Dr. Dobb's Journal

November 2012

Testing Complex Systems

Good design practices can save your system and your sanity

Next

ALSO INSIDE

UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

November 2012

GUEST EDITORIAL

COVER STORY

More on DrDobbs.com

Component Programming in D

Composability, or the ability to write highly reusable software using a data source -> algorithm -> data sink model, is a feature that depends heavily on language support. New facilities in D now make this possible.

http://www.drdobbs.com/architecture-and-design/240008321

Two Years Later: A Report Card On Oracle's Ownership of Java

After a bumpy start, Oracle's stewardship of Java has steadily improved, except for the Google mess and a feckless JCP.

http://www.drdobbs.com/jvm/240008729

Parallel In-Place Merge

Merging sorted arrays in parallel and in place can be done very efficiently, using this algorithm. Comparisons with the performance of similar STL functions are included.

http://www.drdobbs.com/parallel/240008783

What Happens After A Failure?

Imagine that you have asked a program to do something for you, and it has reported that it is unable to do so. What do you do next?

http://www.drdobbs.com/cpp/240008870

Mind Maps: The Poor Man's Design Tool

UML too complex? Flowcharts too old school? Mind maps offer a simple way to capture designs and weave them together elegantly.

http://www.drdobbs.com/tools/240008292

Pex: Microsoft Research's Unit Test Generator and Evaluator

Tired of grinding out unit tests by hand? Worried that you've missed a set of values to test for? Microsoft's little-known Pex is the answer.

By Dino Esposito

Pex is a testing tool developed by Microsoft Research. It provides three very useful capabilities to support testing. First, it can explore your code and suggest which tests you should have. Second, if you have parameterized tests, Pex can figure out which combination of parameters needs be tested in order to give you full coverage of possible scenarios. And finally, if you use Code Contracts, then Pex will employ that information to fine-tune the unit tests it suggests or generates for you. I'll review the three aspects in more detail after a quick look at installation and set up.

Download Pex from Microsoft at http://research.microsoft.com/en-us/projects/pex. If you're curious about the underlying technology, you can also access a few PDF documents explaining the internal workings of the tool from the site. Pex is a standard add-in for Visual Studio and should be installed like any other.

Suggesting Unit Tests

Now let's look at what Pex can do. Suppose you are not a TDD proponent; at some point, you may happen to have a C# class and no unit tests for it. With Pex, all you need do is right-click on the class code in Visual Studio and click the Pex|Create Parameterized Unit Tests menu item. The effect is that a new test project is created and added to the selected solution; moreover, the test project can target MSTest — the default Visual Studio testing framework — as well as other frameworks, including NUnit. Open up the project and you'll find good parameterized tests are there for you to complete and run. However, auto-generated tests don't really do much except invoke a method with parameters. At the very minimum, you'll want to complete them with some assertions.

A parameterized unit test is simply a test that accepts parameters; parameters can be provided in a number of ways depending on the framework. Typically, you pass parameters from a database file. In Visual

Studio MSTest, you need to write a wrapper parameter-less test configured to know about the data source. This wrapper test will be invoked iteratively; it reads values from the current row and passes those values to the parameterized test. The challenge is choosing meaningful input values for the tests. This is where the second key feature of Pex comes to the rescue.

"If Code Contracts are enabled for the project, then Pex will offer to add a proper precondition to your code"

Choosing Meaningful Input Values

Open the previously created unit test file and right-click on it. This time, click on the Run Pex Explorations menu item. At this point, Pex analyzes the code of the method under test and figures out which values it would make sense to test for. Pex uses an implementation of the dynamic symbolic execution technique that basically processes the code and calculates the results of the method execution. Every time a new possible result is found, Pex determines which combination of values leads to the result and adds the values to the list. In this way, it iteratively discovers all possible outcomes of a method and, at the same time, it also determines a way to produce them.

The result of a Pex exploration is a list of auto-generated classic unit tests, which call the method with any meaningful combination of inputs previously determined. You can just copy these tests into a test project and integrate them in the build process with the guarantee of a high coverage and, more importantly, strong insurance against corner cases. Generating parameterized unit tests is often a long and boring process — Pex automates that process and reduces it to just a click or two.

Code Contracts and Pex

Fully integrated in .NET Framework 4.0, Code Contracts are Microsoft's implementation of the tried-and-true idea of software contracts — most notably, preconditions, postconditions, and invariants. In particular, Code Contracts are used to annotate methods with conditions that must be preliminarily verified for the code to run. As an example of Code Contracts and Pex integration, consider the following scenario. Suppose you have a method that does some work on a string parameter. Suppose also that your implementation blindly assumes the received string is a non-null string. You are therefore potentially exposed to a null reference exception. A quick exploration by Pex reveals the issue, in the form of a notification, which allows you to add a test.

A warning about a possible defect is a great help, but Pex can do more. If Code Contracts are enabled for the project, then Pex will offer to add a proper precondition to your code so that you can catch any invalid data and take control of the situation. Pex will run an analysis of the code and build a base of knowledge about your code. Code Contracts provide additional information that contributes to an even more accurate analysis from Pex.

In summary, Pex is an innovative white-box testing tool that can be used both as an aid to generate nontrivial unit tests and as a peer reviewer to quickly look at your code and find holes and omissions in it. Pex is not dependent on Code Contracts, but it can use Code Contracts to enhance your tests if they are already employed in your code. One limitation I should mention is that it works only with managed code.

More Testing

Pex is an innovative testing tool that represented a deeper commitment to unit testing than Microsoft had previously exhibited. Since it was released, Microsoft has continued bringing out tools that facilitate unit testing. In Visual Studio 2012, for example, it shipped a framework called "Microsoft Fakes" (http://msdn.microsoft.com/en-us/library/hh549175.aspx). Fakes facilitates the construction of unit tests with stubs and mocks, plus a runtime redirection of code in what the company calls "shims." These technologies will be discussed in upcoming articles. Meanwhile, if you need help generating humdrum unit tests that exercise code thoroughly, download Pex and have it do most of the heavy lifting.

[For a hands-on look at using Pex, see "Working with Microsoft PEX Framework" (http://www.drdobbs.com/windows/working-with-microsoft-pex-framework/224201591) — *Ed.*]

— Dino Esposito is a frequent writer on Microsoft developer technologies.

Comment

Testing Complex Systems

Good design practices can save your system and your sanity

By Gigi Sayfan

Everybody knows that unit tests are important and you should strive to test every piece of code you write. Test-driven development (TDD) proponents even believe that you should write the tests first and use them as a design tool. Unfortunately, for most teams and systems, the real world rudely intervenes and automated test coverage is far from complete (including unit tests, integration tests, and full system tests).

The reasons for this apparent neglect of a best practice are diverse. The shocking and painful truth for test aficionados is that successful complex systems can be developed without adequate automated testing. The costs may be higher and necessitate more manual testing, but it is possible. Even the most test-oriented development teams will agree that their code is under-tested (except maybe in some life-critical and mission-critical industries). In this article, I drill down and explain how to write tests for traditionally difficult-to-test areas of the code. As you'll see, testing complex systems has a lot to do with the architecture and design of your code. This is also true for other aspects of software systems such as scalability, performance, and security. You can't just bolt them on top of a horrible mess of code. You have to design testability from the get go or refactor toward it. The good news is that good design (modular and loosely coupled elements with well-defined responsibilities and interfaces between modules) leads to systems that are more testable, scalable, performant, and secure.

The Basics

Develop complex software often involves multiple teams — possibly not even colocated and not in the same time zone. You must have a solid development process that involves source control, an automated build/deployment system, and automated tests. A major concern in such an environment is how to avoid breaking the build, because a broken build caused by one developer can block every other developer. A broken build includes code that doesn't behave as expected.

