|  |  |
| --- | --- |
|  | Code Testing Guide  Version 2.0.Beta -April 2011 |

Abstract

This document presents guidelines for testing code as an integral part of the process of developing code for Microsoft Biology Foundation (MBF). MBF uses VSTest, part of the Visual Studio Team System suite, as the primary means for ensuring code quality. As you build the units of your application, you must create a set of validation tests to validate your components.

This paper is intended for developers who contribute to the MBF open source project.

For updates to this document and the rest of the project documentation, see   
<http://mbf.codeplex.com/documentation>

For updates to the project, see <http://mbf.codeplex.com>

Contents

[Introduction to the Testing Process 4](#_Toc289327092)

[The project testing process 4](#_Toc289327093)

[Dependent software products 4](#_Toc289327094)

[Use VSTest for Tests 5](#_Toc289327095)

[Write unit tests for the new code 7](#_Toc289327096)

[Unit test case generation 7](#_Toc289327097)

[Automation test case generation 8](#_Toc289327098)

[Run the test 9](#_Toc289327099)

[To run the tests 9](#_Toc289327100)

[Prepare a Shelveset for code review and submission 10](#_Toc289327101)

[Appendix A - Project.TestAutomation Guide 10](#_Toc289327102)

[Setup 12](#_Toc289327103)

[Bio.Net Object Model 12](#_Toc289327104)

[Encoding & Translation test cases 12](#_Toc289327105)

[Parsers and Formatters which support different file formats 12](#_Toc289327106)

[PAMSAM Assembly Algorithms 13](#_Toc289327107)

[Registration 13](#_Toc289327108)

[Bio Matrix 13](#_Toc289327109)

[Alignment and Assembly Algorithms 14](#_Toc289327110)

[Web-Service Clients for executing Blast queries 14](#_Toc289327111)

[Dependencies 14](#_Toc289327112)

[Test cases Automated 14](#_Toc289327113)

[Overview of Test cases Automated 15](#_Toc289327114)

[Running the Automation 15](#_Toc289327115)

[Validate the Pass/Fail status of Test cases 16](#_Toc289327116)

[Update the Sequences and Files in the xml 16](#_Toc289327117)

[Validate the Code Coverage 17](#_Toc289327118)

Disclaimer: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, Silverlight, Visual Studio, and Windows are trademarks of the Microsoft group of companies.  All other trademarks are property of their respective owners.

# Introduction to the Testing Process

An integral part of writing and contributing code is testing code. This document presents guidelines for testing code for the Microsoft Biology Foundation (MBF).

In order to write, compile and execute the test components, you will need to use the test suites provided by Visual Studio 2010 which the project framework is dependent on to maintain quality. When you create and contribute code, you must create new VS Unit tests for your code contribution, either as unit test or automation. All code contributions to the project require unit testing with the VS Unit tests. A VS Unit test is basically a method appropriately attributed to indicate that it should be considered a test.

This document describes how to prepare unit tests for new code developed for the Microsoft Biology Foundation with a focus on VSTest, part of the Visual Studio Team System suite which includes a full test management system in TFS.

For more information on Visual Studio 2010 testing see [Testing the Application](http://msdn.microsoft.com/en-us/library/ms182409.aspx) on MSDN. Though this document specifically addresses testing code using Visual Studio, the general principles apply whatever Integrated Development Environment you use.

## The project testing process

This document describes the process for you to follow when contributing code to the project. Run unit test frequently, for all changes. Run the automation suite provided and new automation tests you have written for any new features you have added. The process consists of the following steps:

[Use VSTest for Tests](#_Toc287878080)

[Write unit tests for the new code](#_Toc287878081)

[Run the test](#_Toc287878082)

[Prepare a Shelveset for code review and submission](#_Toc287878083)

Note:If you are a professional employee, please check with your employer for your company policy for submitting code to any open source code project**.**

## Dependent software products

To compile the project, Microsoft Visual Studio® 2010 or later is required.

In order to write, compile and execute the test components, you must install Microsoft Visual Studio (VS2010) and use the VSTest suites.

For academic researchers. You can receive free Microsoft software to assist in working with the project. To download and install the latest version of Microsoft Visual Studio (VS2010), visit the Microsoft [DreamSpark](https://www.dreamspark.com/) web site at <https://www.dreamspark.com>.

