某个特定阶段。显然，它不该是在你把系统交付给客户之前完成的最后一项任务。然而，你又如何得知它在何时结束呢？或是你如何得知是否因为修改一个微小的 bug 而破坏了系统的主要功能呢？或是系统可能会演化成超乎现在想象的模样？测试，单元的和功能的都应该是开发的过程中的一部分。

单元测试应成为你编写代码的核心环节，尤其当你在从事一个项目时，紧张的时间约束你的开发进度，你也很想让它是在可控的有序下进行。我希望测试也是在你编写代码之前编写测试时的重要内容。

一套适用的单元测试应具备以下功能：

- 说明可能的最佳适用设计
- 提供类文档的最佳格式
- 判断一个类何时完成
- 增强开发人员对代码的信心
- 是快速重构的基础

在系统中自然要包含单元测试所需的设计文档。重新阅读它，你会发现这是软件开发进程中的圣杯，文档跟随系统的变化而逐步演化。为每一个类提供完备的文档比起为它提供一系列的使用框架，或是一系列可控的输入要好得多。这样，设计文档就会因为单元测试的逐步通过而随时更新。

你应该在你编写代码之前完成编写测试的工序。这样做会为测试所涉及的类提供设计方案，并促使你关注代码中更小的程序模块。这种练习也会使设计方案变得更加简单。你不能试图去了解将来的情形，去实现不必要的功能。编写测试工作也会让你清楚类会在什么时间结束。可以说，当所有的测试通过时，任务也就完成了。

最后，单元测试会提供给你更高级别的依据，这绝对会满足开发者的。如果你在改动代码的同时，进行单元测试，你就会在你破坏的同时立即察觉到事态的发生。

功能测试甚至比单元测试更加重要，因为它们说明了你的系统就要预备发布了。功能测试将把你的工作系统放置于一个可用的状态中。

一套适用的功能测试应具备以下功能：

- ✍ 有效地掌握用户的需求
- ✍ 向项目组成员（包括用户和开发者）给出系统面临这些需求的依据

功能测试要在有效地情况下掌握用户的需求。而传统的开发者是在使用的过程中发现需求的。通常，人们赞同使用项目工程并且花费相当的时间去重新定制它们。当它们被完成时，它们所得到的仅仅是一堆废纸。功能测试雷同于自行生效的使用项目的情况。极端程序设计方法（Extreme Programming）能够说明这种概念。XP 的说法就是对未来发生在用户和开发者之间的交流技巧的描述。功能测试也是这种交流的结果。而没有功能测试，这种说法也不会建立起来的。

功能测试恰好填充了在单元测试和向项目小组提交的代码依据之间的空隙。单元测试会漏过许多的 bug。它可以给出代码中你所需的所有有效部分，它也会给你所需的整个系统。功能测试可以使单元测试里漏掉的问题曝光。一系列可维护的，自动化的功能测试也会有漏网的情况，但是它至少比独立地进行最全面的单元测试要有用得多。

单元测试 VS 功能测试

单元测试告诉开发者代码使事情正确地被执行，而功能测试所说的则是代码在正确地发挥功效。

单元测试

单元测试是从开发者的角度来编写的。它们确保类的每个特定方法成功执行一系列特定的任务。每一个测试都要保证对于给定的一个已知的输入应该得到所期望的输出。

编写一系列可维护、自动化、没有测试框架的单元测试几乎是不可能的。在你开始之前，选择一个项目小组都认可的框架。不断地应用它，逐渐地喜欢它。在极端编程的介绍网页上（见资源一

节），有很多适用的单元测试框架。我喜欢用的是 Juint 来进行 Java 代码的测试。

功能测试

功能测试则是从用户的角度来编写的。这些测试保证系统能够按照用户所期望的那样去运行。

很多时候，开发一个完整的系统更像是建造一座大楼。当然，这种比喻并不是完全地恰当，但我们可以扩展它，来理解单元测试和功能测试之间的区别。

单元测试类似于一个建筑检查员对房屋的建设现场进行检查。他注重的是房屋内部不同的系统，地基，架构设计，电气化，垂直的线条等等。他检查房屋的某个部分，以确保它在安全状态下，正确无误地工作，即是说，直接针对房屋的代码。功能测试在这个剧本里类似于房屋的主人在检查同样的建设场地。他所期望的是房屋的内部系统正常地运转，并且房屋检查员执行了他的任务。房屋的主人看重的是生活在这样的房屋中会是什么样子。他关注这间房屋的外貌，不同的房间有合适的空间，房屋适用于家庭的需要，窗户恰好位于最佳采光的位置。房屋的主人运行的是对房屋的功能测试，他站在用户的角度上。房屋检查员运行的是单元测试，他是站在建设者的角度上。

