Achieving low defect density through a simple automated unit test infrastructure – A practical case study

Balasubrahmanyam P
Intuit Technology Services Private Limited,
Campus 4A, PrlTech Park (Ecospace),
7th Floor, Belandur Village,
Bangalore, Karnataka, India
bpillalamarri@intuit.com

Intuit Technology Services Private Limited.
Campus 4A, PrlTech Park (Ecospace),
7th Floor, Belandur Village,
BANGALORE, India
bpillalamarri@intuit.com

Abstract:
Use of test framework for automated testing – unit and system has evolved over time and is mainly driven by use of third party automation tools. With technology advancements, it forced test engineers to look at other options to automate the test execution process including unit, system and regression testing. Majority of the frameworks in existence is on MFC. With application development becoming more and more complex, there was a need to develop better test frameworks that would be scalable, reliable and easy-to-use. Most often, adequately doing unit testing seems to be a less interesting area for developers – be it the drive or skills. Testing can be as technically challenging as development – and often more challenging especially when it comes to dealing with code level testing. On top of that, unit testing is an effort intensive activity and if not done properly, has a huge impact on down stream QA phases. Unit testing is no longer just a "debugging" activity done through IDEs, instead QA can make a huge impact by catching bugs early in the product life cycle in turn reducing the cost, effort, number of test cycles and delays in the release. This plays a major role in agile testing. As of result of aforesaid activities, development of automated unit test frameworks has become the norm of the day. There are many open source unit test tools available for different technology areas..

Keywords:
NUnit, Unit testing, automated unit test infrastructure.

1 Introduction

This paper talks about effective implementation of an automated unit-test framework using open source unit tests in C# and .Net framework on a new development project. This project is a re-engineering effort to replace an existing legacy platform framework to achieve better performance improvements and quality of the code delivered to the application development teams for GUI development. The tests existed for legacy platform code needed to be reengineered to be able to run them on the new technology framework. It involved developing an entirely new test framework using C# and .NET that eventually led to an ambitious quality goal of 80% code coverage. Prior to this implementing this framework, products at Intuit were tested for code-coverage using heavy tools like McCabe IQ. But, using these heavy weight tools and subsequently triaging them involves a lot of time and resources and hence the need to start using

light weight tools (NUnit, NCover and NCoverExplorer) which would give results of code coverage within no time and very well fit in to the agile development process. A test framework was built using these tools which is reliable, scalable and can generate reports based on function coverage as well as sequence-point coverage (statement coverage). This would give a better idea of where our code stands in terms of the coverage numbers. By using these tools we continuously ensure that we don't fall below the set code coverage goal. Bugs found using these unit tests include but not limited to garbage collection bugs (GC), bugs related to exception handling. Having achieved this success, other projects have started to use this as a best practice in their projects due to its generic nature and coverage reporting.

1 Introduction to Unit tests

A unit is the smallest testable part of an application. Unit testing is a procedure used to validate that individual units of source code are working properly. Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed. Readily-available unit tests make it easy for the programmer to check whether a piece of code is still working properly. Good unit test design produces test cases that cover all paths through the unit with attention paid to loop conditions. In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained. Unit testing provides a sort of "living document". Clients and other developers looking to learn how to use the module can look at the unit tests to determine how to use the module to fit their needs and gain a basic understanding of the API.

Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

Use of test framework for automated testing – unit and system has evolved over time and is mainly driven by use of third party automation tools. With technology advancements, it forced test engineers to look at other options to automate the test execution process including unit, system and regression testing. Majority of the frameworks in existence is on MFC. With application development becoming more and more complex, there was a need to develop better test frameworks that would be scalable, reliable and easy-to-use. Unit test frameworks are a key element of popular development methodologies such as eXtreme Programming (XP) and Agile Development. Unit testing has moved far beyond eXtreme Programming; it is now common in many different types of application development. Unit tests help ensure low-level code correctness, reduce software development cycle time, improve developer productivity, and produce more robust software. Most often, adequately doing unit testing seems to be a less interesting area for developers – be it the drive or skills.