**Note:** When developing code for the project, follow the guidelines in the following documents:

|  |  |
| --- | --- |
| Document | Description |
| C# Coding Conventions | Presents guidelines for developing code for the project. |
| How to Write Code Comments in Source Code | Discusses how to write good documentation comments, specifically for projects that will produce an API reference directly from code comments. |
| Contributors Template | Provides a template for a User/Programming Guide for project contributions. |

All project documentation can be found at <http://mbf.codeplex.com/documentation>

# Use VSTest for Tests

MBI uses VSTest as the primary means for ensuring code quality, which makes it easy to develop, integrate, and run test code against your contributions.

Visual Studio Test System (VSTS) includes a full suite of functionality for Visual Studio Team Test (TT). Team Test is a Visual Studio integrated unit-testing framework that enables:

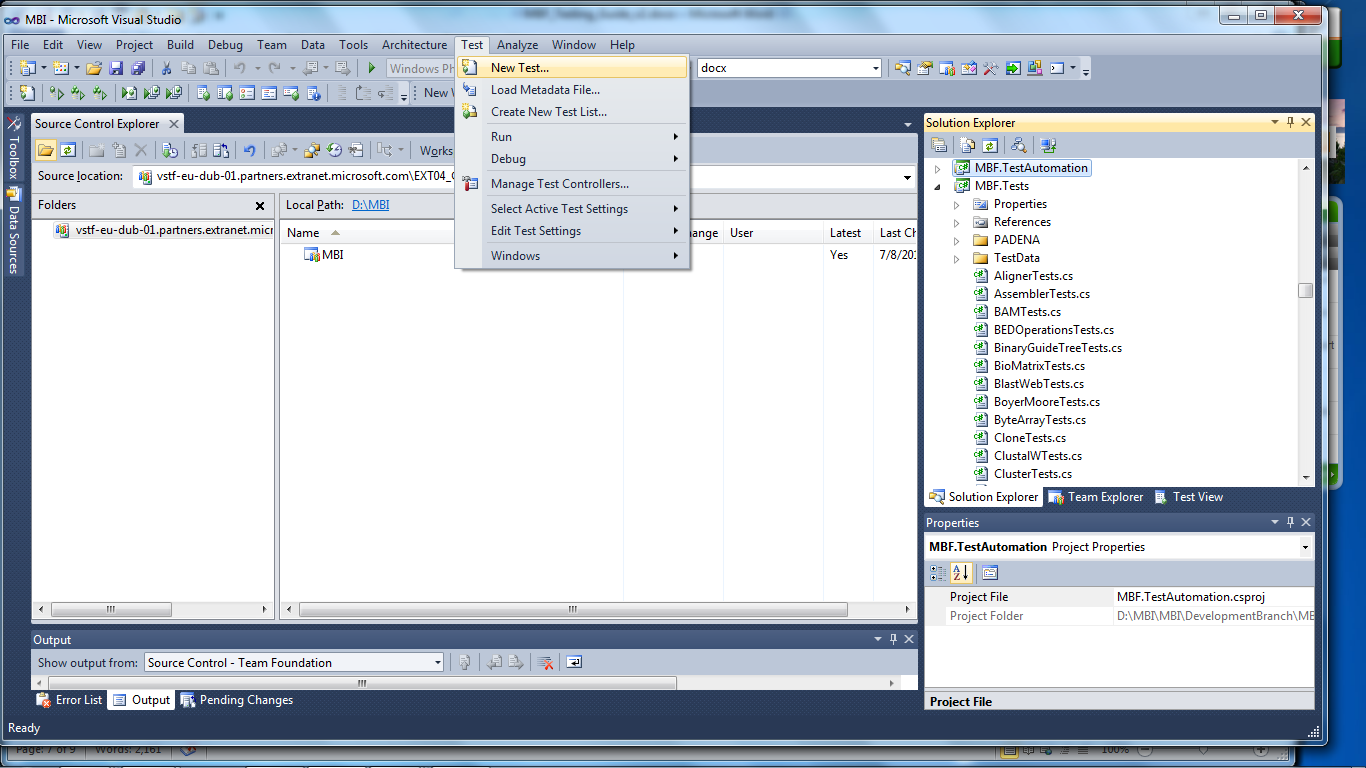
* Code generation of test method stubs.
* Running tests within the IDE.
* Incorporation of test data loaded from a database.
* Code coverage analysis once the tests have run.