象单元测试一样，编写一系列可维护、自动化、没有测试框架的功能测试几乎是不可能的。Junit 在单元测试方面做得很好；然而，它在试图编写功能测试时就显得比较松散。Junit 不等同于功能测试。现在已经有满足这个功能的产品问世了，但是我还没有看到它们被应用于开发产品过程里。如果你不能找到一个测试框架的话，就只好自己创建一个了。

无论我们在建立一个项目时多么聪明，建立的系统多么灵活，如果我们的产品不能用，我们就是在浪费时间。结论是，功能测试是开发进程中最重要的一部分。

因为两种类型的测试都是必要的，你会需要编写它们的指南。

如何编写单元测试

在你开始编写单元测试很容易被激动的情绪感染。最简单的起步方式就是为新的代码创建单元测试。为已经存在的代码创建单元测试是一种比较有难度的开始方式，但是也是可行的。）从新的代码开始，习惯了这样的步骤后，还要坚持重新阅读现有代码，并为它们创建一套测试程序。

就像前面提到过的一样，你应该在你编写要测试的代码之前编写单元测试。如何做到为还不存在的事物编写测试呢？好问题！掌握这个能力需要 90%的智力和 10%技巧。我的意思是你只需假装是在为已有的类编写测试。接下来，进行编写的工作。最初，你将出现很多语法错误，但是 let it be，不要理会它。紧接着进行单元测试，修改语法错误（即是说，只用你自己定义的测试接口来实现类），再一次进行测试。重复这个过程，每一次都写下充足的代码去修改错误，进行测试直到它们通过为止。当所有的单元测试都通过时，代码才算真正地完成了。

一般地说，你的类应具有开放的单元测试方式。然而，带有直截了当的功能性的方法比如说，Java 语言里的 Getting 和 Setting 读写方法，就不需要单元测试，除非它们是以“特殊”的方式进行

的。接下来的指导就是，当你感到需要对代码中的某些特性添加注释时，同时要编写出单元测试。如果你同很多的程序员一样，厌恶为代码写注释，单元测试就是将你的代码的特性文档化的一种好方法。

将单元测试同被测试的相关的类打包在一起。（这种组织的方式允许每一个单元测试都能够直接访问类中被打包和保护的方法和参数）。

要避免在单元测试中用到域对象（domain object）。域对象就是对于一个应用程序特定的对象。例如，电子表格应用程序有个工作簿对象，它就是一个域对象。如果你的一个类已经知道了域对象，在你的测试中用到这些对象是很好的。但是如果你的类没有涉及到这些对象，就不要在测试里让它们同类纠缠不清了。不这样做的话，就会产生打包的代码被重用。经常是为一个项目创建的类也可以应用于其他的项目，这样可能会出现直接重用这些类的情况。但是如果针对这些类的测试也用于另外的项目对象，让测试生效会很费时，通常测试不是被抛弃掉就是被重新编写。

以上的一些技巧会让你从中受益，但最重要的是如果你不实际地去做，就永远不会对单元测试有全面、深入的理解。更早地运行测试，并且在整个过程中都在代码中给出全面的依据。当项目进展时，你会随时添加更多的特性。运行测试就会提醒你，实现刚添加的特性会不会破坏已有的东西。

在你已经掌握编写单元测试的技巧之后，你需要重新阅读已存在的代码。的确，为它们编写代码可能会是一场挑战。但是千万不要为了测试的目的而测试。可以说，编写测试是一件紧跟时效的事情，尤其是当你发现要修改一个没有好的测试程序的类时，那就是添加测试的恰当时机。和平常一样，单元测试应该具备类每个方法的特性。实现测试的一个最简单的方法就是，测试的同时一定要注意代码的注释。在单元测试中，不能放过任何一个注释，在描述测试方法的开始就要为单元测试添加大量的注释中。

如何编写功能测试

尽管功能测试是如此重要，它也有个开发过程里丑陋的继生子的坏名声。在大多数的项目里，是由一个独立的工作组来完成功能测试的工作。通常需要一群人在系统中的相互协助才能保证工序的正确运行。这种通常的看法和队伍的组建的做法，都是非常愚蠢的。

