BizUnit 4.0

Getting Started Guide

BizUnit is sponsored by 

What is BizUnit?

BizUnit is a light weight and extensible framework to enable automated testing; it is primarily targeted at the testing of distributed systems, but is not limited to that domain. It was originally written to drive functional test for complex systems built on the BizTalk Server platform, but it is in no way tied or limited to that domain and indeed has been widely used to test a broad range of systems and technologies across multiple business verticals.

It is developed in .NET 4.0; this does not however limit it to being used to the testing of Microsoft based technologies. Most distributed systems use platform agnostic messaging systems such as MQ Series, Web Services, TCP/IP etc.; and as such BizUnit may drive tests across platforms and technologies.

BizUnit is a framework. That means that it is fully extensibility via a plug-in architecture, whilst it is released with a large number of test steps, new steps maybe easily written and used with BizUnit. The development of plugins for BizUnit is typically trivial.

This guide gives an overview of BizUnit and how to use it in order to successfully drive up the quality of your software.

Testing to Drive Software Quality

Software testing is a multi-dimensional problem, in order to validate the quality of a software system it needs to be tested from different levels and with different focus. The table below illustrates some of the categories of testing that are appropriate to system testing in order to give a feel for which scenarios usage of BizUnit is appropriate.

Test Category	Description	Is BizUnit Appropriate?
Unit	Testing fine grained interfaces and methods, typically using mocking in order to isolate the scope of the test	No
Functional	Testing of end to end functionality, typically the systems that a solution is integrated with will be mocked out. These types of tests are often referred to as Build Verification Tests (**BVT's**) when they are automated.	**Yes**
Functional Integration	Similar to Functional testing, but the real integrations systems are used instead of mocks	**Yes**
Performance	Testing if a system may handle the load profile expected in a production environment	Yes (in conjunction with other tools such as LoadGen, VS Load Tester. BizUnit may be used to orchestrate the test, collect log files, monitor event logs etc)
Stress	Testing if a system may handle load over a period of time representative of a production environment	Yes (in conjunction with other tools such as LoadGen, VS Load Tester. BizUnit may be used to orchestrate the test, collect log files, monitor event logs etc)

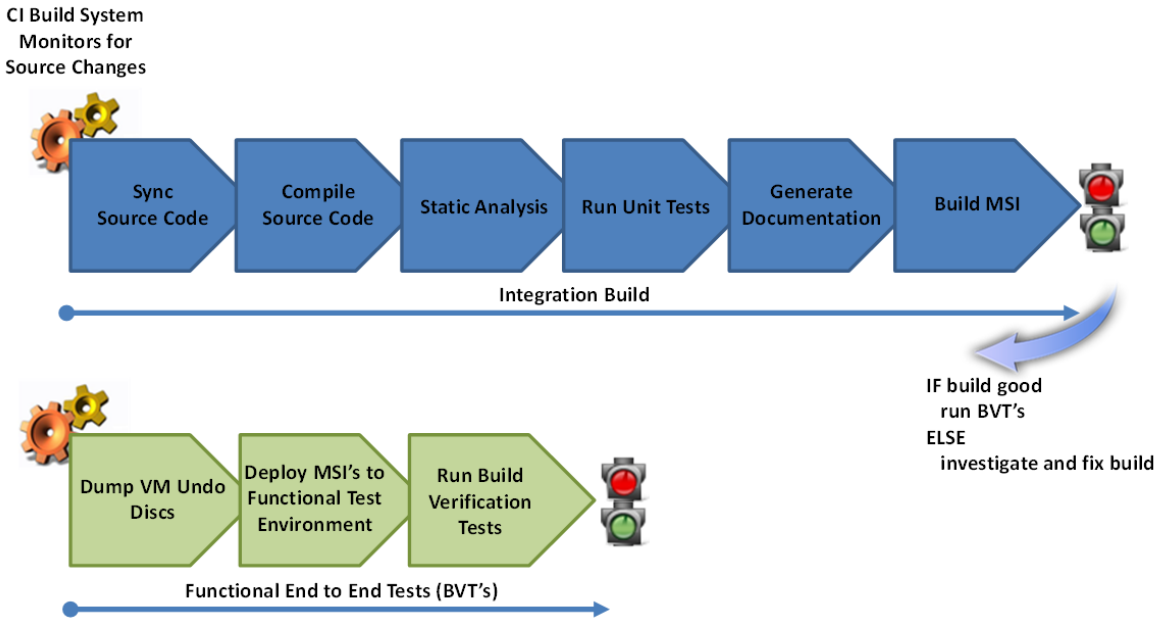
Overload	A variation on stress testing to determine if a system may handle burst of excessive load which may be representative of extreme peeks. For example online stores typically experience much higher load during holiday sales periods	Yes (in conjunction with other tools such as LoadGen, VS Load Tester. BizUnit may be used to orchestrate the test, collect log files, monitor event logs etc)
User Acceptance	The system users test the system by using it as they would in a production environment	No
Disaster Recovery	Test fail over procedures required in a disaster recovery scenario	No (typically this is a manual process)
Live Data Feed	For many systems such as complex trading systems, is it appropriate to test the system against a live feed of data to ensure its behaviour is as expected when used against production data	Yes
Operational Readiness	Testing that the deployment patches, upgrades and system maintenance maybe performed as required in a production environment. Typically this is a very manual process	No

