Matthew Manela

Building high quality software and teams

June 12, 2012 by Matthew Manela | Chutzpah, Visual Studio 2012

# Anatomy of the Chutzpah test adapter for VS 2012 RC

*Note: The following post describes interfaces that are part of the VS 2012 RC. These are still subject to change and may be different when the final version is released.*

Coinciding with the release Visual Studio 2012 RC, I pushed an update of the Chutzpah test adapter for the Unit Test Explorer (UTE). I increased performance of the adapter through better caching and the new UTE notification feature which lets a test adapter notify the UTE when a test has changed. With these changes, the Chutzpah adapter is a strong example of a file based test adapter.

The Chutzpah test adapter revolves around four interfaces:

1. **ITestContainer** – Represents a file that contains tests
2. **ITestContainerDiscoverer** – Finds all files that contain tests
3. **ITestDiscoverer** – Finds all tests within a test container
4. **ITestExecutor** – Runs the tests found inside the test container

This post will cover each of these interfaces, show Chutzpah's implementations and describe how to debug test adapters.

# ITestContainer

A test container is an object that contains tests (shocking right?). You can have a test container represent a file, folder, a dll or anything else.  For Chutzpah a test container represents a JavaScript test file.  The interface for a test container contains several members:

```
1  public interface ITestContainer
2  {
3    ITestContainerDiscoverer Discoverer { get; }
4    string Source { get; }
5    int CompareTo(ITestContainer other);
6    ITestContainer Snapshot();
7    IEnumerable<Guid> DebugEngines { get; }
8    FrameworkVersion TargetFramework { get; }
9    Architecture TargetPlatform { get; }
10   bool IsAppContainerTestContainer { get; }
11   IDeploymentData DeployAppContainer();
12 }
```

but the most important are:

- **Discoverer** – An instance of a test container discoverer, this is covered this in the next section.
- **Source** – A string that identifies this test container. For Chutzpah this is the file path.
- **CompareTo** – Compares two test containers to see which is newer.

To implement CompareTo the Chutzpah test container includes a timestamp field. When the container gets created the timestamp is set to the last modified time of the file (whose path is the source property). The CompareTo method checks if the sources are the same and then compares their timestamps:

```
1  public int CompareTo(ITestContainer other)
2  {
3      var testContainer = other as JsTestContainer;
4      if (testContainer == null)
5      {
6          return -1;
7      }
8
9      var result = String.Compare(this.Source, testContainer.Source,
10 StringComparison.OrdinalIgnoreCase);
11     if (result != 0)
12     {
13         return result;
14     }
15
16     return this.timeStamp.CompareTo(testContainer.timeStamp);
   }
```

You can view Chutzpah's implementation of ITestContainer in JsTestContainer.cs.

# ITestContainerDiscoverer

A container discoverer is responsible for finding test containers and returning them to the UTE. It has a simple interface:

```
1  public interface ITestContainerDiscoverer
2  {
3      Uri ExecutorUri { get; }
4      IEnumerable<ITestContainer> TestContainers { get; }
5      event EventHandler TestContainersUpdated;
6  }
```

- **ExecuteUri** – A string which uniquely identifies a test adapter. This string is used to tie a container discoverer with a test discoverer and executor.
- **TestContainersUpdated** – An event to invoke when test containers are found or are changed.
- **TestContainers** – A property which returns the discovered test containers. This is called when the UTE when a solution is first loaded, before unit tests are executed and after you fire the TestContainersUpdated event.

Since the TestContainers property is called often it is crucial that it returns quickly. In Chutzpah's initial implementation the whole solution was scanned each time looking for test files and returning containers for them. This was far too slow. Thus, the current implementation it keeps a cached list of test containers which are updated incrementally by monitoring solution and

file events. When the TestContainers property is called the cached list is returned.

To manage the cached list of test containers the container discoverer subscribes to IVsSolutionEvents for solution events, IVsTrackProjectDocumentsEvents2 for project events and use a FileSystemWacther for file events.

IVsSolutionEvents fires events when projects and solutions are loaded and unloaded. When a project is loaded the discoverer scans for JavaScript files, checks if they contain tests and adds them to the containers list. On project unload its test containers are removed from the list. IVsTrackProjectDocumentsEvents2 fires events when a file is added, removed or renamed in a project. When these events occur test containers are added or removed. FileSystemWacther fires events when the content of the file has changed. File watchers are added to monitor each JS file in the solution to detect when they change. Whenever a change happens the test container list is updated with a new instance of a test container and the old instance is removed.