功能测试同单元测试相类似。一旦要编写有用户涉入的产品的代码（例如，对话框）时，就要编写测试，但是一定要在实际编写代码之前做。一旦你开始了一项新任务，就要在功能测试的框架里清楚地描述这个任务的内容。你加入的新代码的同时进行单元测试，开发工作就向前持续进行。当所有的单元测试都进行通过后，再进行最初的功能测试来判断项目是否可以通过，或是需要修改。

理想的状况下，功能测试小组的概念应该不存在的。开发者应该同用户一同编写功能测试。系统通过了一系列的单元测试后，负责进行功能测试的小组成员就要改变初试测试的参数，再进行系统的功能测试。

单元测试和功能测试之间的界线

一般情况下，很难划清在单元测试和功能测试之间的界限。说实话，一直以来，我就不知道这个界线应该定在哪里。当编写单元测试时，我用以下几个方法来判定单元测试是不是已经变成了功能测试：

- 如果单元测试超越了类之间的界限，它可能变成了功能测试
- 如果单元测试变得非常的复杂，它可能变成了功能测试
- 如果单元测试变得很脆弱（即是说，它已经成为一个测试，但是却因为要迎合不同用户需求的改变而被动地变化），它可能变成了功能测试
- 如果单元测试比需要测试的代码还要难于编写，它可能变成了功能测试

注意“它可能变成了功能测试”的说法，在这里没有严格的标准。在单元测试和功能测试之间是有界线的，但是你必须自己判定它在哪里。单元测试进行地顺利，特定的测试逾越两者界线的过渡就越明显。

结论

单元测试以开发者的角度来编写，并注重被测试类的特性。当编写单元测试时，利用以下几条指导：

- 在类代码进行测试之前编写单元测试
- 在单元测试里掌握代码的注释
- 测试所有执行特定功能的公用程序（即是说，和 Java 语言中的 Getting 和 Setting 读写方法不同的方法。除非它们是通过一种特殊的方式来完成 Getting 和 Setting 功能的。）
- 将所有的测试项目同被测试的类打包在一起，并且分配它们对在模块包内的和被保护成员的访问权限
- 在单元测试中避免使用某些特定的对象

功能测试也需要从用户的角度出发来编写，并且注重用户所感兴趣的系统功能。选择一个适当的功能测试框架，或是开发出一种，并利用这些功能测试来制定用户们想要的东西。通过这种方式，功能测试的人员可以获得一个自动的工具，并且对使用工具的习惯有了一个好的起点。

将单元测试和功能测试作为开发进程的核心内容。这样做，你就会确定系统在正常运转。如果没有，你恐怕不能保证系统是正常工作的。测试可能不是一件好玩的事情，但是从事单元测试和功能测试会使开发过程里含有更多的乐趣。

资源

- “利用 Ant 和 JUnit 改进开发过程”（开发工作，2000 年 12 月）揭示了单元测试的益处，尤其是应用了 Ant 和 Junit 之后。
- 开始了解极端编程的方法
- 从极端编程的网页上下载各种单元测试的框架

本文作者

Jeff Canna 从 1982 **年开始从事软件系统的开发工作。他在大型机和手提式平台上有很丰富的开发经验，其中擅长在** UNIX **上的** GUIs **的开发工作。最近他主要从事通过服务器端的** Java **技术来传送内容的技术工作。同时他对一种叫做极端编程的开发方法非常感兴趣，这种开发方法改变了他一贯处理项目开发方面的工作方式。同** Canna **先生联系：** jcanna@rolemodelsoft.com**。**

译者注：

1、**极端编程**-XP（Extreme Programming）**是一种轻量级软件开发方法学，为中小型团队而设计，特别适合在需求迅速变化的环境中快速地开发软件。**XP **的目标是在用户需要的时候交付软件。**XP **能够迅速提高软件开发生产力和产品质量。**

自由、协作、创造 — 为了明天
“来自大雪山的大雁阿卡”

更多精彩文章，请访问：http://www.AKA.org.cn **精彩文章 栏目**
本文如有翻译错误或不妥，请 Email **至** AKAMagazine@yahoo.com

附件：英文原文

译者注：本文原文来自：http://www-106.ibm.com/developerworks/library/j-test.html ***在*** *dW* ***中国网站还有该文的一篇更好的译文：***http://www-900.ibm.com/developerWorks/java/j-test/index.shtml

Using unit and functional tests in the development process

Jeff Canna (jcanna@rolemodelsoft.com)

RoleModel Software, Inc.

March 2001

> *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.*

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

- "Incremental development with Ant and JUnit" (developerWorks, November 2000) explores the benefits of unit testing, in particular using Ant and JUnit.
- Become acquainted with the methodoligies of Extreme Programming.
- 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.