Test Automation

What makes a good test case? In my experience it should be:

- **Repeatable** – you should be able to re-run tests and get the same results
- **Predictable** – assuming the code being tested is working as per its requirements and design, a test should always yield the same result
- **Timely** – tests need to be executed in a timely manner, this reduces the overhead when releasing software into a production environment
- **Extensible** – it should be relatively easy to extend a test suite in order to add additional coverage for new features

This means – **automated.** Automation is the key to shipping high quality software where the cost of doing so is acceptable; BizUnit is targeted at making the cost of automated testing of distributed systems as cheap as possible. In essence BizUnit may be used in scenarios where testing maybe automated, that means the automation of inputs and automation of the validation of the outputs. In general, this also means integrating automated functional tests into the continuous integration build process. The diagram below illustrates where BizUnit often fits within the build process, assuming a continuous integration approach has been adopted. Typically, BizUnit is used to execute the BVT's, which may either be tagged onto the end of the continuous integration (CI) build, or in more sophisticated build environments split out as a separate build that is fed from the CI build, in this scenario, the deployment process is also tested.



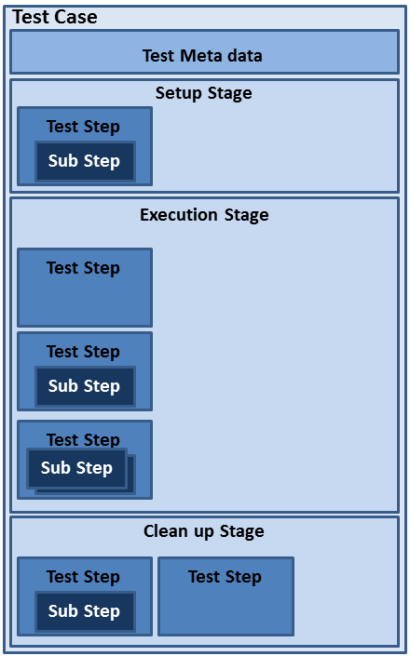
BizUnit Architecture and Features

BizUnit test cases consist of three stages, setup, execution and clean-up. The setup stage is intended to be used to prepare the 'platform' for the test case, the execution stage is about running the scenario that needs to be tested and the clean-up stage is about returning the platform to the condition that it was in prior to the test running.

As mentioned, BizUnit has a plug-in architecture making it very extensible. Each test stage consists of zero or more 'test steps'. A test step is a unit of work that is typically required to succeed in order for a test to succeed. Test cases also contain a collection of meta-data describing the test. The diagram below illustrates this.

Test Cases

A test case may be defined in code or in XAML, and either may be converted to the other. For example a test case may be programmatically constructed in C# and then saved to XAML. At a later time, the XAML test case may be loaded back into the C# object graph and manipulated programmatically.