These testing tools are integrated with Team Foundation Server, which lets you define your testing based on the same team projects used by others on the project.

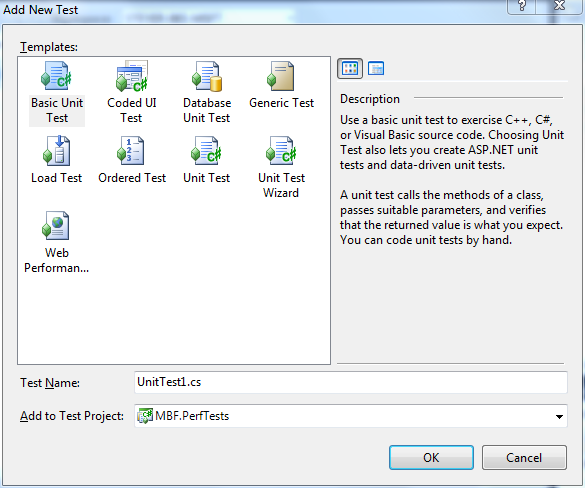
Both Microsoft Visual Studio 2010 Ultimate and Visual Studio Test Professional 2010 include the application Microsoft Test Manager to help you define and manage your testing effort by using test plans. You create a test plan and add any test suites, test cases, or configurations that you need.

To create a test or test plan

1. Click on the **Test** tab in Visual Studio and select **New Test**.



1. Select the test you want in the **Add New Test** window and click **OK**.



1. Create your test.

For more information on using Visual Studio 2010 test capabilities see [Testing the Application](http://msdn.microsoft.com/en-us/library/ms182409.aspx) on MSDN.

The MBI source tree includes VSTest tests that support both unit and automated tests that exercise each interface, function, and feature. The test code can be found in the Bio.Tests and Bio.TestsAutomation projects.

Important: If you develop any new code, or extend any existing code, you must implement a corresponding VSTest test and include it when you submit the changes. This ensures that subsequent development does not inadvertently cause a breaking change to another part of the code base. It also informs others about the intent of a particular piece of code—thus helping people who are new to the project to come up to speed on areas of interest.

# Write unit tests for the new code

The project uses VSTest as the primary means for ensuring code quality, which makes it easy to develop, integrate, and run test code against your contributions.

All code contributions to the project require unit testing with VSTest. You must create a new VS Test for your code contribution, either as unit test or automation.

The project source tree includes VSTest tests that support both unit and automated tests that exercise each interface, function, and feature. The test code can be found in the Bio.Tests and Bio.TestsAutomation projects.

Important: If you develop any new code, or extend any existing code, you must implement a corresponding VSTest test and include it when you submit the changes. This ensures that subsequent development does not inadvertently cause a breaking change to another part of the code base. It also informs others about the intent of a particular piece of code—thus helping people who are new to the project to come up to speed on areas of interest.

After passing unit tests, submit your code for code review as described in the “Committer\_Guide**”** document.

## Unit test case generation

Unit tests verify that the code module is behaving as expected. Including your unit tests with your new code provides a readily available way for your code to be tested and makes it easy for the reviewer to check whether a piece of code is working properly and whether or not the tests are sufficiently robust.

Unit tests are written to define the functionality before the code is written. Only when all tests pass is the code considered complete.

The project uses the following test namespace:

namespace Bio.Test | Bio.Tests.dll

Here are some recommendations for creating new unit tests:

* Identify all the public APIs—methods, constructor, properties, or others—available in the framework.
* Create at least a “happy path” test case for each public API and property.
* Write test cases to cover a specific scenario, where one or more public APIs are involved.
* Cover as many scenarios as possible.
* Make sure that every line of code is hit to get the maximum code coverage.
* Automate the test cases based on the priority— for example, BVTs, Priority 1 and Priority 2.

For example, the FastA parser has many different override public APIs for parsing. The project includes many individual public API test cases, such as TestFastaWhenParsingOneOfMany().