For example, if I introduce a bug to the data access layer, every piece of code that tries to access the data will fail. The more complex the system, the harder it is to gauge the impact of a particular change (this can be minimized by good design). Automated tests become critical here. If all the tests pass after you make your change, you can be pretty sure that you didn't break anyone else's code (assuming reasonable test coverage).

MindReader

For demonstration purposes, I use a C# application, called "MindReader," that reads your mind and tells you what you are thinking about. Figure 1 shows what it looks like in action.

It has a single button for initiating mind reading, a progress bar to report the progress, and an output area to tell you what you are thinking about. There can be only one mind reading going on any one time, so the button is disabled while mind reading is occurring. It's a pretty simple app, but it will allow us to explore sophisticated testing techniques.

Third-Party Code

In a complex system, your team may write only part of the code. You may use open-source libraries, licensed code from external vendors, and (most commonly) code from other teams. As a rule of thumb, you shouldn't create tests for code you didn't write. You normally use third-party code through an API. In some cases, you just use the API directly in your code. When operating in this mode, the third-party is equivalent to built-in libraries for your programming language or the OS. Often, however, raw API access is not the best approach. The API may be very

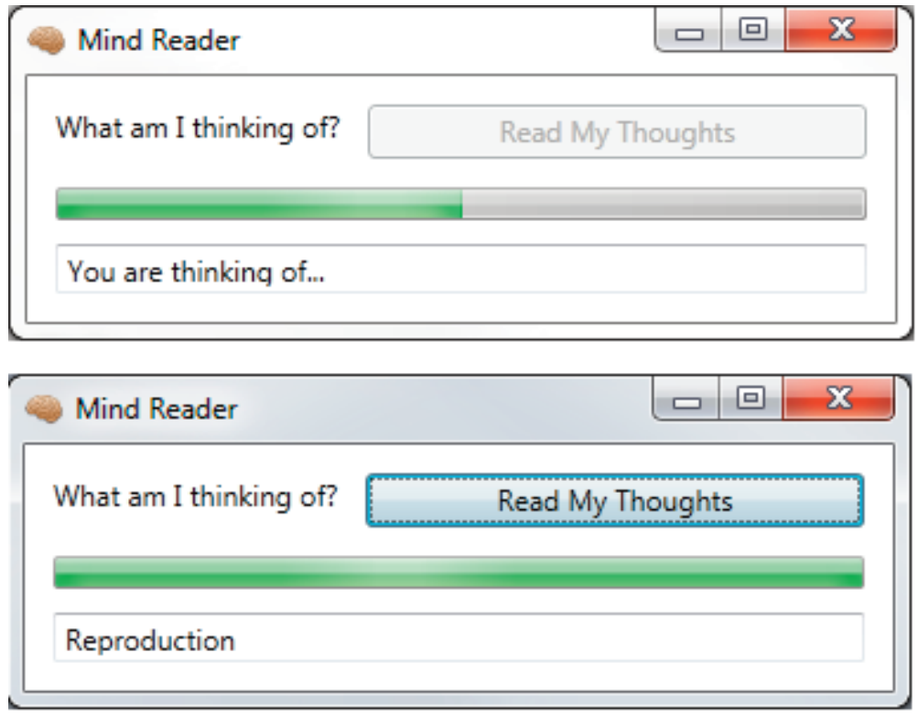

Figure 1.

complicated, it may not be stable across versions, it may be too low-level, it may be a C API while your system is implemented in C++. In all these cases, teams can elect to write a wrapper around the third-party code that exposes the required functionality, then use the wrapper. These wrappers can be convenient for testing, too, because (if designed properly) they're easy to mock. Sometimes, mocking third-party code is the main justification for writing a wrapper.

When writing wrappers, I recommend that you create a very thin wrapper only around the functionality you need. The goal is to make sure you don't introduce any bugs or performance issues with your

wrappers. If the API is large and changes often, look into generating the wrappers automatically if possible. You shouldn't need to write dedicated tests for these wrappers. You may later write another layer on top of the thin wrapper to adapt the API to your application needs, but by then, you are already back in the domain of your code and the access layer can be tested using a mocked version of the thin wrapper if needed.

The only other concern about third-party code is that if you mock it, you need to really understand its behavior — especially for error cases. If an API method returns an error status or throws an exception and your mock doesn't mimic it, you didn't test your code properly against the API.

Another realm of third-party code involves frameworks and plugin-containers. In these cases, you should develop your code to be as host/container-agnostic as possible and test it separately.

The MindReader UI uses Windows Presentation Foundation (WPF) as third-party code and also as the framework where the application starts executing. WPF is usually coupled to your code via event handlers, which may be considered loose coupling, but it means you must deal with WPF signatures and data types (so when a button is clicked, your event handler gets a `System.Windows.RoutedEventArgs` object that it needs to handle).

The `IMainWindow` interface abstracts all the operations the MindReader app performs on the main window:

```
public interface IMainWindow
{
    void EnableGoButton(bool enable);
    void SetThoughtText(string thought);
    void UpdteProgressBar(int percent);
}
```

The `IMainWindowEvents` interface abstracts all the relevant events that the main window generates and the application wants to respond to:

```
{
    void OnGoButtonClick();
    void OnKey(string key);
    void OnClose();
}
```

Note that these interfaces are completely WPF-agnostic. You can switch the UI technology at any time or mock it for testing purposes, as you'll see later. The `MainWindowWrapper` class (see Listing One) implements the `IMainWindow` interface and forwards calls to the actual `MainWindow` WPF class. It also listens to certain WPF events (via event handlers) and forwards them to its `IMainWindowEvents` sink.

Listing One

```
using System;
using System.Windows.Controls;

namespace MindReader
{
    class MainWindowWrapper : IMainWindow
    {
        MainWindow _window;
        IMainWindowEvents _sink;

        Button _goButton;
        public MainWindowWrapper(MainWindow window)
        {
            _window = window;
            _goButton = (Button)_window.FindName("goButton");

            // Hook up event handlers
            _goButton.Click += new System.Windows.
                RoutedEventHandler(_goButton_Click);
```

```
    _window.KeyUp += new System.Windows.Input.
        KeyEventHandler(_window_KeyUp);
    _window.Closed +=
       new EventHandler(_window_Closed);
}

public void AttachSink(IMainWindowEvents sink)
{
    _sink = sink;
}

// MainWindow events
void _window_Closed(object sender, EventArgs e)
{
    _sink.OnClose();
}

void _window_KeyUp(object sender, System.
   Windows.Input.KeyEventArgs e)
{
    _sink.OnKey(e.Key.ToString());
}

void _goButton_Click(object sender,
   System.Windows.RoutedEventArgs e)
{
    _sink.OnGoButtonClick();
}

// IMainWindow
public void EnableGoButton(bool enable)
{
    _window.Dispatcher.Invoke((Action)(() =>
    {
        _goButton.IsEnabled = enable;
    }));
}

public void SetThoughtText(string thought)
{
    _window.Dispatcher.Invoke((Action)(() =>
    {
        _window.thoughtBox.Text = thought;
    }));
}

public void UpdteProgressBar(int percent)
{
    _window.Dispatcher.Invoke((Action)(() =>
    {
        var min = _window.progressBar.Minimum;
        var max = _window.progressBar.Maximum;
        var range = max - min;
        _window.progressBar.Value = min +
           (percent / 100.0) * range;
    }));
}
   }
}
```

Testing Cross-Platform Systems

This section discusses systems that run on different operating systems with identical behavior.