Most people who write software have at least some experience with unit testing. If you have ever written a few lines of throwaway code just to try something out, you've built a unit test. On the other end of the software spectrum, many large-scale applications have huge batteries of test

cases that are repeatedly run and added to throughout the development process. Unit tests are useful at all levels of programming.

Testing can be as technically challenging as development – and often more challenging especially when it comes to dealing with code level testing. On top of that, unit testing is an effort intensive activity and if not done properly, has a huge impact on down stream QA phases. Unit testing is no longer just a “debugging” activity done through IDEs, instead QA can make a huge impact by catching bugs early in the product life cycle in turn reducing the cost, effort, number of test cycles and delays in the release. This plays a major role in agile testing.

Agile testing involves testing from the customer perspective as early as possible, testing early and often as code becomes available and stable enough from module/unit level testing. Unit testing plays a very important role in agile testing.

A single unit test should test a particular behavior within the production code. Its success or failure validates a single unit of code. Well-written tests set up an environment or scenario that is independent of any other conditions, then perform a distinct action and check a definite result. These tests should avoid dependencies on the results of other tests (called test coupling), and they should be short and simple. By starting with tests of the most basic functionality, then gradually building to tests of compound objects and behaviors, a unit test framework can be used to verify very complex architectures. Having such a test framework to build upon not only is much easier than developing standalone tests, but also produces more thorough, effective tests. A comprehensive suite of unit tests enables rapid application development, since the effects of every change can be immediately and thoroughly verified.

In the traditional jargon of testing, tests are categorized as black box or white box, depending on the amount of access to the internal workings of whatever is being tested. Functional and structural tests are related ideas. For example, a test that simply runs a program and checks its return code is a black box (functional) test, since nothing is known about how the program is written. Unit tests are usually white box (structural) tests, since the test framework is able to access the internal structure of the code being tested. Most object-oriented languages provide access protection, preventing outside classes from accessing protected or private code elements. Because of this, unit tests often are written to test only the public interfaces of the objects tested. This encourages the design of objects with discrete, testable interfaces and a minimum of complex hidden behavior. Thus, writing testable objects promotes good object-oriented development practices.

Writing simple tests comes naturally to most programmers. The classic beginner exercise of writing a three-line program that prints "Hello world!" is a basic unit test of the development language and environment. Find a software shop with no unit test framework in place (if such a prehistoric place could possibly exist), and you may see developers writing their own little "toy programs" or "test utilities" to try out new code. The sad thing about this approach is that the toy programs are thrown away once the developer is done with them. Later, when something breaks, someone has to laboriously debug the production code, without benefit of the developer's test.

Another common low-level testing technique is to build tests into the production code with ASSERT macros. In debug builds, the macro tests a condition and sends a message if it fails. In release builds, the macro is defined to be empty, so no test code is included. This allows a developer to sprinkle assertions throughout the code, reporting any condition that is worthy of someone's attention. Asserts can be a useful thing to have in your software toolbox, but far less so than true unit testing. For an assert to be evaluated, the production code must be run to the point

where it is defined. It's not convenient for automated testing, since an automated system doesn't know how to cause a particular assert to fire. Failures don't leave the developer with a clear path to correct the problem. Fixing a failure is no guarantee that the same problem will not happen again under different circumstances. Reliance on testing with this type of assert is unlikely to produce high-quality software. It is a forerunner to formal unit testing, which uses test asserts contained within well-defined tests, rather than placed randomly in the production code.

Just as many developers take the initiative and write test programs to try out small pieces of code, it's common to find developers putting together basic, home-grown unit test frameworks that take care of their testing needs. A test framework can be just a few lines of code to run unit tests and report the results. Even a very simple framework can be the foundation for thorough testing of complex applications.

A useful rule of thumb is that a test method should only contain a single test assertion. The idea is that a test method should only test one behavior; if there is more than one assert condition, multiple things are being tested. When there is more than one condition to test, then a test fixture should be set up, and each condition placed in a separate test method.

As a project grows in size, organizing the files containing production and test code becomes an issue. Although keeping the test and production code in the same directory is the simplest solution, it is better to have a clean separation between the two categories of code. This strategy helps avoid build complications that occur when a directory contains some code that should be linked into the production application and some that should not. Including the test code in the delivered application is undesirable because it unnecessarily increases the size of the delivery, and also because the tests may expose behavior or design details that the developer meant to keep "under the hood."