An example scenario would be to pass an input FastA file to the parser and get the output sequence object. Then, pass those output sequence objects into FastA formatter, where it produces an output file, which is exactly similar to the original file.

Minimal negative cases are written as part of unit test cases. For example, BadContent() is a negative FastA parser test case, to check bad content in FastA file format.

## Automation test case generation

Test automation essentially is the use of software to control the execution of tests and the comparison of actual outcomes to expected outcomes. Test automation involves automating a manual process already in place that uses a formalized testing process. Once tests have been automated, they can be run quickly and repeatedly.

What, when and whether or not to automate are crucial decisions for the testing and development team. Selecting the correct features of the product for automation largely determines the success of the automation. Automating unstable features or features that are undergoing changes should be avoided.

The project uses the following test automation namespace:

namespace Bio.TestAutomation | Bio.TestAutomation.dll

* Identify a work item/feature for automation.
* Identify all the public APIs—methods, constructor, properties, and others—available in that feature.
* Come up with Positive, Negative and Boundary test cases and document them in the Product Studio/Excel work sheet.
* Identify the priority for all the test cases—for example, BVT, Priority 1 and Priority 2.

As an example, consider the FastA Parser API, which uses a Parse method exposed as public. We would test these scenarios for that API as follows:

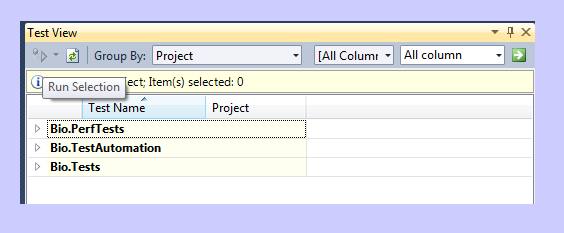
1. Parse FastA file that has one line sequence
2. Parse FastA file of size less than 35 K (small size)
3. Parse FastA file of size more than 35 K and less than 100 K (medium size)
4. Parse FastA file of size more than 100 K and less than 350 K (large size)
5. Parse FastA file of size more than 350 K (very large size)
6. Parse FastA file that contains only DNA sequence
7. Parse FastA file that contains only RNA sequence
8. Parse FastA file that contains only Protein sequence
9. Parse FastA file that contains DNA and RNA sequence
10. Parse FastA file that contains DNA and Protein sequence
11. Parse FastA file that contains Protein and RNA sequence
12. Parse FastA file that contains DNA, RNA and Protein sequence
13. Parse FastA file that contains Ambiguous DNA sequence
14. Parse FastA file that contains Ambiguous RNA sequence
15. Parse FastA file that contains Ambiguous Protein sequence
16. Parse FastA file with valid file path
17. Parse FastA file with invalid file path
18. Parse FastA file with space in file path
19. Parse FastA file with Unicode characters in file path
20. Parse FastA file with Unicode characters in the FastA file
21. Parse FastA file with file in network location
22. Parse GenBank file instead of FastA
23. Parse GFF file instead of FastA
24. Parse SAM file instead of FastA
25. Parse FastA file that is empty
26. Parse FastA file that is too big for the parser to handle

# Run the test

When your code is ready and your tests are written run the tests to verify your code produces the expected results. All test cases must pass before you submit your code.

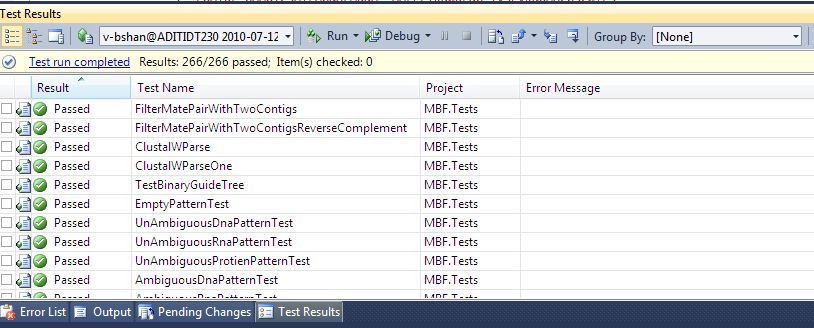
## To run the tests

1. Click **Windows** on the **Test** menu and open the **Test View** window.

2. In the **Test View** window select Bio.Tests and click **Run Selection**—the green arrow in the upper left corner—as shown in the following figure.