I've written a lot of cross-platform code, mostly in C++. Very often, nasty cross-platform problems turn out to be a build issues (wrong flags, wrong dependencies, wrong version, etc). The key for cross-platform programming is to minimize platform-specific code as much as possible. This can be largely accomplished by using cross-platform libraries and avoiding direct system calls. With the possible exception of the user interface (which I'll discuss next), it is possible to develop entire cross-platform systems with a single code base. When that's not possible, you should strive to wrap any platform-specific code. The end result of such a design is that, by and large, the code you write is platform-agnostic and you have small, well-isolated pockets of platform-specific code that present a uniform interface to the rest of your code. This makes testing much easier. Developers can run tests on

their development systems and, unless they touched some platform-specific code, they can be pretty sure that their changes will work on other systems.

Your build system should take care of testing on all platforms. This testing depends on your development process. Some teams run every test on every platform for every change. Some teams have layered testing policies where more expansive and exhaustive tests are run periodically (nightly or over the weekend, for example).

Some systems have to target platforms with different capabilities. It could be multiple hardware devices, different browsers, different versions of the same operating system/browser, etc. This situation can often lead to a combinatorial explosion of target platforms. You need to address it (as usual) with good design first. A common approach is to create flexible systems where capabilities are discovered dynamically or configured and integrated into the main system as plugins. From a testing point of view, this approach may reduce the testing load, too. If you have five optional capabilities, and your target platforms can support any subset of these capabilities, then you have 32 targets to test. But, if you can test each capability separately, then you only need to test five. This is an important distinction.

Another approach that can alleviate the burden is to simulate/emulate missing capabilities. For example, in the world of 3D graphics, software rendering is very common when hardware-acceleration is not available. From a testing point of view, this can be a double-edged sword. The code itself may become more streamlined and just assume a capability is available, not caring whether it's simulated or not. But, you will need to test the system behavior in both actual and simulated modes. This is especially true if you develop the simulator.

Testing the User Interface

Ah, testing the user interface…the bane of automated testing. There are several aspects of any user interface that need to be tested. In most cases, you will use some library that knows how to display widgets like buttons, text boxes, and images on the screen and allows you to hook up event handlers to events like button clicks or selection changes. If you develop your own custom UI, it is a different story and falls outside of the scope of this article. Directly testing a UI by trying to automate it is a nightmare. You have to deal with message loops, event queues, timers, input device latency, as well as take into account screen resolution and user response time. It can be done, but it's tedious and very brittle. A better approach isolates the display and low-level event handling from the management of the UI state. This approach uses an evolution of the famous MVC pattern called "Model

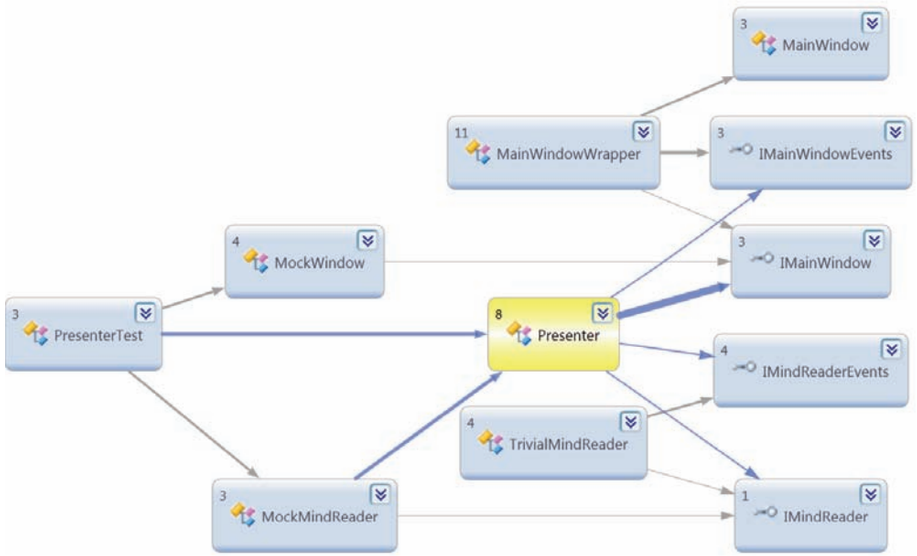

Figure 2.

View Presenter" (MVP). The models are the domain objects, which are completely agnostic of the UI. The views are a thin layer around the UI widgets (and compositions, such as windows and dialog boxes with a bunch of widgets). The presenter is the smart guy. It talks to model/domain objects and is responsible for updating the proper view when the model state changes. In addition, it also listens to events from the views, and manages the UI state. What does it buy you from a testing perspective? A complete freedom to test in isolation both your domain objects and your user interface logic and state. You rely on your widgets library to do the right thing and then validate the visual design manually.

The various class interfaces and their relationships are displayed in Figure 2. In the mind reader application, the domain or model is implemented by the `TrivialMindReader` class, which exposes the `IMindReader` interface:

```
public interface IMindReader
{
    void Read();
}
```

It also generates mind reading events through the outgoing `IMindReaderEvents` interface:

```
public interface IMindReaderEvents
{
    void OnProgress(int percent);
    void OnReadComplete(string thought);
}
```

These two interfaces isolate the domain from the rest of the application and allow mocking the domain for testing purposes, as you'll soon see. The user interface state and logic are managed by the `Presenter` class (See Listing Two). The `Presenter` accepts in its constructor an `IMainWindow` and `IMindReader` reference, and it implements the `IMainWindowEvents` and `IMindReaderEvents` interface.

Listing Two

```
namespace MindReader
{
    /// <summary>
    /// In charge of MainWindow
    /// </summary>
    public class Presenter :
        IMainWindowEvents,
        IMindReaderEvents
    {
```

```
        IMainWindow _window;
        IMindReader _reader;

        public Presenter(IMainWindow window, IMindReader reader)
        {
            _window = window;
            _reader = reader;
        }

        public void OnGoButtonClick()
        {
            _window.EnableGoButton(false);
            _window.SetThoughtText("You are thinking of...");
            _reader.Read();
        }

        public void OnKey(string key)
        {
        }

        public void OnClose()
        {
        }

        public void OnProgress(int percent)
        {
            _window.UpdteProgressBar(percent);
        }

        public void OnReadComplete(string thought)
        {
            OnProgress(100);
            _window.SetThoughtText(thought);
            _window.EnableGoButton(true);
        }
    }
}
```

The `Presenter` is responsible for the following UI state:

- When the go button is pressed, it should start a mind reading session, reset the progress bar to 0, reset the text box to the generic message: "You are thinking of…" and disable the button.
- When progress reports arrive via `OnProgress()`, it should update the progress bar.
- When `OnReadComplete()` is called, it should display the new thought, update progress to 100%, and enable the go button.

The `PresenterTest` verifies this entire sequence with the help of two mock objects in a few lines:

```
[TestMethod]
public void Test()
{
    _presenter.OnGoButtonClick();

    // The presenter should first disable the button and in
    // the end re-enable it.
    CollectionAssert.AreEqual(new List<bool>()
       { false, true }, _mockWindow.EnableGoButtonCalls);

    // The presenter should set the thought twice, first to
    // the generic message and then to the test thought
    CollectionAssert.AreEqual(new List<string>() {
        "You are thinking of...", "whatever" },
            _mockWindow.SetThoughtCalls);

    // The presenter should update the progress bar 5 times
    // (initially to 0, finally to 100)
    CollectionAssert.AreEqual(new List<int>() {
        0, 25, 50, 75, 100 }, _mockWindow.UpdateProgressCalls);
}
```

When I ran this test, I discovered a bug — I forgot to reset the progress bar to 0 in the `Presenter`. This is exactly the type of bug that may slip through the cracks and get discovered only in production.

