Test for Success With BBj Unit Test

You finally finished writing your BBj® application, and it's time to put it out in front of your end users. Or maybe it's not – how do you know when software really is ready for public consumption? The answer may be "*When the scheduled release date arrives*," or "*When the boss says it's time*," or maybe even "*When we've put it through its paces and there are no more significant bugs to fix*." Sure, we all know that any good software development life cycle includes time for testing, starting with unit testing and ending with some form of system or acceptance testing. But how much time and money can you afford to invest in testing? How do you use your limited testing dollars to get the most bang for your buck?

Common sense says that the earlier you find a bug the easier and cheaper it is to fix it, and our experience at BASIS supports this conclusion. But this is only helpful if you don't have to spend large amounts of money or time in order to find those bugs early. So what you need is a relatively cheap tool to help you find bugs in your BBj code as early as possible. How about finding bugs as a code change creates them?

By Jerry Karasz
Software Architect

Sebastian Adams
Software Developer

While you are creating a new BBj application, wouldn't it be nice if you could just push a button and find out what works and what doesn't? What would you give to have a series of tests that you can run any time, over and over, quickly turning out a clear and concise report of which passed and which failed? Unit testing offers an opportunity for just such a return on even a small investment. But unit tests are not free – somebody has to write every one of them. The return comes once you have written your unit tests. You can run them over and over again, not only to find bugs as you are developing your code, but even afterwards to find out that you broke something in your code with that latest bug fix.

Unfortunately, unit testing has never before been an option for BBj developers – until now. BASIS is proud to announce the first step in providing a BBj Unit Test framework that you can use to develop and run unit tests for your BBj code – the BBj Unit Test Eclipse plug-in. For those of you familiar with CppUnit, JUnit, or any of the other xUnit frameworks, you will see a lot that you are familiar with here.

Preparing for Unit Testing

Although this plug-in is still in its infancy, it does provide the basis for a good unit testing strategy. Let's look at where to get it and how to use it.