To help keep the list of test containers small the container discoverer doesn't create test containers for all JS files. When an event occurs on a file it is opened and checked to see if it contains tests. If it does then a test container for it is added to the list. This saves the ITestDiscoverer from needed to do extra work and results in a quicker testing experience.

You can see the implementation of these interfaces in JsTestContainerDiscoverer.cs, SolutionEventsListener.cs,

TestFileAddRemoveListener.cs and TestFilesUpdateWatcher.cs.

# ITestDiscoverer

Now that the UTE has the test containers the next step is discovering what tests are within them. The UTE looks for an implementation of ITestDiscoverer that has an **DefaultExecutorUri** attribute set to the same value as the **ExecutorUri** property in ITestContainerDiscoverer.

This interface contains one method.

```
public interface ITestDiscoverer
{
  void DiscoverTests(
          IEnumerable<string> sources,
          IDiscoveryContext discoveryContext,
          IMessageLogger logger,
          ITestCaseDiscoverySink discoverySink);
}
```

The key arguments are **sources** and **discoverySink**. Each string in the sources corresponds to the **Source** property on a test container. The discoverySink argument is an instance of ITestCaseDiscoverySink which also contains one method:

```
public interface ITestCaseDiscoverySink
{
  void SendTestCase(TestCase discoveredTest);
}
```

For each source in the sources list the test discoverer will open that file and scan it for tests. For each test found a test case object is created and sent to the discovery sink.

In addition, the test discovery implementation needs the **FileExtension** attribute set to the file extension the discoverer is interested, for Chutzpah this is **.js**. Here is Chutzpah's test discovery class:

```
[FileExtension(".js")]
[DefaultExecutorUri(Constants.ExecutorUriString)]
public class JsTestDiscoverer :ITestDiscoverer
{
    public void DiscoverTests(IEnumerable<string> sources,
                              IDiscoveryContext discoveryContext,
                              IMessageLogger logger,
                              ITestCaseDiscoverySink discoverySink)
    {
        var chutzpahRunner = TestRunner.Create();
        foreach (var testCase in
 chutzpahRunner.DiscoverTests(sources))
        {
            var vsTestCase = testCase.ToVsTestCase();
            discoverySink.SendTestCase(vsTestCase);
        }
    }
}
```

You can also view this on CodePlex in JsTestDiscoverer.cs.

# ITestExecutor

And finally we arrive at the last step in the process which is test execution. A test executor implements the ITestExecutor

interface and has an **ExtensionUri** attribute on it that defines the executor uri we saw earlier. The ITestExecutor interface contains three members:

```
public interface ITestExecutor
{
  void RunTests(IEnumerable<string> sources, IRunContext runContext,
IFrameworkHandle frameworkHandle);
  void RunTests(IEnumerable<TestCase> tests, IRunContext runContext,
IFrameworkHandle frameworkHandle);
  void Cancel();
}
```

- **RunTests(string)** – Called when running all tests. It receives a collection of strings which correspond to the sources in the test containers.
- **RunTests(TestCase)** – Called when running selected tests. Chutzpah doesn't yet support running individual tests (it runs the whole js file). It will grab the test container source from the TestCase object and call the other RunTests method that takes a list of test container sources.
- **Cancel** – Called when the user tries to cancel the test. Chutzpah doesn't implement this yet.

Here is Chutzpah's test execution class:

```
[ExtensionUri(Constants.ExecutorUriString)]
public class JsTestExecutor : ITestExecutor
{
  public void Cancel()
  {
          // Will add code here when streaming tests is implemented
  }

  public void RunTests(IEnumerable<string> sources, IRunContext
runContext, IFrameworkHandle frameworkHandle)
```

```
  {
          if (runContext.IsDataCollectionEnabled)
          {
              // DataCollectors like Code Coverage are currently
unavailable for JavaScript
              frameworkHandle.SendMessage(TestMessageLevel.Warning,
"DataCollectors like Code Coverage are unavailable for JavaScript");
          }

    var chutzpahRunner = TestRunner.Create();
    var callback = new ExecutionCallback(frameworkHandle);
    chutzpahRunner.RunTests(sources, callback);
  }

  public void RunTests(IEnumerable<TestCase> tests, IRunContext
runContext, IFrameworkHandle frameworkHandle)
  {
    // We'll just punt and run everything in each file that contains
the selected tests
    var sources = tests.Select(test => test.Source).Distinct();
    RunTests(sources, runContext, frameworkHandle);
  }
}
```