Here is the test setup that hooks up the real `Presenter` to the mock window and the mock mind reader:

```
[TestInitialize]
   public void Setup()
   {
       _mockWindow = new MockWindow();
       _mockMindReader = new MockMindReader(new int[] {
          25, 50, 75 }, "whatever");
       _presenter = new Presenter(_mockWindow,
          _mockMindReader);
       _mockMindReader.AttachPresenter(_presenter);
   }
```

The test itself simulates the button by calling the `Presenter`'s `OnGoButtonClick()` method (from `IMainWindowEvents`).

The `MockWindow` (see Listing Three) implements the `IMainWindow` interface, but all it does is record the calls to its methods.

Listing Three

```
using MindReader;
using System.Collections.Generic;
namespace MindReaderTest
{
  public class MockWindow : IMainWindow
  {
      public List<bool> EnableGoButtonCalls = new List<bool>();
      public List<string> SetThoughtCalls = new List<string>();
      public List<int> UpdateProgressCalls = new List<int>();

      public void EnableGoButton(bool enable)
      {
          EnableGoButtonCalls.Add(enable);
      }

      public void SetThoughtText(string thought)
      {
          SetThoughtCalls.Add(thought);
      }
```

```
    public void UpdteProgressBar(int percent)
    {
        UpdateProgressCalls.Add(percent);
    }
  }
}
```

The `MockMindReader` (see Listing Four) implements the `IMainReader` interface.

Listing Four

```
using System;
using MindReader;
using System.Collections.Generic;

namespace MindReaderTest
{
    public class MockMindReader : IMindReader

    {
        int[] _progressReports;
        string _thought;
        Presenter _presenter;

        public MockMindReader(int[] progressReports,
            string thought)
        {
            _progressReports = progressReports;
            _thought = thought;
        }

        public void AttachPresenter(Presenter presenter)
        {
            _presenter = presenter;
        }

        public void Read()
        {
            foreach (int progress in _progressReports)
            {
                _presenter.OnProgress(progress);
            }

            _presenter.OnReadComplete(_thought);
        }
    }
}
```

The `MockMindReader` also provides a canned mind reading session by accepting in its constructor a list of progress reports and a hard-coded thought:

```
public MockMindReader(int[] progressReports, string thought)
{
    _progressReports = progressReports;
    _thought = thought;
}
```

Later, the test attaches the `Presenter` to it as a sink. When the `Presenter` calls the `Read()` method (in response to the simulated button click), the mock mind reader sends its canned progress reports via `OnProgress()` and finally calls `OnReadComplete()`. The `Presenter` calls the appropriate methods on the mock main window, which records everything. Finally, the test verifies that the correct methods were invoked by checking the mock window's recorded calls.

This whole setup took very little time to write. It allows full testing of the UI management state, without dealing with an actual hard-to-test UI, and can be extended or modified easily when the real UI changes (and it will, count on it).

The steps I applied to enable testing transcend .NET, WPF, and the user interface:

- Access all dependencies via interfaces (`IMainWindow`, `IThoughtReader`).

- Mock dependencies with simple mock objects that capture the state and flow you want to test.
- Hook up the object/system under test to all its mocked dependencies.
- Run the object/system through the test scenarios.
- Verify the object/system behaves as expected.

Testing Networking Code

Networking has become pervasive. Today, people often invoke a Web service, where they used to add a library to their application or, god forbid, write some code themselves. Designing a safe, robust, and performant connected system is much more complicated than a standalone system. The reason is that you have to handle many more failure modes and a lot of stuff that is not related to your code functionality, such as security and authentication. The remote system might disappear, slow down, or change at any moment, including in the middle of sending data or halfway through a complicated handshake. The reverse is true if your system is the remote service. How do you support old clients? How do you make sure your system scales and is highly available? How do you make sure attackers don't steal your data or bring your system down? These are important concerns and require a lot of engineering. To make sure you've got it right, you have to be able to simulate and test all these scenarios.

The first rule is to isolate your system code from the low-level networking. This is very similar to the MVP pattern for the user interface. You want your domain objects to be implemented in a network-agnostic fashion. But, you have to be careful when you design your interfaces, because chatty or blocking interfaces won't cut it when you have to send a lot of data over the wire. In general, for high-performance networking, you would want asynchronous interfaces. I'll talk more about asynchronous code later.

To demonstrate the process, I created a little mind reading TCP server. The protocol is very simple. The server waits for a client to send the command: `"Read thought remotely."` The server responds with a progress report in the format `"progress:"` (for example. `"progress:25"`), and finally, it sends the thought itself as `thought:` (for example, `"thought:I think therefore I am"`). The server is not very interesting, but it can handle multiple concurrent clients. I also created a `Client` class, which exposes the `IMindReader` interface and sends `IMindReaderEvents` to a sink (see Listing Five) .

Listing Five

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;
using System.Threading;
using System.IO;

namespace MindReader.Client
{
    public class Client : IMindReader
    {
        string _hostname;
        int _port;
        IMindReaderEvents _sink;

        public Client(string hostname, int port)
        {
            _hostname = hostname;
            _port = port;
        }
```

```
public void AttachSink(IMindReaderEvents sink)
{
    _sink = _sink;
}

public void Read()
{
    new Thread(() =>
    {
        var client = new TcpClient();
        client.Connect(_hostname, _port);

        // Wait for connection
        while (!client.Connected)
        {
            Thread.Sleep(100);
        }

        var s = client.GetStream();

        // Send read command to server
       var bytes = ASCIIEncoding.ASCII.GetBytes
           ("Read Thought Remotely\r\n");
        s.Write(bytes, 0, bytes.Length);

    // Read progress reports and finally the thought
        var reader = new StreamReader(s);
        while (true)
        {
            var line = reader.ReadLine();
            if (line.StartsWith("progress:"))
            {
                var progress = int.Parse(line.
                   Split(':')[1]);
                _sink.OnProgress(progress);
            }

            if (line.StartsWith("thought:"))
            {
                var thought = line.Split(':')[1];
                _sink.OnReadComplete(thought);
            }
```

```
                }

            }).Start();
        }
    }
}
```

This `Client` class is a drop-in replacement for the local mind reader and requires a single line change in the factory to hook all the components together:

```
var mainWindow = new MainWindow();
var wrapper = new MainWindowWrapper(mainWindow);

//var mindReader = new TrivialMindReader();
var mindReader = new cli.Client("localhost", 54321);

var presenter = new Presenter(wrapper, mindReader);
wrapper.AttachSink(presenter);
mindReader.AttachSink(presenter);

mainWindow.Show();
```

The `IMindReader` and `IMindReaderEvents` were designed for asynchronous interactions initially, so this is a very easy transition. The `Client` runs on a separate thread and uses blocking APIs to interact with the server. In this example, I created two simple test cases to verify the entire session. I have a simple `Sink` class that implements the `IMindReaderEvents` and collects the calls (similar to the `MockWindow` in the `PresenterTest`):

```
class Sink : IMindReaderEvents
{
    public List<int> OnProgressCalls = new List<int>();
    public List<string> OnReadCompleteCalls =
        new List<string>();

    public void  OnProgress(int percent)
    {
```

```
        OnProgressCalls.Add(percent);
    }

    public void  OnReadComplete(string thought)
    {
        OnReadCompleteCalls.Add(thought);
    }
}
```

I pass `Sink` objects to the `Client` and call its `Read()` method to start a mind reading session against the server:

```
[TestMethod]
public void SingleClient_Test()
{
    var cli = new Client(_hostname, _port);
    cli.AttachSink(_sink);
    cli.Read();
    while (_sink.OnReadCompleteCalls.Count == 0)
    {
        Thread.Sleep(100);
    }
    Assert.AreEqual(3, _sink.OnProgressCalls.Count);
    Assert.AreEqual("I think therefore I am", _sink.
        OnReadCompleteCalls[0]);
}
```

I also created a test for two concurrent clients:

```
[TestMethod]
public void TwoConcurrentClients_Test()
{
    var sink1 = new Sink();
    var sink2 = new Sink();

    var cli1 = new Client(_hostname, _port);
    cli1.AttachSink(sink1);
    var cli2 = new Client(_hostname, _port);
    cli2.AttachSink(sink2);

    // Start 2 mind reading sessions
    cli1.Read();
    cli2.Read();

    // Wait until both clients got a reading
    while (sink1.OnReadCompleteCalls.Count * sink2.
        OnReadCompleteCalls.Count == 0)
    {
        Thread.Sleep(100);
    }
    Assert.AreEqual(3, sink1.OnProgressCalls.Count);
    Assert.AreEqual(3, sink2.OnProgressCalls.Count);
    Assert.AreEqual("I think therefore I am", sink1.
        OnReadCompleteCalls[0]);
    Assert.AreEqual("I think therefore I am", sink2.
        OnReadCompleteCalls[0]);
}
```

Mock Servers and Clients

In networking code, it is common to talk about clients and servers. There are many topologies and protocols, but generally, clients are configured or know how to find their server/s and initiate communication. In the ideal testing scenario, you completely isolate your code via interfaces. If your code is a server, you create a mock client and, in the test, your mock client will start talking to your server. If your code is a client, you create a mock server and, in the test, your code will start talking to the mock server. Sometimes the same code is both a server to some clients and a client of other servers/services. In such cases, you will provide multiple mocks and test the server roles and the client roles separately.

The fun thing about testing against a mock server or client is that you have complete control and can easily simulate any failures, such as an unreachable server, sudden disconnect at any point, missing responses, corrupt responses, and slow responses. You can even insert breakpoints

and step through each interaction in your code and in the mock. When testing against real servers/clients, it is often very difficult to create these many error conditions and behaviors. Mock servers are also great for emulating clusters, where you dynamically reconfigure your mock cluster on the fly from test to test or even in the middle of a test.

The MindReader client and server repeat the same concepts precisely. Instead of working directly with the .NET `TcpListener` and `TcpClient`, you would create `ITcpListener`, `ITcpListenerEvents`, and `ITcpClient` interfaces (if you want a non-blocking client, then add `ITcpClientEvents`). Then add a `TcpListenerWrapper` and `TcpClientWrapper`, hook everything up, and pass it to the MindReader client and server. From there, you would be in a position to simulate any TCP networking scenario.

Localhost Test Servers

Sometimes mocking a server or a client faithfully is complicated. In these cases, the next best thing is a local test server (or client). You don't have full visibility into all the interactions (usually only via log files), but at least the connection is greatly simplified. You can start and kill test servers on your machine without impacting anyone else. Your code also interacts with a real server, which gives you confidence that it will work in the wild, too. The debugging experience is usually degraded in such cases.

When testing the MindReader networking scenario, I ran the server locally. All it takes from the client side is to make sure the connection information is not hard-coded:

```
[TestClass]
public class ClientServerTest
{
    Sink _sink;
    string _hostname = "localhost";
    int _port = 54321;

    ...

    [TestMethod]
    public void SingleClient_Test()
    {
        var cli = new Client(_hostname, _port);
        cli.AttachSink(_sink);
        ...
    }
```

Dedicated Test Servers and Environments

Local servers can be hard to configure (for example, when the server depends on many other services that need to be mocked or configured). Often, you will need to test your code against multiple servers and/or clients. This is where dedicated test servers or full-fledged test environments come into play. Test environments try to mimic the production environment to some degree. There are two ways to work against a test environment:

- Run your code locally and connect to the test environment. This option is usually good for testing client code.
- Deploy your code to the test environment. Test via the exposed API (REST, SOAP, plain HTTP, custom TCP/UDP). This option is usually good for testing server code.

Test environments are useful for integration testing. The main thing to pay attention to is that the test environment doesn't deviate from the production environment: You must have a process in place (ideally automated). The other concern is version control of

different components and resources under test. Test environments are a scarce resource shared by many developers. If multiple developers deploy their latest code at the same time, they can easily step on each other's toes and break the test environment for everyone.

Latency, Throughput, and Stress Testing

When dealing with distributed systems, two crucial metrics are throughput and latency. Throughput defines how much data can be transferred over the wire, and latency defines how fast can you respond to a request. Due to the nature of distributed systems, these attributes depend on many factors, such as the hardware specifications of different nodes, the bandwidth available between nodes, the protocols in use, and the load on your system. There is also often an interplay between latency and throughput. In general, when the load on the system increases the latency will start to go up, and the throughput may degrade, too. It is very difficult, if not impossible, to predict the behavior of a distributed system under heavy load because it is non-linear. Various thresholds and interactions kick in at different loads. Knowing how much load your system can handle is crucial to ensuring quality of service and scalability.

The way to acquire this knowledge to test the system under various simulated loads and observe its behavior. Sometimes, you isolate different components of the system and stress test them independently. There are many tools available for stress testing, but for sophisticated systems, you would want to create your own stress tool, which can simulate realistic loads and patterns of interactions.

Limited Production Testing and Gradual Deployment

Stress/load testing is important, but it is not enough. Another good technique for testing code changes is to deploy the new version of the code to a small number of servers side-by-side with the existing system and observe the behavior of the new servers. If anything goes wrong, you switch back to the existing version and analyze the problem. If everything runs OK, you can transition more and more servers to the new system. This gradual deployment is an effective technique, but is a complicated process with multiple steps, especially if code changes involve database schema, protocol, or file formats.

Testing Persistent Storage

Persistent storage is another area that is not always easy to test. Most likely, you will not be able to run tests against actual production storage (especially not tests that modify the data). This means that you will have to test against test data, which comes with its own challenges involving synchronization with production data. There are two levels here:

- The DB schema or file formats used for persistence
- The actual content

When you keep persistent test data, each level (or both) may get out of sync. You have to be very diligent to stay ahead of the curve. The first line of attack is as usual — abstracting away data access and testing your code against mock data sources. This will enable running lots of tests quickly without actually accessing the real data store. In some cases, you may need to test more sophisticated data access scenarios involving caching at different levels. The key is to be able to control every aspect of the data access process that you want to test. As long as you interact with dependencies through interfaces and use dependency injection, you're good to go.

Testing Asynchronous Code

Asynchronous code, often found related to networking or UI, is probably the most difficult code to test. When you invoke an asynchronous operation, the flow of control returns immediately to the calling code, but the operation hasn't completed yet. In your test code, you need to wait for the operation to complete, then proceed as usual, checking the result, state, and side effects. Asynchronous APIs fall into three broad categories:

- Callback-based APIs
- Future-based APIs
- Queue-based APIs

With Callback-based APIs, you provide a callback function to the asynchronous operation; and when it completes, your callback function is called. Future-based APIs return an object you can query periodically to check whether the operation has completed. Queue-based APIs deposit the response in a queue you can poll.

In your test, you want to busy-wait for the async operation to complete either by setting a flag in your callback function or by directly checking the future object or queue and then carrying on as usual.

The events interfaces I favor most of the time are a form of Callback-based APIs. In my networking test, I used the following code to wait for the mind reading session to complete:

```
while (_sink.OnReadCompleteCalls.Count == 0)
{
    Thread.Sleep(100);
}
```

This is not ideal. If the code fails and `OnReadComplete()` is never called, the test will hang. A very common and simple approach is just sleeping for a while:

```
// Sleep for 5 seconds to give the operation enough time to
// complete
Thread.Sleep(5000);
```

This is pretty bad, too. If the operation completes in a jiffy, you still have to wait for five seconds; and if you run a battery of tests, it can slow your test run significantly. I recently tested a lot of code where I had to kill, start, and reconfigure a remote RabbitMQ cluster in about 50 test cases. If I used hard-coded sleeps to wait for each operation to complete, I wouldn't be sitting here writing this article.

The best approach is to do periodic checks with a timeout. The pseudo code looks something like the following:

```
timeout = Now + 5 seconds
while (!ok and Now() < timeout)
{
    ok = is operation complete?
    sleep 100 milliseconds;
}

if (ok)
{
    // good
}
else
{
    // oh-oh. operation times out
}
```

This is cumbersome code to write every time you want to wait for an operation to complete. In a future article, I will demonstrate a neat class that compresses the code to the following snippet:

```
var ok = Wait.For(3000, () =>
    { _sink.OnReadCompleteCalls.Count > 0 });
```

This statement will wait for three seconds, periodically checking the expression, and will bail out immediately when it is true. If it is never true after three seconds, it will exit anyway and 'ok' will be false.

"The problem with polyglot systems is that error reporting is often not streamlined"

Some asynchronous code relies on timers and timeouts. You should make sure these timeouts are configurable, so your test doesn't have to wait 10 minutes to discover what your code is doing when some operation times out.

Testing Multithreaded or Multiprocess Code

Multithreaded code with low-level locks and multiple threads modifying shared resources is a nightmare to test. This should give you pause. In every case I have ever encountered where multithreaded code was too complex to test, the code itself was just too complex, period. There were nasty bugs related to fine-grained locking (not to mention heroic attempts at lock-free algorithms). Consider modern approaches like message passing and share-nothing designs. If you have to manage low-level concurrency, try to minimize the surface area — code review and analyze it, and then bombard the code with fuzz testing. I don't recommend trying to mock threads. You will never be able to simulate all the ways real threads can destroy your buggy code and it will just give you a false sense of security.

Testing multiprocess code poses its own challenges. It is similar to multithreaded code in that some operations happen outside your test control flow. The difference is that it is harder to check the state of objects in other processes. There are many inter-process communication (IPC) mechanisms, and you should look into them to get visibility into the state of processes. You can also mock processes (just like you mock servers when testing networking code). If you want to wait for some operation in another process to complete, you can use a file as a lock (each process tries to get an exclusive access to the locked file).

Testing Systems Created with Multiple Programming Languages

Many complex systems have components written in different programming languages. As long as the interaction between the components is over a network using common protocols, this is not very interesting. However, sometimes multiple programming languages interact in a more direct way through language bindings. For example, many dynamically typed languages (scripting languages) like Python and Ruby provide a C extension API that allows integration with any C library. The other direction is also very common, where you embed a scripting language in a C/C++ program and allow users to script various items or write plugins in a dynamic language. For example, Lua is often embedded in game engines. JavaScript, by way of Google Chrome's V8, can also play with C. The Java Virtual Machine is a powerhouse when it comes to multiple language, and in recent years, it has become fashionable to target the JVM from many languages like Scala, Clojure, Jython, and JRuby. Java itself always had the Java native interface (JNI) to interact with C. But the poster child of cross-language development is definitely the .NET framework, which was designed from the start as a multi-language framework. Visual Studio support

for multi-language development is superb and even allows you to debug and step through different languages.

The problem with polyglot systems is that error reporting is often not streamlined and failures across the language boundaries can be masked and fail quietly (or the program can just exit). To test such systems, you need to adopt a systematic approach to collect information form the both sides of the language boundary. Designing language bindings is a black art. Marshaling data types is often the main culprit. If you ever tried to Swig a cool C++ class with a bunch STL types and custom templates, you would never forget it. Swig can generate C/C++ bindings to almost any language and is very powerful. But as the saying goes: "With great power comes great confusion." If you use Swig/C++, I highly recommend you keep the bindings themselves as just a thin veneer and avoid any extension.

Testing polyglot systems with a core C/C++ engine and a layer of bindings is best done in layers. In general, the native code should be completely agnostic to the fact that it can be accessed from other languages. You should be able to unit test the core engine in C/C++. If you have a layer of scripting code (say, Python) that exercises the native code via bindings, consider mocking the native code. This is usually very easy in dynamic languages via monkey patching.

Sometimes, the polyglot nature of the system plays to your advantage. Writing and running tests is much easier in Python than in C++, and you can find many fine Python unit test frameworks. You don't have to compile and link your tests, and you can even explore your objects in a REPL.

Testing Failure Code Paths and Error Handling

One of the great challenges in complex systems is testing failures and error conditions. The major difficulty is that for every successful interaction, there is usually a single path through the system: an input event arrives (network packet, button click, a message from another process, a timer expires), it is processed, and the system returns a result and/or records the result. Everybody knows how to test the happy path. But every step along the way might fail. For each failure, there is a different response and possible fallback. Suddenly, the neat interaction branches out into a Hydra and you need to test each head. Here are a few common failures you've probably never tested comprehensively:

- Input file or directory doesn't exist / has wrong permissions / contains corrupt data
- Out of memory on any operation
- Out of disk space
- Network connection drops out of the blue
- An external component you call hangs
- First name string is 30MB (someone is doing your buffer overflow testing for you!)

If you neglect to test how your code behaves in these situations, then you don't know how it will behave at the most critical time. For example, your code might need to alert the operations team when the response time of some server is greater than five seconds. Now, consider the consequences of a bug in this piece of code…

OK, you get the picture. Everything can fail and the error handling code itself should be tested. But how do you address the seemingly combinatorial explosion of failure code paths? It all starts with proper design. The same old principles of modular, loosely coupled, highly co-

hesive components will serve you well. Identify the risk of each component and design an error-handling policy. For example, for sensitive data, use transactions to ensure you don't end up in inconsistent state. For components that must always be running, build in redundancy and switch-over capability. For large subsystems that expose an API, make sure to control and safeguard the surface area so no crashes or unexpected exceptions escape to the user (it's OK to throw exceptions as part of the design).

Now that your system is a collection of components with well-defined interactions, you can systematically test how the system behaves if any component fails in one of those well-defined ways. It sounds like a lot of work and it is. The only solace is that this approach pushes out really excellent APIs: very small, with minimal interactions, as few side effects as possible, and few externally exposed failures.

Let's examine how to test for errors in the sample MindReader application. My goal is to test how the `Presenter` handles exceptions. This is very easy to do with the mock mind reader. The test passes null references to progress reports and the thought. This will cause the mock mind reader to throw a null reference exception in the `Read()` method:

```
[TestMethod]
public void MindReader_CrashTest()
{
    var mockMindReader = new MockMindReader(null, null);
    var presenter = new Presenter(_mockWindow,
       mockMindReader);
    mockMindReader.AttachPresenter(presenter);

    presenter.OnGoButtonClick();
}
```

The question was, "How does the `Presenter` handle a MindReader exception?" and the answer is that it doesn't. It will crash and burn. The

test exposed the fragility of the `Presenter` and now you can come up with a policy (catch the exception and display a message to the user, retry a couple of times quietly, or log the exception and exit). If there is some error-handling mechanism in place, the test can verify that it is indeed invoked properly.

At the tactical level, I recommend using exceptions and letting them propagate to a point where they can be handled properly. Make sure that the state of the system stays intact in the face of an error. Utilize design patterns (such as RAII) and clean up after yourself. If you use a language like C or Go, where there are no exceptions, you just have to be extra diligent.

Conclusion

Today's software is growing more complex, but with rare exceptions, it is not tested properly. Even software development processes that emphasize testing usually have only limited unit tests. This is often a deliberate cost-benefit decision due to the enormous challenges of deep testing complex systems. The trade-off is time to market vs. quality. In this article, I demonstrated how to deeply test the most challenging aspects of complex systems by relying on tried and true design principles. By using factories, interfaces, and events, any software component or sub-system can be isolated and tested using mocked dependencies.

— Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/C#/Python/Java with emphasis on large-scale distributed systems, and is a long-time contributor to **Dr. Dobb's.**

Comment

Testing Browser Apps

The QUnit framework makes it easy to test Web apps directly in the browser. By showing the red/green results in the browser with links to the failing tests, it makes it possible to write, test, and correct quickly.

By Andy Sylvester

Dr. Dobb's has covered HMTL5 development on many occasions, as well as how to write extensions to Google Chrome (http://www.drdobbs.com/mobile//232602420). This article extends that coverage by examining how to test browser apps. The testing tool I'll use is QUnit (http://qunitjs.com), which is a convenient unit-testing framework that first saw light of day as part of the jQuery JavaScript framework. It is now an independent product, no longer dependent on jQuery. But like jQuery, it greatly facilitates useful work in JavaScript.

Setting Up QUnit to Run in a Web Page

To get started with QUnit, download the files qunit.js (http://build-browserapps.com/files/chrome/qunit.js) and qunit.css (http://build-browserapps.com/files/chrome/qunit.css), and copy them to a new folder in your ChromeProjects folder. Qunit.js contains the functions that make up the test framework, and quinit.css provides styles and other helpers to present the results of your QUnit tests.

To start, let us demonstrate a simple test using a standalone HTML file. We will create a simple function and test if the function returns a result of TRUE. In QUnit, the basic test function is named `ok`. It takes two arguments: The first is the code or condition to be tested, and the second is a text string that appears in the results of the test. An example could be:

```
ok (2 + 2 == 4, "two plus two equals four");
```

The `ok` function evaluates the expression in the first argument, and displays in the test results whether the test passed or failed.

Simple Tests

Here is a simple Web page containing the expression to be tested and the test environment in which to observe the test results (save this as qunittest01.html):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
```

```
<link rel="stylesheet" href="qunit.css">
<script src="qunit.js"></script>
<script>
test("hello", function() {
      ok(2 + 2 == 4, "Hello, world");
});
</script>
<h1 id="qunit-header">QUnit Hello World</h1>
<h2 id="qunit-banner"></h2>
<ol id="qunit-tests"></ol>
```

Open the file qunittest01.html in a browser, and you should see the results shown in Figure 1.

Clicking on the "hello" link in the page displays each of the test results.

This simple Web page has links to the QUnit CSS file that formats the output of the test, a link to the QUnit JavaScript file containing the test framework functions, and the JavaScript for the test. The script includes a call to the `test` function, which is how tests in QUnit are grouped. The first argument of the `test` function is a text string that is the name of the group of tests. The second argument, a function, contains the tests in this group. In our first example, there is only one test. However, a test group can contain as many tests as you wish. We could add more tests for other algebraic expressions as follows:

```
ok (2 + 2 == 4, "two plus two equals four");
ok (2 + 5 == 7, "two plus five equals seven");
ok (5 + 5 == 10, "five plus five equals ten");
```

Save the following Web page as qunittest02.html:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<link rel="stylesheet" href="qunit.css">
```

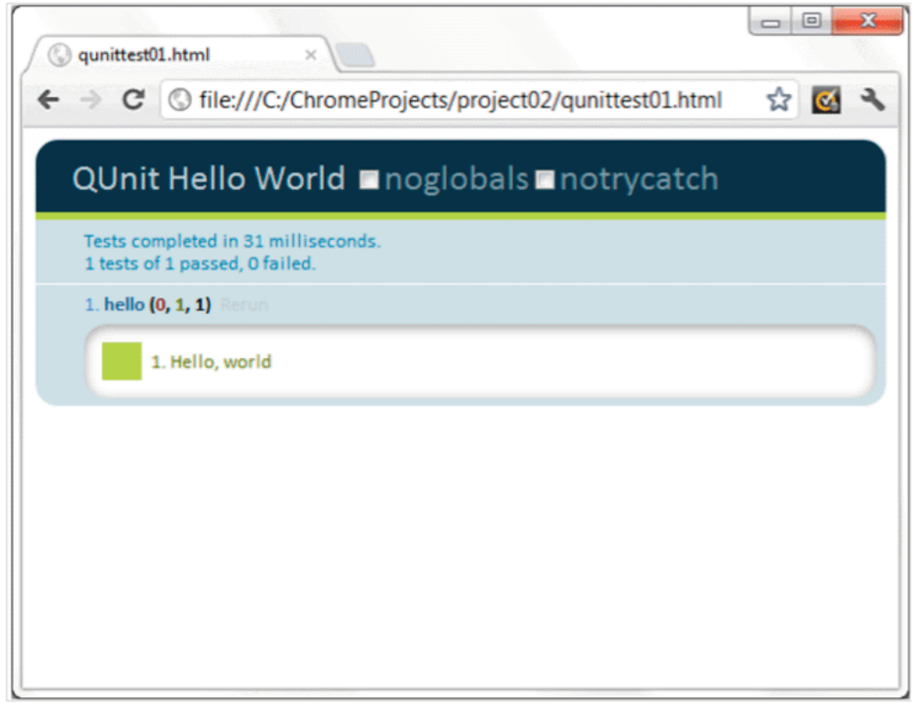

Figure 1.

```
<script src="qunit.js"></script>
<script>
test("hello", function() {
   ok(2 + 2 == 4, "two plus two equals four");
   ok(2 + 5 == 7, "two plus five equals seven");
   ok(5 + 5 == 10, "five plus five equals ten");
});
</script>
<h1 id="qunit-header">QUnit Hello World</h1>
<h2 id="qunit-banner"></h2>
<ol id="qunit-tests"></ol>
```

Open qunittest02.html in your Web browser and you should see the results shown in Figure 2.

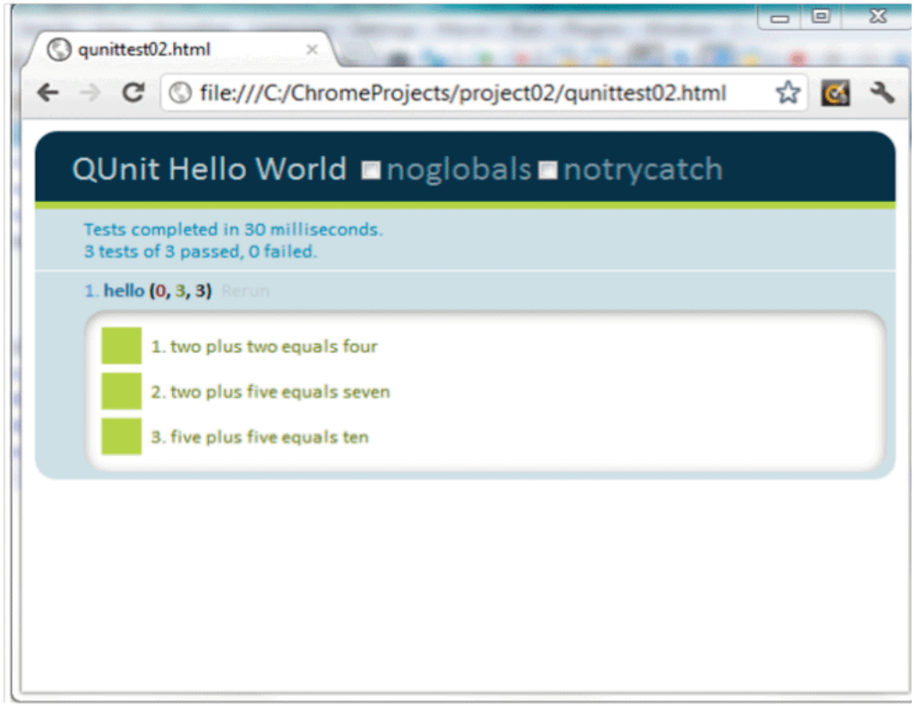

Figure 2.

When you click on the "hello" link in the page, each of the test results are displayed.

Add Tests for Text Field/Button Part of Hello World Example

Now that you have seen some examples of algebraic tests, let's move on to using QUnit to perform tests of DOM elements. A useful feature of QUnit is the ability to add test HTML that your tests can use to the page. QUnit uses a `div` with an ID of `"qunit-fixture"` to designate the test HTML code. An example might look like this:

```
<div id="qunit-fixture">
This is a test
</div> <!-- qunit fixture markup -->
```

To show how this works, let's add some additional logic to the testing function to check the text in the fixture HTML.