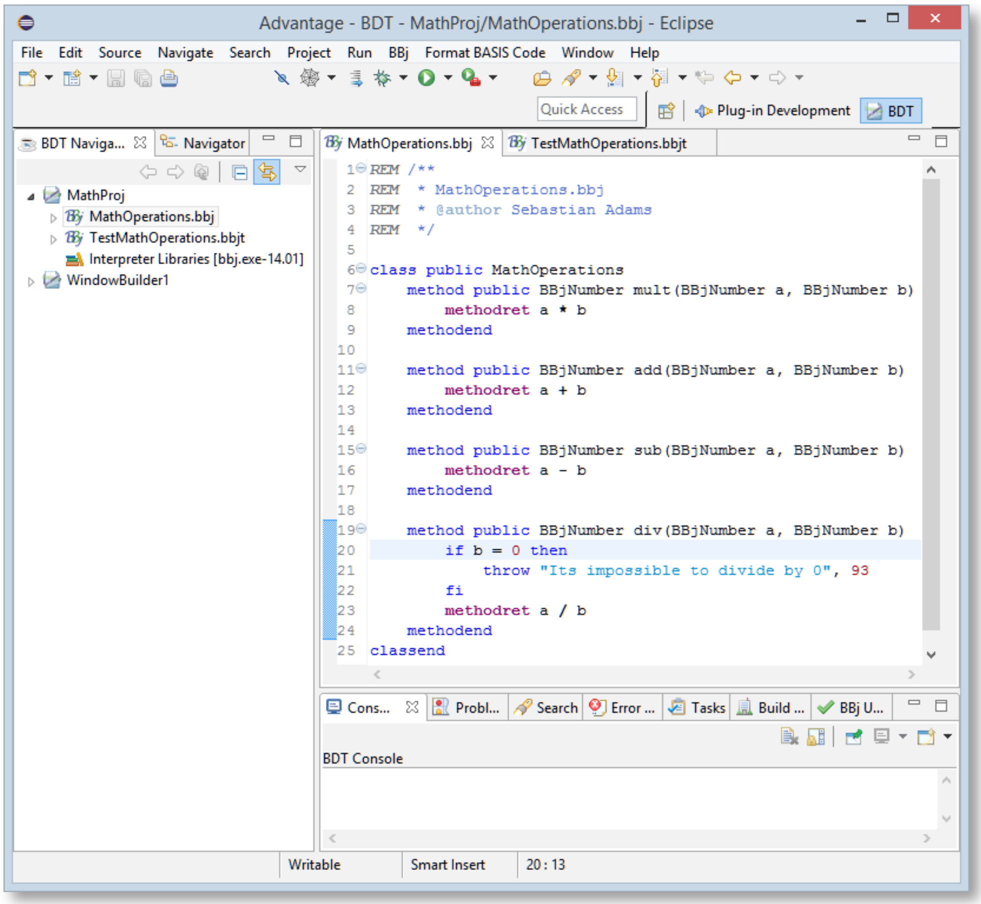
Getting the Plug-in

To install the BBj Unit Test plug-in into Eclipse, set Eclipse to use links.basis.com/bbutils as an available software site. Whenever you check for software updates after that, Eclipse will offer to download a newer version when one is available.

Creating Unit Test Code

Suppose you have a BBj project named **`MathProj`** that contains a BBj code file named **`MathOperations.bbj`** as shown in **Figure 1**.

MathOperations is a simplistic class and offers just enough methods to be useful but not enough to complicate this example. It has methods defined to do multiplication, addition, subtraction, and division. But does it do them correctly? To find out, let's set up some unit tests.



```
REM /**
REM  * MathOperations.bbj
REM  * @author Sebastian Adams
REM  */

class public MathOperations
    method public BBjNumber mult(BBjNumber a, BBjNumber b)
        methodret a * b
    methodend

    method public BBjNumber add(BBjNumber a, BBjNumber b)
        methodret a + b
    methodend

    method public BBjNumber sub(BBjNumber a, BBjNumber b)
        methodret a - b
    methodend

    method public BBjNumber div(BBjNumber a, BBjNumber b)
        if b = 0 then
            throw "Its impossible to divide by 0", 93
        fi
        methodret a / b
    methodend
classend
```

Figure 1. A 'BBj Project' that contains a simple mathematics class

What Exactly is a Unit Test?

In BBj Unit Test, unit test is a method on a class that returns nothing (it has a void return type) and announces its success or failure through the special Assert methods it calls (more on the Assert methods later). In order for the BBj Unit Test framework to recognize your method as an official "unit test" and be able to run it, your method also needs to be flagged as a unit test method by making the line of code immediately preceding the method declaration an annotation remark, **`REM @Test`** (case insensitive). For an example of a simple unit test method, see **Figure 2**.

Where do I put my Unit Test Code?

Unit test methods such as addTest() must be part of a unit test class. We could create a class directly in the **`MathOperations.bbj`** file to hold all of our unit test methods, but once MathOperations and our tests get large, this will become harder to manage, and running our tests in Eclipse will conflict with running the regular .bbj file. Besides, it is considered poor programming style. For these reasons, the BBj Unit Test plug-in requires test classes to be in a separate file in the same BBj project with a specific extension: .bbjt. We also recommend you name the test class and file something intuitive so that the relationship between the file being tested and the unit test file is obvious. For our example, let's name our unit test class **`TestMathOperations`** and our unit test file **`TestMathOperations.bbjt`**, and put **`TestMathOperations.bbjt`** in the same **`MathProj`** project (see how this looks in the BDT Navigator view shown in **Figure 1**). Starting the unit test class name and filename with "Test" is a convention that we recommend, but it is not required.

```
REM @Test
method public void addTest()
    Assert.Equals(#mathOperations!.add(8,8),16)
methodend
```

Figure 2. An example unit test method

How do I Structure my Unit Test Class?