This ability to round trip test cases between coded tests and XAML has a number of benefits:

- Firstly, the programming model provides a relatively trivial and easy approach when constructing test cases
- Constructing coded test cases provides a type safety when defining test cases, and the developer or tester may enjoy all of the benefits from Visual Studio such as intelisense
- The serialisation of test cases to XAML enables data driven test cases to be developed by varying parameters to produce test case variations, i.e. the XAML test case may be copied and modified to test variations of the scenario at very low development cost
- By providing seamless round tripping of coded tests and XAML tests, the design opens the way for tooling to be created around BizUnit, for example auto documentation of test cases
- All test cases have a consistent serialisation format regardless of who developed them
- Test cases may be modified and executed

The code below illustrates how to create a very trivial test that simple waits for 500 milliseconds. First the test case is created; next the test step(s) that the test is going to execute are created and configured. The steps are added to the test case, then BizUnit is create, passing the new test case into its constructor, and then BizUnit is run, which in turn executes the test case.

```
[TestMethod]
public void DelaySampleTest()
{
    // Create the test case
    var testCase = new TestCase();

    // Create test steps...
    var delayStep = new DelayStep {DelayMilliSeconds = 500};

    // Add test steps to the required test stage
    testCase.ExecutionSteps.Add(delayStep);
```

```
    // Create a new instance of BizUnit and run the test
    var bizUnit = new BizUnit(testCase);
    bizUnit.RunTest();
}
```

A test case may be saved to XAML which is the serialisation technology that BizUnit uses, the code below illustrates how to save a test case:

```
    // Save Test Case
    TestCase.SaveToFile(testCase, "DelaySampleTest.xml");
```

The XAML for the test case defined above may be seen below. You will notice that there are properties which were not set when creating the test case and test step, some of which have been defaulted by BizUnit while some are null. As shown the test case holds meta-data which may be set in order to document the specifics of the test. As will be seen later, all test steps derive from a base test step, which provides meta-data to define how the step should behave, for example if the step fails should the test fail – by default it will. In addition, the execution semantics of the step are defined, by default all steps run sequentially, though for some scenarios it is useful to execute the concurrently. An example of this might be a SOAP step that issues a request and waits for a response that it validates, whilst the step is waiting for the response the test may require other steps to be executed, the 'RunConcurrently' option enables this.

```
<TestCase Category=”{x:Null}”
          Description="{x:Null}"
          ExpectedResults="{x:Null}"
          Name="{x:Null}" Preconditions="{x:Null}"
          Purpose="{x:Null}" Reference="{x:Null}"
          BizUnitVersion="4.0.133.0"
          xmlns="clr-namespace:BizUnit.Xaml;assembly=BizUnit"
          xmlns:btt="clr-namespace:BizUnit.TestSteps.Time;assembly=BizUnit.TestSteps"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <TestCase.ExecutionSteps>
    <btt:DelayStep
          SubSteps="{x:Null}"
          DelayMilliSeconds="500"
          FailOnError="True"
          RunConcurrently="False" />
  </TestCase.ExecutionSteps>
</TestCase>
```

A XAML test case may be loaded and executed in exactly the same manner as a coded test.