2 What are unit test frameworks and how are they used?

Simply stated, they are software tools to support writing and running unit tests, including a foundation on which to build tests and the functionality to execute the tests and report their results. They are not solely tools for testing; they can also be used as development tools on a par with preprocessors and debuggers. Unit test frameworks can contribute to almost every stage of software development, including software architecture and design, code implementation and debugging, performance optimization, and quality assurance.

Unit test frameworks are a key element of Test Driven Development (TDD), also known as "test-first programming." TDD is one of the most significant and widely used practices in Extreme Programming (XP) and other Agile Development methodologies. Test frameworks achieve their maximum utility when used to enable TDD, although they still are useful when TDD is not followed. This paper concentrates on unit test frameworks as a family of tools, rather than specifically on TDD, though the two topics are closely related.

Unit test frameworks are valuable when used for automated software testing as part of a quality assurance (QA) process. In many software development groups, the QA process starts when new code is submitted, built, and unit tested. Often, the unit tests include not only programmer tests, but also acceptance tests designed or written by the QA team. If all the unit tests succeed, the code is provisionally accepted and sent to a QA engineer for inspection and testing. Running the full suite of unit tests as the first step in QA has many benefits. Most importantly, the tests ensure

that the code is solid the moment it has left the developers' hands. No human intervention is required to run the tests and evaluate the results. Either they all succeed, or there is a failure. The success of the unit tests confirms that the developers' assumptions are valid, and that the low-level functionality is working correctly at a level of scrutiny that functional tests can never achieve. When numerous developers are making changes at once, the unit tests provide confidence that nobody's changes caused someone else's code to break. Furthermore, unit tests help to provide accountability. Knowing exactly which test fails usually makes it apparent whose change broke things. "Breaking the build" once meant submitting code that caused a compile to fail, but now often refers to causing a unit test failure as well. Many teams employ heinous punishments (such as making the responsible developer buy donuts or beer for coworkers) to remind everyone that breaking the build is a serious offense. The failure of a unit test clearly places a high priority on fixing the problem.

NUnit is a unit test framework for the Microsoft .NET architecture. Conceptually, it follows the xUnit model, serving as a foundation for building unit test classes and methods. It is implemented in C#, but supports writing unit tests in any .NET language, including C#, J#, Managed C++, and Microsoft Visual Basic .NET (VB.NET). NUnit defines tests using C# attributes rather than object inheritance, so the details of its software architecture differ significantly from JUnit. NUnit is open source software released under a public license. The license permits NUnit to be freely redistributed and altered, as long as the original copyright notice is included and any alterations are acknowledged. The copyright holders and main developers of NUnit are James Newkirk, Michael Two, Alexei Vorontsov, Philip Craig, and Charlie Poole.

NUnit is a full-featured unit test framework built using TDD. The distribution includes unit tests covering all of NUnit's functionality. Aside from the core framework, NUnit also includes GUI and console test runners, code samples, extensions, and utilities. NUnit relies on C# attributes to structure test code. In contrast to the conventional object-oriented definition of an attribute, a C# attribute is metadata attached to a code element such as a class or method. These attributes contain descriptive declarations that may be accessed at runtime. NUnit attributes such as Test and TestFixture allow the test framework to identify test methods and classes. This approach makes it possible to build unit tests with minimal knowledge of the underlying NUnit code structure.

3 Case Study

An automated unit-test framework was developed using open source unit tests in C# and .Net framework on a new development project. This project is a re-engineering effort to replace an existing legacy platform framework to achieve better performance improvements and quality of the code delivered to the application development teams for GUI development. The tests existed for legacy platform code needed to be reengineered to be able to run them on the new technology framework. It involved developing an entirely new test framework using C# and .NET that eventually led to an ambitious quality goal of 80% code coverage. Prior to this implementing this framework, products at Intuit were tested for code-coverage using heavy tools like McCabe IQ. But, using these heavy weight tools and subsequently triaging them involves a lot of time and resources and hence the need to start using light weight tools (NUnit, NCover and NCoverExplorer) which would give results of code coverage within no time and very well fit in to the agile development process. A test framework was built using these tools which is reliable, scalable and can generate reports based on function coverage as well as sequence-point coverage (statement coverage). This would give a better idea of where our code stands in terms of the coverage numbers. By using these tools we continuously ensure that we don’t fall below the set

code coverage goal. Bugs found using these unit tests include but not limited to garbage collection bugs (GC), bugs related to exception handling. Having achieved this success, other projects have started to use this as a best practice in their projects due to its generic nature and coverage reporting.