In this example, we will probably need an instance of the MathOperations class in every unit test method we write, because that is what we are testing. We could allocate a MathOperations variable in every unit test method, but that seems redundant and a lot of work. We could put in a field that is accessible to every unit test method such as **`field private MathOperations mathOperations!`**, and then our class would look something like **Figure 3**.

```
use ::MathOperations.bbj::MathOperations

class public TestMathOperations
    field private MathOperations mathOperations!

    rem @Test
    method public void addTest()
        Assert.Equals(#mathOperations!.add(8,8),16)
    methodend
classend
```

Figure 3. A partial BBj Unit Test class, TestMathOperations

But that won't work – we need a way to initialize **field mathOperations!** to an instance before we can use it, and that needs to be done exactly once before we start running our tests.

If we just had some way to run some code before we executed any of our unit tests. BBj Unit Test gives us just that – the ability to define a "setup" method that it calls exactly once before any of our unit tests run using the **@BeforeClass** annotation remark. We can add a method to TestMathOperations that will do exactly that. Our new method named setup() is shown in **Figure 4**.

```
rem @BeforeClass
method public void setup()
    #mathOperations! = new MathOperations()
methodend
```

Figure 4. Setup done exactly once before any of our unit tests run

The **@BeforeClass** is the right place for any initialization including preparing globals like STBL values, opening file channels, or any other preparation your tests require.

But what if we have setup code that needs to be executed before each unit test is run? Try **@Before**. Or cleanup code that needs to be executed after each unit test is run? Try **@After**. It turns out that a need for "setup" and "cleanup" code is pretty common, so BBj Unit Test offers you two different times to run setup and cleanup. **Figure 5** lists the recognized annotation remarks and their purposes, which includes both setup and cleanup options.

Annotation	Purpose
@BeforeClass	Run this method once before running any **@Before** or **@Test** methods (class setup)
@Before	Run this method immediately before running each **@Test** method (method setup)
@Test	Run this method as an actual unit test
@After	Run this method immediately after running each **@Test**method (method cleanup)
@AfterClass	Run this method once after running all **@After** and **@Test** methods (class cleanup)
@Ignore	Do not run this method as a unit test (it shows as ignored in the test results)

Figure 5. Annotation remarks for unit testing

How do I Determine Success or Failure in my Unit Test?

But how do we actually determine success or failure in each **@Test** method? Currently, BBj Unit Test offers a number of methods to determine this (see **Figure 6**).

Method	Explanation
Assert.Equals(BBjNumber A, BBjNumber B)	Successful if the number **A** equals the number **B**; fails otherwise
Assert.Equals(BBjNumber A, BBjNumber B, BBjNumber Delta)	Successful if the number **A** equals the number **B** within **+/- Delta**; fails otherwise
Assert.Equals(BBjString A$, BBjString B$)	Successful if the string **A$** equals the string **B$**; fails otherwise
Assert.Equals(Object A!, Object B!)	Successful if the reference to object **A!** equals the reference to object **B!**; fails otherwise
Assert.NotEquals(BBjNumber A, BBjNumber B)	Successful if the number **A** does not equal the number **B**; fails otherwise
Assert.NotEquals(BBjNumber A, BBjNumber B,BBjNumber Delta)	Successful if the number **A** does not equal the number **B** within **+/- Delta**; fails otherwise
Assert.NotEquals(BBjString A$, BBjString B$)	Successful if the string **A$** does not equal the string **B$**; fails otherwise
Assert.IsNull(Object obj!)	Successful if the object **obj!** is null; fails otherwise
Assert.IsNotNull(Object obj!)	Successful if the object **obj!** is not null; fails otherwise
Assert.Fail()	Always fails
Assert.Expect(BBjNumber exceptionA, BBjNumber exceptionB)	Successful if the exception number **exceptionA** thrown equals the exception number **exceptionB**; fails otherwise

Figure 6. Success and failure methods

With the exception of the Expect() method, all of the methods listed in **Figure 6** also offer a second method signature that is identical to the one shown but that takes a **BBjString errorMessage$** as the first argument. The **errorMessage$** allows you to specify a particular failure message to report if the invocation fails.