Test View window

All test cases should pass and VS Test should display a Test Results window that looks similar to the following screen shot.



Test Results window

# Prepare a Shelveset for code review and submission

Once all tests have passed prepare your Shelveset. Use the Shelve command in TFS to store your pending changes—together with pending check-in notes, a comment, and a list of associated work items such as the accompanying unit tests—on the Team Foundation Server without actually checking them into the source control server.

The output of the Shelve command is known as a shelveset.

You can give the shelveset a descriptive name and update your CodePlex or TFS work item with the name of the shelveset.

For more information, see [Shelve Command in the VS Team System online help](http://msdn.microsoft.com/en-us/library/w6y8ezzs(VS.80).aspx).

**Note**: Be sure when creating a Shelve set that you do not accidentally check in the changes. The Shelveset can be named, and together with your user id, a reviewer can connect to the TFS server and find your shelveset for review. There is no need to submit code nor email any files for review.

# Appendix A - Project.TestAutomation Guide

This appendix describes the project’s test automation.

Contents

Appendix A - Project.TestAutomation Guide 10

Setup 12

Automation Code Path 12

Test Automation Code Structure 12

Bio.Net Object Model 12

Sequence Test cases 12

Encoding & Translation test cases 12

Translation Test cases 12

Parsers and Formatters which support different file formats 12

FastA Test cases 12

GenBank Test cases 12

Gff Test cases 12

FastQ Test cases 13

Phylogenetic Test cases 13

Snp Test cases 13

Bed Test cases 13

SAM Test cases 13

BAM Test cases 13

ClustalW parser Test cases 13

Nexus parser Test cases 13

Phylip parser Test cases 13

PAMSAM Assembly Algorithms 13

PAMSAM Test cases 13

Registration 14

Registration Test cases 14

Bio Matrix 14

Matrix Test cases 14

Alignment and Assembly Algorithms 14

Alignment Test cases 14

Assembly Test cases 14

Web-Service Clients for executing Blast queries 14

NCBI Blast Web Service Test cases 14

EBI Blast Web Service Test cases 14

Azure Blast Web Service Test cases 14

BioHPC Web Service Test cases 14

ClustalW Web Service Interface Implementation Test cases 15

Dependencies 15

Test cases Automated 15

Overview of Test cases Automated 15

Running the Automation 15

Pre-condition for Microsoft/External Internal 16

Running in Visual Studio 16

Running at Command Prompt 16

Validate the Pass/Fail status of Test cases 16

Update the Sequences and Files in the xml 17

Validate the Code Coverage 17

Pre-requisite for running the Code Coverage tool Magellan 17

Steps for Running Code Coverage in Visual Studio 17

## Setup

### Automation Code Path

Automation Code for the project core components is present under the path *”$\Bio\Tests\Bio.TestAutomation”.*

### Test Automation Code Structure

Automation code is structured on the same lines as the project development code mentioned below. The paths where the test automation suite is located are included.

## Bio.Net Object Model

### Sequence Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation”*. This folder contains all the BVT, P1 and P2 test cases for Derived, Segmented, Sparse, Sequence Range, Virtual, Virtual Data, Qualitative, Compound and Sequences test cases.

## Encoding & Translation test cases

### Translation Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Algorithms\Translation”*. This folder contains all the BVT and P1 test cases for Translation for example, Codons, Protein translation, Complementation and Transcription.

## Parsers and Formatters which support different file formats

### FastA Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO\Fasta”*. This folder contains all the BVT and P1 test cases for FastA Parsers & Formatters.

### GenBank Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO\Genbank”*. This folder contains all the BVT and P1 test cases for GenBank Parsers & Formatters.

### Gff Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO\GFF”*. This folder contains all the BVT and P1 test cases for Gff Parsers & Formatters.

### FastQ Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO\FastQ”*. This folder contains all the BVT and P1 test cases for FastQ Parsers & Formatters.

### Phylogenetic Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO\Newick”*. This folder contains all the BVT and P1 test cases for Newick Parsers & Formatters.