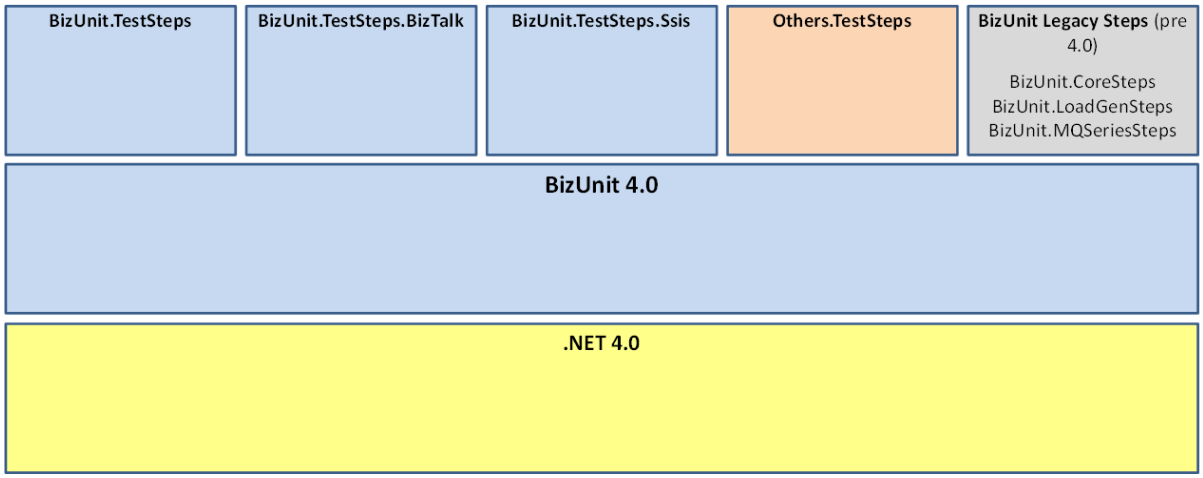
```
// Execute test csse using serialised test case to test round tripping of serialisation...
var bizUnit = new BizUnit(TestCase.LoadFromFile("DelaySampleTest.xml"));
bizUnit.RunTest();
```

BizUnit Architecture

BizUnit consists of a number of assemblies, the core of which is BizUnit.dll, this assembly defines the BizUnit engine, test case class, base classes for test steps, sub steps and data loaders and number of utility classes, it does not include any test steps. Test steps are packaged in separate assemblies; this

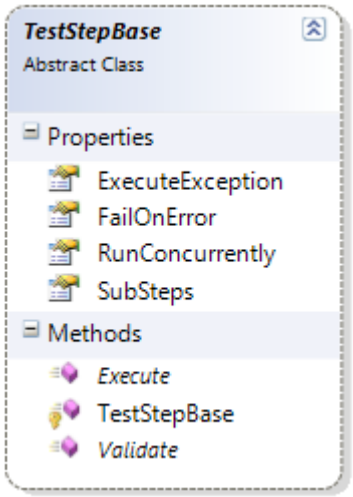
release contains 3 initial test step assemblies for BizUnit 4.0 and 3 legacy assemblies for BizUnit 3.0. Whilst all 3.0 test steps are compatible with BizUnit 4.0, this is the last release that will support the legacy XML format.

BizUnit.TestSteps.dll contains a number of general purpose test steps to support operations such as: FILE operations, Event Log, HTTP, SOAP, SQL as well as some validation steps. The other two assemblies with 4.0 test steps should be self-explanatory. The legacy test steps shipped in this release have not been ported to the new 4.0 interfaces and hence are provided as a stop gap until they are ported.



In order to use BizUnit, BizUnit.dll should be referenced by a test solution, and typically one or more test step library.

Test Steps



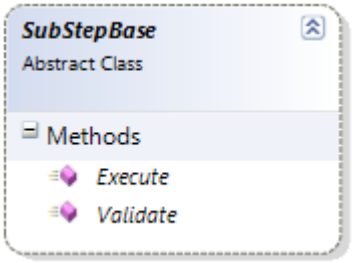
All test steps derive from the abstract class TestStepBase, test steps are required to overwrite the Execute and Validate methods. At test run-time, the Validate method is called first to ensure that the step has been correctly configured, once all test steps have been validated the test is executed, at the appropriate time the test steps Execute method is called. Both of these methods take a Context object which will be discussed later.

Test steps may have zero or more sub steps, the Execute method is responsible for executing these. A typical use of sub-steps is to perform validation, for example an MQ Series Get step may be configured to validate the data that it has read from a queue.

When developing test steps you should ensure that they maybe serialised and deserialised by saving and then loading a test case.

Sub Steps

Similarly to test steps, sub steps should be derived from SubStepBase and override the Execute and Validate methods. Again, sub steps are validated prior to the test case being executed, they should ensure that their configuration is valid in their overridden Validate method.