As mentioned earlier, our framework surrounds around NUnit which is the tool used to execute unit tests in .Net and the following image shows a sample executed set of tests.

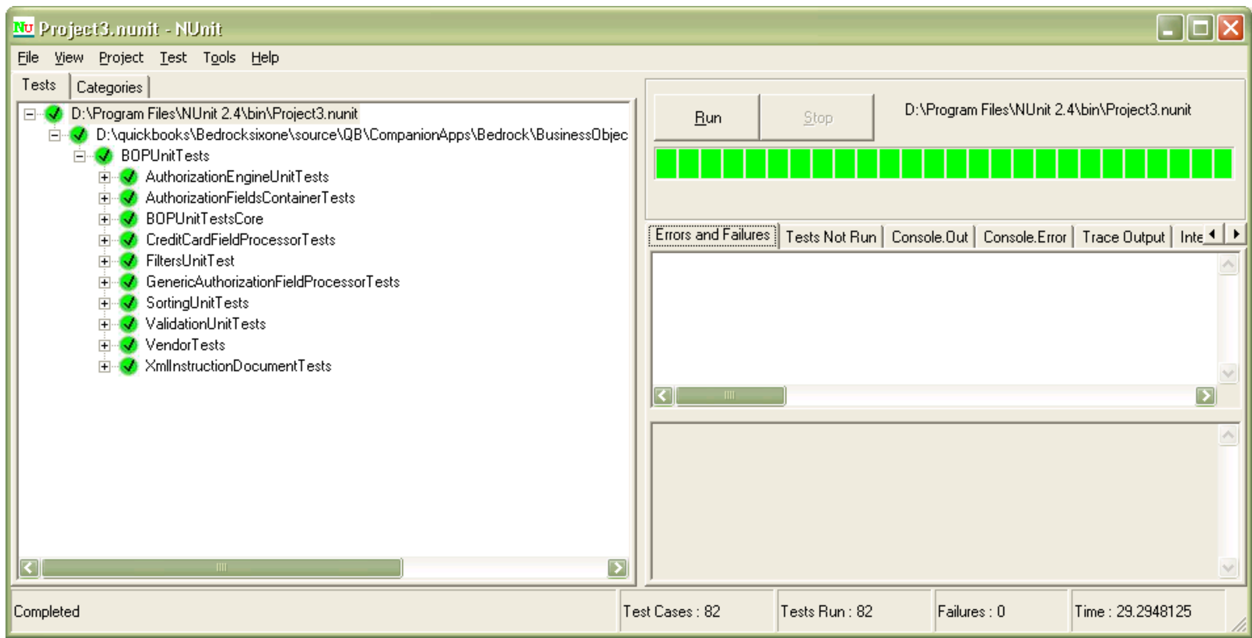

Figure 1. NUnit tests window.

NCoverExplorer is the tool that would let us know the percentage of the code covered. Following images are generated at various levels of test execution and report generation.