### Snp Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for Snp Parsers.

### Bed Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for Bed Parsers & Formatters.

### SAM Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for Sam Parsers.

### BAM Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for Bam Parsers & Formatters.

### ClustalW parser Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for ClustalW Parsers.

### Nexus parser Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for Nexus Parsers.

### Phylip parser Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\IO”*. This folder contains all the BVT and P1 test cases for Phylip Parsers.

## PAMSAM Assembly Algorithms

### PAMSAM Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\PAMSAM”*. This folder contains all the BVT and P1 test cases for PAMSAM assembly algorithm.

## Registration

### Registration Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Registration”*. This folder contains all the BVT and P1 test cases for Bio.Registration.

## Bio Matrix

### Matrix Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Matrix”*. This folder contains all the BVT test cases for Bio.Matrix.

## Alignment and Assembly Algorithms

### Alignment Test cases

Present under location *“$\Bio\Tests\Bio.TestAutomation\Algorithms\Alignment”*. This folder contains all the BVT, P1 and P2 test cases for Alignment i.e., NeedlemanWunsch algorithm, SmithWaterman algorithm, PairwiseOverlap algorithm, MUMmer algorithm, NUCmer algorithm and Sequence alignment.

### Assembly Test cases

Present under location *“$\Bio\Tests\Bio.TestAutomation\Algorithms\Assembly”*. This folder contains all the BVT and P1 test cases for Assemblies i.e., Simple sequence assembler, Padena and Simple consensus method.

## Web-Service Clients for executing Blast queries

### NCBI Blast Web Service Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Web\Blast”*. This folder contains all the BVT and P1 test cases for NCBI Blast web service test cases.

### EBI Blast Web Service Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Web\EbiBlast”*. This folder contains all the BVT and P1 test cases for EBI Blast web service test cases.

### Azure Blast Web Service Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Web\AzureBlast”*. This folder contains all the BVT and P1 test cases for Azure Blast web service test cases.

### BioHPC Web Service Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Web\BioHPC”*. This folder contains all the BVT and P1 test cases for BioHPC web service test cases.

### ClustalW Web Service Interface Implementation Test cases

Tests are located at “$\*Bio\Tests\Bio.TestAutomation\Web\ClustalW”*. This folder contains all the BVT and P1 test cases for ClustalW web service interface implementation test cases.

## Dependencies

The binaries on which the automation is dependent are

* “$\Bio\Build\Binaries\Debug\Bio.dll”
* .Net 4.0

## Test cases Automated

3388 test cases are totally automated, the spread sheet embedded below shows the test cases which are automated are marked as “Automated” as per the PS query below. This 3388 also includes the Unit test cases which are already automated as part of Unit testing.



## Overview of Test cases Automated

The below matrix shows the number of test cases automated out of 6223 test cases which was identified as part of M2, M3, M4, M5, Beta 1, M6, M7, Beta 2, M8, M9 and V1.

|  |  |  |
| --- | --- | --- |
| 1. **Priority** | 1. **Number of Test cases** | 1. **PS Query** |
| 1. BVT | 1. 1044 |  |
| 1. P1 | 1. 1667 |  |
| 1. P2 | 1. 677 |  |

## Running the Automation

Observe the following pre-conditions and Steps to run the automation.

### Pre-condition for Microsoft/External Internal

.NET Framework 4.0 must be installed on the machine where the automation is run.

### Running in Visual Studio

1. Launch Visual Studio 2010
2. Build Bio.TestAutomation.proj
3. In **Test View**, window select the test cases which needs to be run
4. Click on the **Run Selection** button in the **Test View** window

### Running at Command Prompt

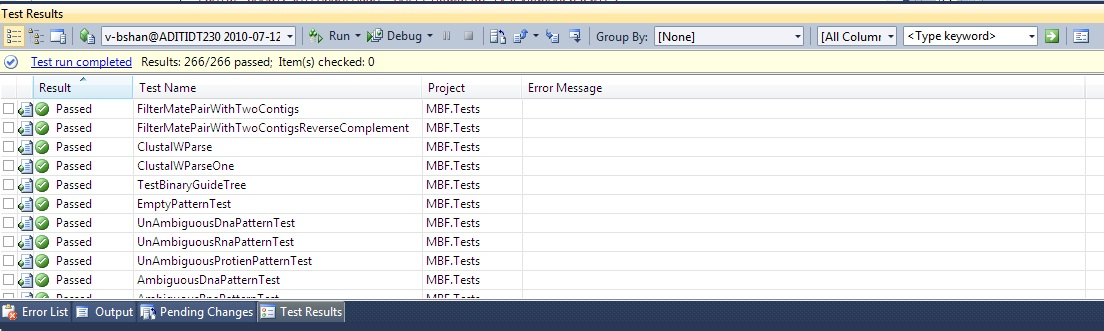
1. Launch Visual Studio command prompt
2. Run command “**MSTest/testcontainer:**Bio.TestAutomation.dll” (more command line options can be viewed at <http://msdn.microsoft.com/en-us/library/ms182489.aspx>)