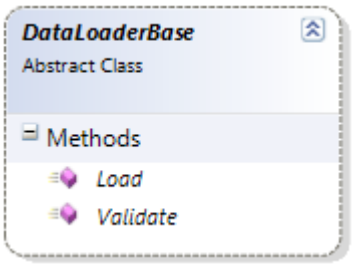
The Execute method is where the sub step does its work, the sub step should use the data stream and context passed into it, is should return the data stream ready for the next consumer. Sub steps should make no assumptions around what the next consumer of the stream is.

Data Loaders

Data loaders provide a mechanism to decouple test steps from their data source and how that data is loaded. For example, a SOAP step will typically require a request payload to send to a service, often this payload data will be loaded from disc. However, for some test cases using this SOAP step any of the following may be required:

- Load the request payload data from a file, but its contents may need to be modified prior to handing the data to the test step
- Load the request payload data from a database
- Load the request payload data form a web service

Data Loaders provide a mechanism to de-coupling test steps so that they may load their data from a separately configured data source, and to modify the data before they consume it.



Test Context

All test cases are executed with a 'Test Context', the context may be passed into the BizUnit constructor or if one is not supplied BizUnit will create one. The context is passed to all test steps, sub steps and data loaders, as such it may be used to hold state within a test case which all 'plug-ins' have access to.

In essence the context contains data specific to the currently executing test case, this data is held as name value pairs. When a test case starts BizUnit stamps two properties into the context, which are:

Property Name	Type	Description
BizUnitTestCaseName	String	The name of the currently executing test case
BizUnitTestCaseStartTime	DateTime	The time that the test case started

The context also supports wild cards which may be useful, the table below shows the list of available wild cards to test steps. A test may use the context to substitute wild cards within its configuration.

Wild Card Name	Description
"%DateTime%"	Substitutes the current data time, the format is HHmmss-ddMMyyyy
%DateTimeISO8601%	Substitutes the current data time, the format in ISO 8601 format
%ServerName%	Substitutes the server name that this test is running on
%Guid%	Substitutes a new GUID
%Test_Start_DateTime%	Substitutes the start time of the test case

If a test step supports wild cards, it should replace them at 'Validation' time. The code below illustrates how the FILE Create step uses substitution in order to allow files to be created with unique and meaningful names.

```
public override void Validate(Context context)
{
    if (string.IsNullOrEmpty(CreationPath))
    {
        throw new ArgumentNullException("CreationPath is either null or of zero length");
    }
    CreationPath = context.SubstituteWildCards(CreationPath);

    DataSource.Validate(context);
}
```

For example the CreationPath property could be set as follows:

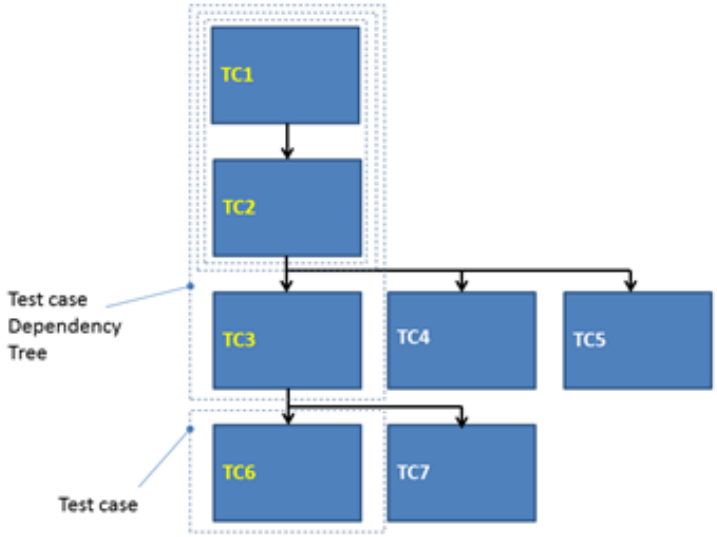
```
    var step = new CreateStep();

    // After wild card replacement: "C:\Output\PO_{676D620F-5FA3-43F3-A30D-C60C512FE2C6}.xml"
    step.CreationPath = "C:\Output\PO_%Guid%.xml";

    // After wild card replacement: "C:\Output\PO_Thunderbolt_ 123354-22032011.xml"
    step.CreationPath = "C:\Output\PO_%ServerName%_%DateTime%.xml";
```

Importing Test Cases

Often test cases are built up, progressively testing more and more functionality, or they are built in a manner whereby they have dependencies. The diagram below illustrates this, test case #6 is dependent on test case 3, which is in turn dependant on test case 1 and 2.