For the sake of simplicity, we will focus on the two simplest functions:
`Assert.Equals(BBjNumber, BBjNumber)`, and
`Assert.Expect(BBjNumber, BBjNumber)`.

`Assert.Equals(#mathOperations!.add(8,8),16)` shows how to test the MathOperations.add() operation when we know the BBjNumber value we will get if the code is successful. In this case, we know that if we add 8 to 8 we will get 16, and our unit test will report success. If, however, MathOperations.add() returns any value other than 16, the unit test will fail.

`Assert.Expect(#mathOperations!.div(20,0),93)` shows how to test the MathOperations.div() operation under a condition when we expect it to throw a particular exception (93 in our example). In this case, we know what should happen if we try to divide any number by 0. The code should throw an exception 93, and our unit test should report success (because we got the expected exception). If, however, MathOperations.div() returns any value without throwing an exception 93 or throws any exception other than 93, the unit test will fail (because we expected an exception 93 but did not get it).

Putting All of the Pieces Together
Let's go ahead and define a full `@Test` method for each of MathOperations' methods: addTest(), multTest(), subTest(), and divTest(). Our unit test class now looks something like **Figure 7**.

You are not limited to testing only class operations like mathOperations!.div(). You can also invoke legacy functions like CALL as shown in **Figure 8**.

```
use ::MathOperations.bbj::MathOperations

class public TestMathOperations
    field private MathOperations mathOperations!

    rem @BeforeClass
    method public void setup()
        #mathOperations! = new MathOperations()
    methodend

    rem @Test
    method public void addTest()
        Assert.Equals(#mathOperations!.add(8,8),16)
        Assert.Equals(#mathOperations!.add(-9,8),-1)
        Assert.Equals(#mathOperations!.add(0,10),10)
        Assert.Equals(#mathOperations!.add(-9,-8),-17)
    methodend

    rem @Test
    method public void multTest()
        Assert.Equals(#mathOperations!.mult(8,8),64)
        Assert.Equals(#mathOperations!.mult(0,8),0)
        Assert.Equals(#mathOperations!.mult(-6,8),-48)
        Assert.Equals(#mathOperations!.mult(-6,-6),36)
    methodend

    rem @Test
    method public void subTest()
        Assert.Equals(#mathOperations!.sub(50,25),25)
        Assert.Equals(#mathOperations!.sub(5,25),-20)
        Assert.Equals(#mathOperations!.sub(-15,-15),0)
        Assert.Equals(#mathOperations!.sub(-15,0),-15)
    methodend

    rem @Test
    method public void divTest()
        Assert.Equals(#mathOperations!.div(50,2),25)
        Assert.Equals(#mathOperations!.div(20,-2),-10)
        Assert.Expect(#mathOperations!.div(20,0),93)
    methodend
classend
```

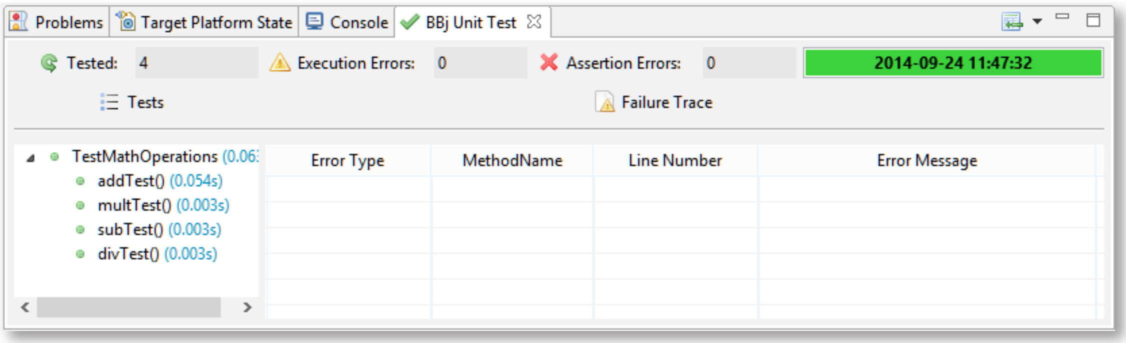
Figure 7. A complete example BBj Unit Test class, TestMathOperations

```
rem @Test
method public void callTest()
    CALL "myAddCall.bbj",5,7,x
    Assert.Equals(x,12)
methodend
```