In the above code the IFrameworkHandle argument on the RunTests is wrapped in an ExecutionCallback class. This class implements the TestFinished callback that Chutzpah calls when running tests. The results of the test are converted into TestCase and TestResult objects and are passed to methods on the IFrameworkHandle object. The IFrameworkHandle interface inherits from ITestExecutionRecorder which provides methods for recording the beginning, end and results of a test case.

```
public interface ITestExecutionRecorder : IMessageLogger
{
  void RecordResult(TestResult testResult);

  void RecordStart(TestCase testCase);

  void RecordEnd(TestCase testCase, TestOutcome outcome);

  void RecordAttachments(IList<AttachmentSet> attachmentSets);
}
```

Inside of TestFinished, the methods RecordStart, RecordResult and RecordEnd are called:

```
public void TestFinished(Chutzpah.Models.TestResult result)
{
  var testCase = result.ToVsTestCase();
  var vsresult = result.ToVsTestResult();
  var outcome = result.ToVsTestOutcome();

  // The test case is starting
  frameworkHandle.RecordStart(testCase);

  // Record a result (there can be many)
  frameworkHandle.RecordResult(vsresult);

  // The test case is done
  frameworkHandle.RecordEnd(testCase, outcome);
}
```

It may seem odd that Chutzpah is invoking both RecordStart and RecordEnd from its TestFinished method but this is a result of Chutzpah's execution of test files. Chutzpah executes the whole JS test file and collects the results. This means that it can't notify when an individual test has started until the test is already completed. There are plans to change this in the future by adding the ability to stream test results while a test file is running.

You can view Chutzpah's implementation of ITestExecutor in JsTestExecutor.cs.

# Debugging Test Adapters

The UTE will call into the interfaces listed above from three different processes. In order to debug, you must be attached to the correct ones.

- ITestContainerDiscoverer is called from the main Visual Studio process named devenv.exe. This is the default process you attach to when debugging.
- ITestDiscoverer is called from a process named vstest.discoveryengine.x86.exe. This process starts when the UTE is first opened.
- ITestExecutor is called from a process named vstest.executionengine.x86.exe. This process starts during first discovery pass.

If you are attached to all three then you can be sure that your breakpoints will be hit.

That was a quick tour of Chutzpah's test adapter implementation. When the final version of Visual Studio 2012 is released I will update this post to reflect any changes.

Snippet Designer now supports Visual Studio 2012 RC →

•

← Snippet Designer now supports C++

# 8 thoughts on “Anatomy of the Chutzpah test adapter for VS 2012 RC”

**Chris** says:
July 12, 2012 at 4:41 pm

Thanks for your article, I found it helpful. Especially the debugging part. I have one question, where does the testrunner come from in this code?

```
var chutzpahRunner = TestRunner.Create();
```

Thanks again.

**Matthew** says:
July 12, 2012 at 9:24 pm

That is code from Chutzpah library that returns an instance of the chutzpah test runner. That is defined here

Pingback: 将JavaScript测试集成到开发工作流中 | chainding

**Akram Alhinnawi** says:
January 17, 2013 at 2:21 pm

Greate work Matthew,
One question: what are the steps to begin debugging the adapter? meaning how can you do the 3 process attachemnts? In fact, I'm more interested with your XMLTestAdapter because it has the same code structure.

**Rafael L.** says:
February 3, 2014 at 10:15 am

Maybe something like that will help you if you want to debug the adapter (not the tests).

```
#if LAUNCHDEBUGGER
Debugger.Launch();
#endif
```

I used to debug my adapter during the development.

Pingback: 将JavaScript测试集成到开发工作流中 | 蛋丁-互联网的一道菜

**Arthur Strutzenberg** says:
December 30, 2014 at 12:46 pm

Is there any way for you to specify which adapter you want the unit test system to use at run time? What I'm trying to

accomplish is to recycle the test discovery for the built in unit test structure, but pass the test case execution off to a different executor?

**Matthew Manela** says:
December 30, 2014 at 2:04 pm

I am not sure if this is possible. But I have not delved deeply into trying that.

Comments are closed.

## Recent Posts

- Exploring the power of LLM Embeddings
- Joint Accountability
- Solving the hard problem of opinions and feedback
- A culture of experimentation
- Technical Debt and your Sock Drawer

## Featured Posts

- Psychological Safety
- Idiomatically.net – A site for exploring idioms across languages and locales
- Building productive, customer focused teams

## Search

Search ...