A real world example of this might be where a GetBookingStatus web service operation needs to be tested, GetBookingStatus takes a bookingId, so before it can be called, CreateBooking needs to be called which returns a bookingId. In order to test GetBookingStatus, a CreateBooking test can be created, which returns a bookingId and sets it in the BizUnit context, next GetBookingStatus is called, but the request message is modified using an Xml data loader setting the bookingId of the request message to the value previously set in the context. This example shows how the various BizUnit features maybe used to achieve a reasonably complex scenario.

To achieve this, BizUnit offers the ability to import a test case at any stage within another test case. The code below illustrates a simple import scenario. The first test case creates a file “File1.xml”. The second test case imports the first test case, then creates a second file “File2.xml”, then uses a validating file read step to check that two files were created that also validate against the defined schema. The code below illustrates this.

```
Using BizUnit.TestSteps.Common;
using BizUnit.TestSteps.DataLoaders.File;
using BizUnit.TestSteps.File;
using BizUnit.TestSteps.ValidationSteps.Xml;
using BizUnit.Xaml;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BizUnit.TestSteps.Tests.ImportTestCase
{
    [TestClass]
    public class ImportTestCaseTest
    {
        [TestMethod]
        public void ImportSingleTestCaseTest()
        {
            // Create the first test case i a helper method...
            var testCase1 = BuildFirstTestCase();

            // Create the second test case and import the first test case into it...
            var testCase2 = new TestCase {Name = "Copy First File Test"};
```

```
            var createFileStep = new CreateStep {CreationPath = @"File2.xml"};
            var dl = new FileDataLoader
                        {
                            FilePath =
                        @"..\..\..\Test\BizUnit.TestSteps.Tests\TestData\PurchaseOrder001.xml"
                        };
            createFileStep.DataSource = dl;

            testCase2.ExecutionSteps.Add(createFileStep);

            var import = new ImportTestCaseStep {TestCase = testCase1};
            testCase2.ExecutionSteps.Add(import);

            // Create a validating read step...
            var validatingFileReadStep = new FileReadMultipleStep
                            {
                                DirectoryPath = @".",
                                SearchPattern = "File*.xml",
                                ExpectedNumberOfFiles = 2
                            };

            var validation = new XmlValidationStep();
            var schemaPurchaseOrder = new SchemaDefinition
            {
                XmlSchemaPath =
                    @"..\..\..\Test\BizUnit.TestSteps.Tests\TestData\PurchaseOrder.xsd",
                XmlSchemaNameSpace =
                    "http://SendMail.PurchaseOrder"
            };
            validation.XmlSchemas.Add(schemaPurchaseOrder);

            var xpathProductId = new XPathDefinition
                                    {
                                        Description = "PONumber",
                                        XPath =
                                            "/*[local-name()='PurchaseOrder' and namespace-
uri()='http://SendMail.PurchaseOrder']/*[local-name()='PONumber' and namespace-uri()='']",
                                        Value = "12323"
                                    };
            validation.XPathValidations.Add(xpathProductId);
            validatingFileReadStep.SubSteps.Add(validation);
            testCase2.ExecutionSteps.Add(validatingFileReadStep);

            // Run the second test case...
            var bizUnit = new BizUnit(testCase2);
            bizUnit.RunTest();
        }

        private TestCase BuildFirstTestCase()
        {
            var testCase1 = new TestCase {Name = "Copy First File Test"};

            var step = new CreateStep();
            step.CreationPath = @"File1.xml";
            var dl = new FileDataLoader();
            dl.FilePath =
                @"..\..\..\Test\BizUnit.TestSteps.Tests\TestData\PurchaseOrder001.xml";
            step.DataSource = dl;
            step.Execute(new Context());

            testCase1.ExecutionSteps.Add(step);
```

```
            return testCase1;
        }
    }
}
```