Figure 8. An example of testing legacy code

Running Unit Tests

Now that we have defined our tests, it's time to run them. There are a number of ways to do this, but to keep it simple we'll run our tests from inside of the editor window. With the TestMathOperations class open in a code editor window, right-click anywhere in the code and select Run As > BBj Unit Test. Our tests all run, and the BBj Unit Test View appears as shown in **Figure 9**.



Tested: 4 Execution Errors: 0 Assertion Errors: 0 2014-09-24 11:47:32

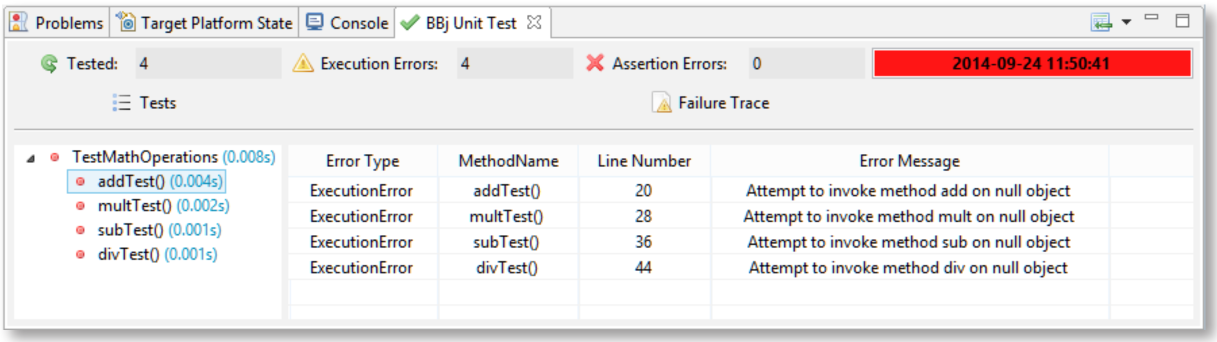
- TestMathOperations (0.06
  - addTest() (0.054s)
  - multTest() (0.003s)
  - subTest() (0.003s)
  - divTest() (0.003s)

Error Type	MethodName	Line Number	Error Message

Figure 9. The results of successfully running TestMathOperations' unit tests

If we wrote all of the TestMathOperations unit test code correctly, and we wrote all of the MathOperations methods correctly, then BBj Unit Test reports success and we get the nice green result display shown in **Figure 9**.

If, however, we made any coding errors in any of our unit test methods, we get a red result display with an 'Execution Error'. The view provides additional information to help us fix our mistake: the name of the offending unit test method, the line in that method that failed, and the matching error message as shown in **Figure 10**.