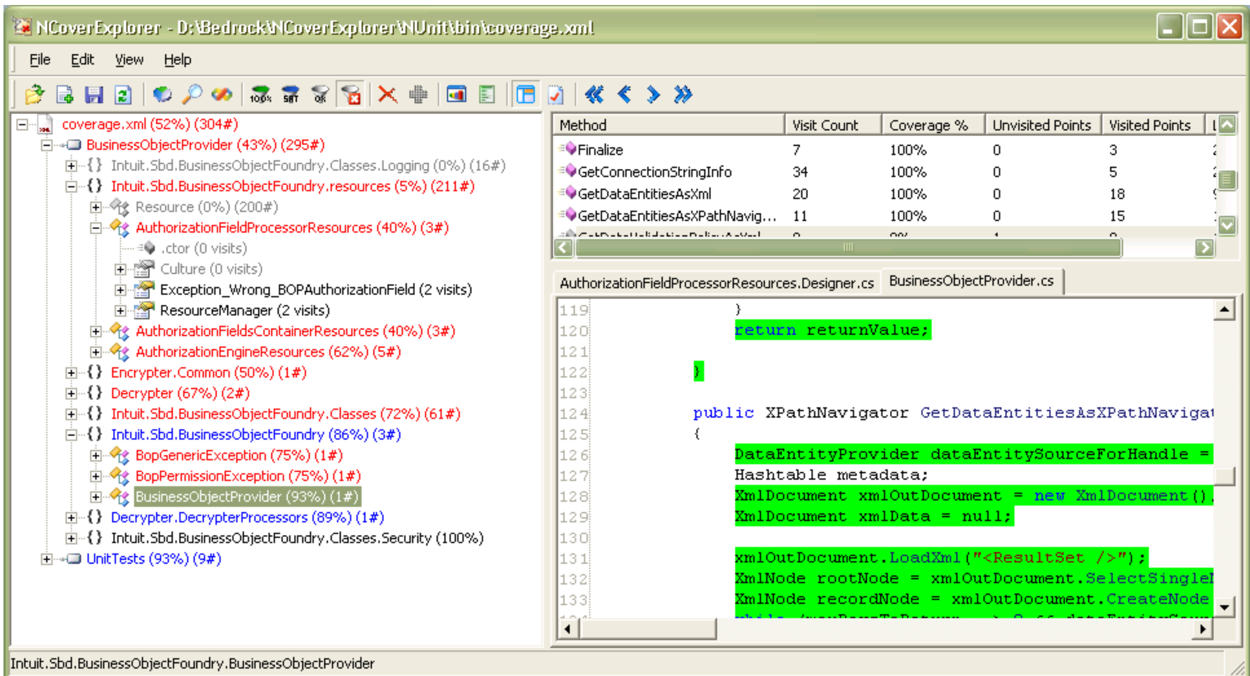

Figure 2. Figure showing number of sequence point visits .

Reports from NCoverExplorer can be delivered in different formats like reports based on sequence points or function level. A sequence point typically equates to a statement of code. The one above is the report generated based on function coverage.
Reports can also be generated based on sequence points like the one below.