The same coded test case shown above may also be defined in XAML as shown below:

```
<TestCase Category="{x:Null}" Description="{x:Null}" ExpectedResults="{x:Null}"
Preconditions="{x:Null}" Purpose="{x:Null}" Reference="{x:Null}" BizUnitVersion="4.0.133.0"
Name="Copy First File Test"
  xmlns="clr-namespace:BizUnit.Xaml;assembly=BizUnit"
  xmlns:btc="clr-namespace:BizUnit.TestSteps.Common;assembly=BizUnit.TestSteps"
  xmlns:btdf="clr-namespace:BizUnit.TestSteps.DataLoaders.File;assembly=BizUnit.TestSteps"
  xmlns:btf="clr-namespace:BizUnit.TestSteps.File;assembly=BizUnit.TestSteps"
  xmlns:btvx="clr-namespace:BizUnit.TestSteps.ValidationSteps.Xml;assembly=BizUnit.TestSteps"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
  <TestCase.ExecutionSteps>
    <btf:CreateStep SubSteps="{x:Null}" CreationPath="File2.xml" FailOnError="True"
      RunConcurrently="False">
      <btf:CreateStep.DataSource>
        <btdf:FileDataLoader
          FilePath="..\..\..\Test\BizUnit.TestSteps.Tests\TestData\PurchaseOrder001.xml" />
      </btf:CreateStep.DataSource>
    </btf:CreateStep>
    <ImportTestCaseStep SubSteps="{x:Null}" TestCasePath="{x:Null}" FailOnError="True"
      RunConcurrently="False">
      <ImportTestCaseStep.TestCase>
        <TestCase Category="{x:Null}" Description="{x:Null}" ExpectedResults="{x:Null}"
          Preconditions="{x:Null}" Purpose="{x:Null}" Reference="{x:Null}"
          BizUnitVersion="4.0.133.0" Name="Copy First File Test">
          <TestCase.ExecutionSteps>
            <btf:CreateStep SubSteps="{x:Null}" CreationPath="File1.xml" FailOnError="True"
              RunConcurrently="False">
              <btf:CreateStep.DataSource>
                <btdf:FileDataLoader
                  FilePath=
                    "..\..\..\Test\BizUnit.TestSteps.Tests\TestData\PurchaseOrder001.xml" />
              </btf:CreateStep.DataSource>
            </btf:CreateStep>
          </TestCase.ExecutionSteps>
        </TestCase>
      </ImportTestCaseStep.TestCase>
    </ImportTestCaseStep>
    <btf:FileReadMultipleStep DeleteFiles="False" DirectoryPath="." ExpectedNumberOfFiles="2"
      FailOnError="True" RunConcurrently="False" SearchPattern="File*.xml" Timeout="0">
      <btf:FileReadMultipleStep.SubSteps>
        <btvx:XmlValidationStep>
          <btvx:XmlValidationStep.XPathValidations>
            <btc:XPathDefinition ContextKey="{x:Null}" Description="PONumber" Value="12323"
              XPath="/*[local-name()='PurchaseOrder' and namespace-
uri()='http://SendMail.PurchaseOrder']/*[local-name()='PONumber' and namespace-uri()='']" />
          </btvx:XmlValidationStep.XPathValidations>
          <btvx:XmlValidationStep.XmlSchemas>
            <btvx:SchemaDefinition XmlSchemaNameSpace="http://SendMail.PurchaseOrder"
              XmlSchemaPath="..\..\..\Test\BizUnit.TestSteps.Tests\TestData\PurchaseOrder.xsd"
/>
```

```
          </btvx:XmlValidationStep.XmlSchemas>
        </btvx:XmlValidationStep>
      </btf:FileReadMultipleStep.SubSteps>
    </btf:FileReadMultipleStep>
  </TestCase.ExecutionSteps>
</TestCase>
```

Hopefully this guide has given a flavour for the functionality available in BizUnit 4.0 and how to use it. It's important to remember that it is a framework and can be extended very easily.

Happy testing!

Kevin B Smith