## Validate the Pass/Fail status of Test cases

On **Running the Automation**, the **Test Results** window in the Visual Studio 2010 would display:

* Total Number of Test cases executed,
* Total test cases which was run successfully,
* Total test cases failed, Total test cases ignored,
* Total test cases aborted
* The time taken to run all the test cases, as shown in the following screen shot.

The following screen shot shows the time taken to run all the test cases.



A transaction file (.trx) will be created for every run and can be found under *“$\Bio\TestResults”* which would provide the log information. Old logs can also be viewed by just importing the .trx file in the Test Results window.

Also additional log information can be found in the location *“$\Bio\TestResults\Out\Bio.log”.*

Ideally the **Results** should say **Test run Completed** with all test cases passed as in the screen shot above should show as 266/266, which means 266 out of 266 passed.

If any test cases failed, Double click on the test case and that would say the details of failure.

## Update the Sequences and Files in the xml

The Config file for running the automation is located at “$\Bio\Tests\Bio.TestAutomation\TestData\TestUtils\TestsConfig.xml”.

The following example shows one of the nodes used to run one of the BVT test cases, where we update the information of FastA file used for in the test case, the expected sequence and other details related to the Test case which is required for validation of a specific sequence present in the FastA file.

<!-- This node contains the Fasta file information for running the BVT test cases. -->

<SimpleFasta>

<FilePath>TestUtils\Simple\_Fasta\_Protein.fasta</FilePath>

<ExpectedSequence>IFYEPVEILGYDNKSSLVL</ExpectedSequence>

<ExpectedSequenceCount>435</ExpectedSequenceCount>

<AlphabetName>Protein</AlphabetName>

<SequenceID>gi|186972391|gb|ACC99454.1| maturase K [Scaphosepalum rapax]</SequenceID>

<FormatString>>gi|186972391|gb|ACC99454.1| maturase K [Scaphosepalum rapax]IFYEPVEILGYDNKSSL</FormatString>

<ExpectedSequenceAfterAdd>IFYEPVEILGYDNKSSLVL</ExpectedSequenceAfterAdd>

</SimpleFasta>

This file can be updated to change the sequence or the file from which the Sequence is to be read and accordingly other values in the node needs to be updated for validation and the file should be updated added if any in the location “$\Bio\Tests\Bio.TestAutomation\TestData\TestUtils” and the project needs to be built and then follow the steps as said in section [Running the Automation](#_Running_the_Automation). Similar to the TestsConfig.xml, MUMmerTestsConfig.xml, FastQTestsConfig.xml, NUCmerTestsConfig.xml, QualitativeTestsConfig.xml, MSAConfig.xml , GFFTestsConfig.xml and other config xml’s can be updated.

## Validate the Code Coverage

You must validate the code coverage for Bio.dll.

### Pre-requisite for running the Code Coverage tool Magellan

Install Visual Studio 2010

### Steps for Running Code Coverage in Visual Studio

To run Code Coverage in Visual Studio

1. Launch Visual Studio 2010
2. Enable the checkbox for **Code Coverage** (under **Data and Diagnostics Tab**) in local.testsettings & TraceAndTestImpact.testsettings files under “*$\Bio\”* and double click on **Code Coverage** option and specify the dll which needs to be instrumented and specify the key for the same
3. Change all the project reference in *“$\Bio\Bio.Sln”* to dll reference for Bio.TestAutomation.csproj and its references
4. Build in **Debug** mode
5. Select and Run all the test cases
6. Once all the test cases are run, launch **Code Coverage Results** window to display the results.