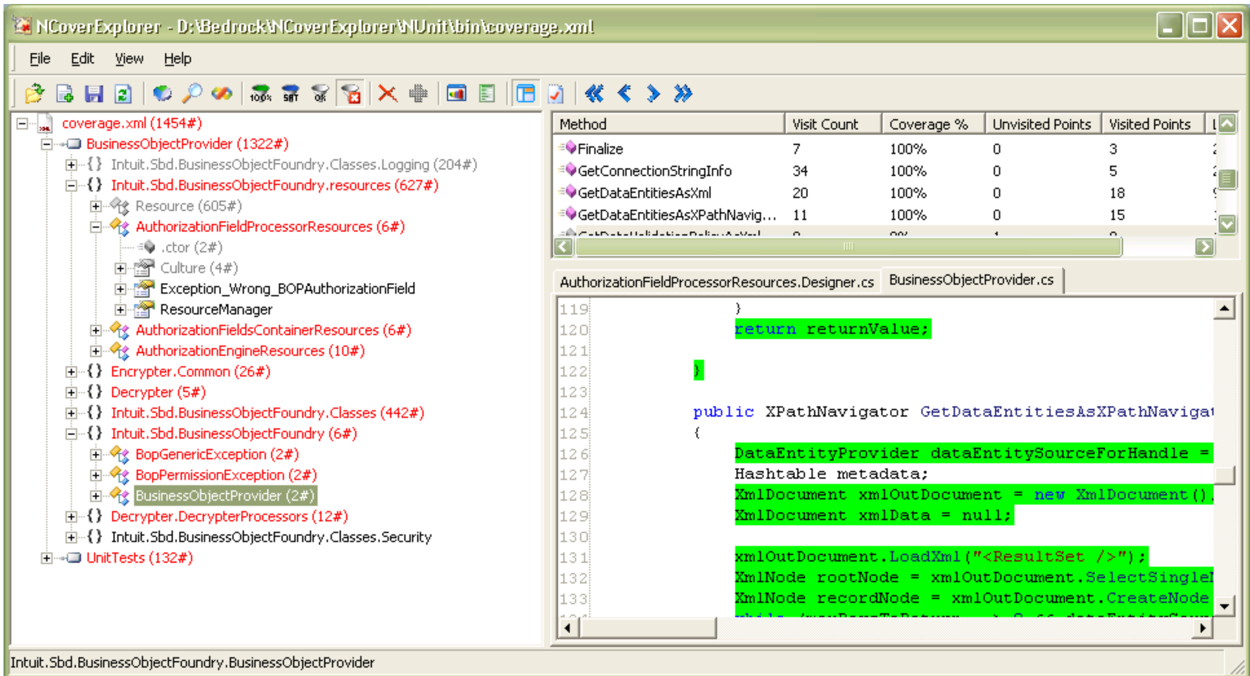

Figure 3. Figure showing number of function visits.

```
[TestFixture]
public partial class BOPUnitTestsCore : BOPUnitTestBase
{
   // Method to test StartSession
    [Test]
    public void TestStartSessionValid()
    {
            BusinessObjectProvider objBusinessObjectProvider = new BusinessObjectProvider();

            objBusinessObjectProvider.StartSession(m_correctConnectionString, m_qbUserId);
            objBusinessObjectProvider.EndSession();
            Assert.IsTrue(true);

    }

    // Method to test invalid case of StartSession
    [Test]
    public void TestStartSessionInvalid()
    {
        try
        {
            BusinessObjectProvider objBusinessObjectProvider = new BusinessObjectProvider();
            objBusinessObjectProvider.StartSession(m_incorrectConnectionString, m_qbUserId);
            objBusinessObjectProvider.EndSession();
            Assert.IsTrue(true);
        }
        catch (Exception)
        {
            BopGenericException bopEx = new BopGenericException();
            Assert.IsNotNull(bopEx);
        }
    }
```

Figure 4. Sample code after test execution.

As you can see in the code snippet above, code which was visited with our unit tests is marked green and the code which was not visited with our tests is marked red.

```
public class BOPUnitTestBase
{
    protected string m_correctConnectionString = "uid=apollo;pwd=rocks";
    protected string m_incorrectConnectionString = "uid=apollo1;pwd=rocks1";
    protected string m_qbUserId = "1";
    protected bool m_getDataOnlyWhenFullPermission = true;
                                    Code was visited 5 times
    // Method for throwing customexceptions
    public void ThrowCustomExceptions(ExceptionEnum ex)
    {
        ExceptionEnum myExceptionEnum = new ExceptionEnum();

        switch (myExceptionEnum)
        {
```

Figure 5. Number of times the code was covered.

NCoverExplorer would also let the user the number of times the code was covered with the tests. As you can see tells that the above part of code was covered five times.

Reports can be grouped on different parameters like namespaces, functions or modules.

NCoverExplorer Coverage Report - P55_India		Project Statistics:			
Report generated on:	Fri 22-Jun-2007 at 16:44:45	Files:	60	NCLOC:	3943
NCoverExplorer version:	1.3.6.32	Classes:	62		
Filtering / Sorting:	None / FunctionCoverageAscending	Functions:	636	Unvisited:	304
		Seq Pts:	3906	Unvisited:	1454

Project	Acceptable	Unvisited SeqPts	Coverage
P55_India	95.0 %	1454	62.8 %

Modules	Acceptable	Unvisited SeqPts	Coverage
BusinessObjectProvider.DLL	95.0 %	1322	52.2 %
UnitTests.DLL	95.0 %	132	88.5 %

Figure 6. Report based on module summary.