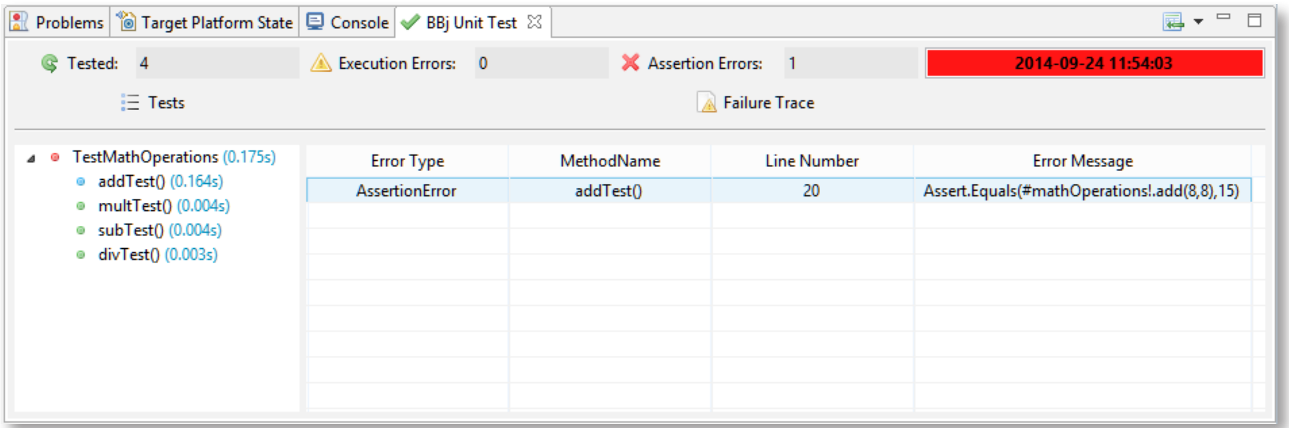
Tested: 4 Execution Errors: 4 Assertion Errors: 0 2014-09-24 11:50:41

- TestMathOperations (0.008s)
  - addTest() (0.004s)
  - multTest() (0.002s)
  - subTest() (0.001s)
  - divTest() (0.001s)

Error Type	MethodName	Line Number	Error Message
ExecutionError	addTest()	20	Attempt to invoke method add on null object
ExecutionError	multTest()	28	Attempt to invoke method mult on null object
ExecutionError	subTest()	36	Attempt to invoke method sub on null object
ExecutionError	divTest()	44	Attempt to invoke method div on null object

Figure 10. Execution errors in our unit test methods

If we wrote all of the TestMathOperations unit test code correctly but one MathOperations method works incorrectly, then BBj Unit Test will report a failure for that test and we get a red result display. This error will show as an 'Assertion Error' indicating the name of the offending unit test, the assertion in that method that failed, and an error message to help us see what went wrong as shown in **Figure 11**.



Tested: 4 Execution Errors: 0 Assertion Errors: 1 2014-09-24 11:54:03

- TestMathOperations (0.175s)
  - addTest() (0.164s)
  - multTest() (0.004s)
  - subTest() (0.004s)
  - divTest() (0.003s)

Error Type	MethodName	Line Number	Error Message
AssertionError	addTest()	20	Assert.Equals(#mathOperations!.add(8,8),15)

Figure 11. An assertion error in addTest()

If we made any mistakes, we now have to fix our problems and test again, and again ... and again. Remember, the goal of running our unit tests is to (eventually) see green, to have all of our unit tests run successfully. If we have written our unit tests to cover all of the necessary functionality, then success will mean just that – the code we tested (the MathOperations class) works the way we want it to. And we are ready to move on to our next coding task.

Summary

Unit testing is very valuable in determining whether or not a unit (such as a class or a function) works the way we want it to, to tell you when you are (finally) finished. One thing you may not have realized, though, is that the set of automated tests you just created can now be run again any time you modify your code or you prepare to release it. The industry calls this process Regression Testing and it can help you find any side effects or problems you may have inadvertently created before a release. The value of unit testing is not only in helping you know when you are finished developing, but it is also in helping you after that to know that everything is still working perfectly!

The BBj Unit Test Eclipse plug-in is free to use with the BDT Eclipse plug-ins and is built to be extensible, but it is a framework that is only in its infancy. It offers a number of useful methods to help you to unit test your programs, but there is still a great deal more functionality that we can add if you would find it helpful. So give it a try and send us your feedback on what you would like to see added next to BBj Unit Test. Help us to help you turn out better Business BASIC programs for your customers. ■

- Try out the BBj Unit Test plug-in today by installing BBj Utilities
- Download and run the code samples