Project	Acceptable	Unvisited SeqPts	Coverage
P55_India	95.0 %	1454	62.8 %

Namespaces	Unvisited SeqPts	Coverage
Intuit.Sbd.BusinessObjectFoundry.Classes.Logging	204	0.0 %
Intuit.Sbd.BusinessObjectFoundry.resources	627	5.0 %
Encrypter.Common	26	25.7 %
Decrypter	5	84.4 %
Intuit.Sbd.BusinessObjectFoundry.Classes	442	71.1 %
Intuit.Sbd.Bop.UnitTestCommon	9	71.0 %
Intuit.Sbd.BusinessObjectFoundry	6	94.4 %
Decrypter.DecrypterProcessors	12	86.8 %
BOPUnitTests	123	88.9 %
Intuit.Sbd.BusinessObjectFoundry.Classes.Security	0	100.0 %

Figure 7. Report based on namespace summary.

Modules	Acceptable	Unvisited SeqPts	Coverage
BusinessObjectProvider.DLL	95.0 %	1322	**52.2** %
UnitTests.DLL	95.0 %	132	**88.5** %

Module	Acceptable	Unvisited SeqPts	Coverage
BusinessObjectProvider.DLL	95.0 %	1322	**52.2** %
Namespace / Classes			
Intuit.Sbd.BusinessObjectFoundry.Classes.Logging		204	**0.0** %
Logger		190	**0.0** %
LogSourceConverter		14	**0.0** %
Intuit.Sbd.BusinessObjectFoundry.resources		627	**5.0** %
Resource		605	**0.0** %
AuthorizationFieldProcessorResources		6	**53.8** %
AuthorizationFieldsContainerResources		6	**53.8** %
AuthorizationEngineResources		10	**65.5** %
Encrypter.Common		26	**25.7** %
AlgorithmFactory		26	**25.7** %
Decrypter		5	**84.4** %
DecrypterHelper		3	**78.6** %
PublicKeys		2	**88.9** %
Intuit.Sbd.BusinessObjectFoundry.Classes		442	**71.1** %
GdsIndexerQueries		26	**0.0** %
GdsQueryParameters		26	**0.0** %
NoOpProcessor		9	**0.0** %
XmlDBResultSet		28	**0.0** %
XmlQuerySource		56	**0.0** %
GdsXPathExpressionBuilder		7	**36.4** %
BusinessObjectProviderMisc		37	**17.8** %
GdsIndexerNamespaces		2	**66.7** %
SqlAnywhereQuery		40	**46.7** %
QueryNameComponents		8	**66.7** %
BOPResourceManager		2	**80.0** %
FilterValue		2	**71.4** %

Figure 8. Report based on module/Class summary.

4 Conclusion

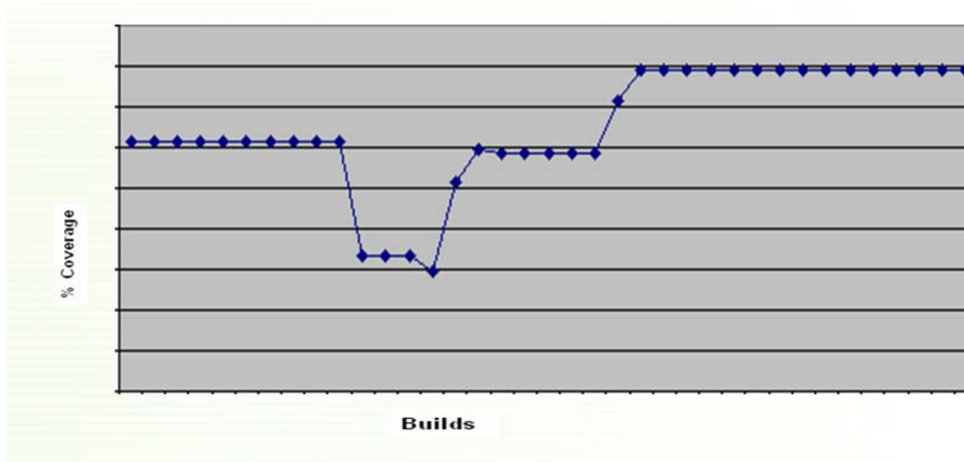

Figure 9. Coverage graph.

When we started off with coverage exercise, function coverage was at 60%. As stated previously, we set a quality goal of 80%. There was a drop in the percentage covered in the middle of the project when it hit 30%. Because of the new unit tests added, we are almost able to achieve our quality goal. Right now it stands at 79.2%. This also helped us reduce the number of system test defects. We are able to catch 80% of the defects prior to system test phase.

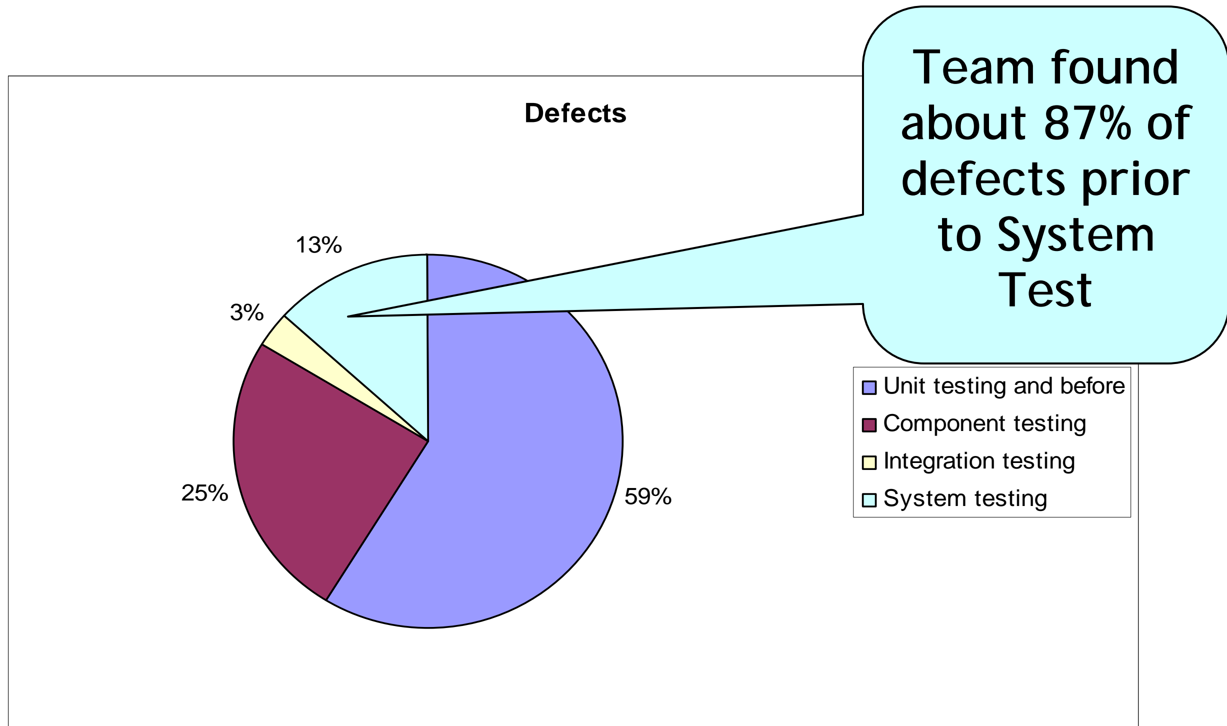

Figure 10. Defects found across different phases of test execution.

5 Reference Materials

5.1 Websites

The best source of up to date information about unit testing and tools is the Internet. Hundreds of sites exist, offering test tool downloads, tutorials, forums, articles, and examples. Several of the most prominent sites are described in the following list:

1. http://www.xprogramming.com
2. http://www.testdriven.com
3. http://www.extremeprogramming.org
4. http://www.nunit.org
5. http://www.oreilly.com
6. http://NCover.org
7. http://www.kiwidude.com/dotnet/NCoverExplorerFAQ.html

6 Biography

Biography of Balasubrahmanyam P

Balasubrahmanyam P has 6 yrs of professional experience in QA & Testing. He worked with reputed organizations like Symphony Services and Intel. His professional experience includes work on various domains that include instant messaging, Wireless Networking, Finance and Telecom Cost Management. Balasubrahmanyam is currently working with Intuit Technology Services Pvt Ltd as Senior Software QA Engineer.