```
test("hello", function() {
   // Find element with id="qunit-fixture"
   var testMessage = document.getElementById("qunit-fixture");
   var testMessageresult = testMessage.innerHTML;
   ok(2 + 2 == 4, "two plus two equals four");
   ok(2 + 5 == 7, "two plus five equals seven");
   ok(5 + 5 == 10, "five plus five equals ten");
   equal(testMessageresult, "This is a test"
      "text in div is correct");
});
```

In this modified function, logic has been added to save the HTML element with the ID of `qunit-fixture` to a variable, and to extract the HTML within that element. A new type of test was added (`equal`) that compares the actual and expected values of the item being tested, along with the description of what the test is doing. Here is what the new file looks like (save as qunittest03.html):

```
<link rel="stylesheet" href="qunit.css">
<script src="qunit.js"></script>
<script>
test("hello", function() {
   // Find element with id="qunit-fixture"
   var testMessage = document.getElementById("qunit-fixture");
   var testMessageresult = testMessage.innerHTML;
   ok(2 + 2 == 4, "two plus two equals four");
   ok(2 + 5 == 7, "two plus five equals seven");
   ok(5 + 5 == 10, "five plus five equals ten");
   equal(testMessageresult,
```

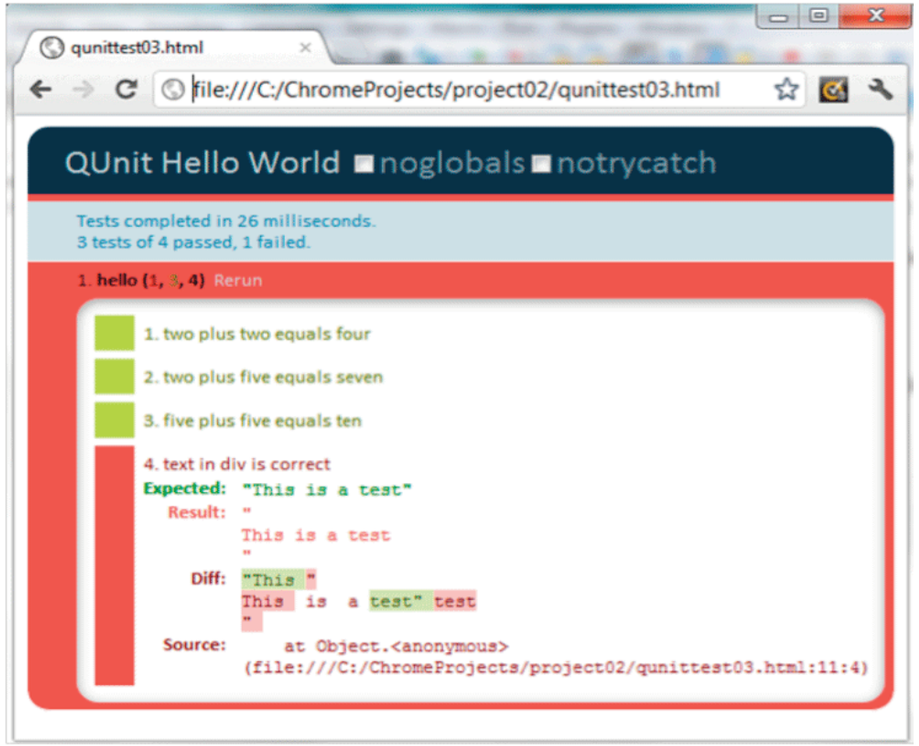

Figure 3.

```
    "This is a test", "text in div is correct");
});
</script>
<h1 id="qunit-header">QUnit Hello World</h1>
<h2 id="qunit-banner"></h2>
<ol id="qunit-tests"></ol>
<div id="qunit-fixture">
This is a test
</div> <!-- qunit fixture markup -->
```

When this HTML file is loaded, the results in Figure 3 appear. As you can see, the new test fails and a red area is shown in the part of the page with the test results. Looking at the results, it appears that when the data within the `div` element is extracted, a quotation mark is added for each carriage return. Let's try making a change in the `"qunit-fixture"` test HTML code to see whether that has an effect. Put all of the logic on one line, making it look like this:

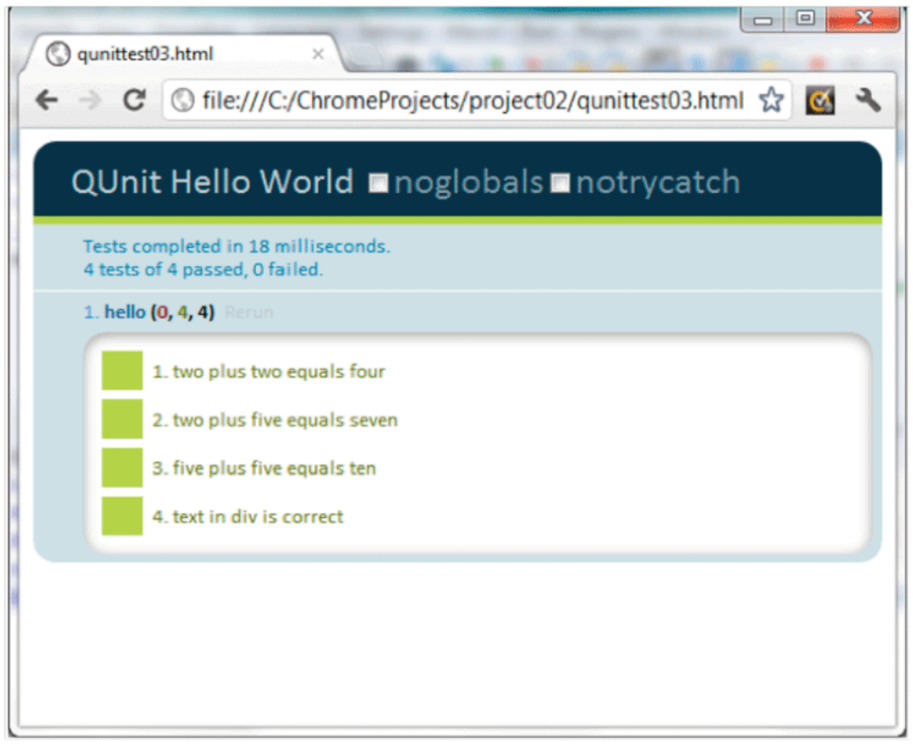

Figure 4.

```
<div id="qunit-fixture">This is a test</div>
   <!-- qunit fixture markup -->
```

When this change is added to the app, the test results now look like Figure 4.

Next, let's add another element to the test HTML logic (an input text field). Save qunittest03.html as qunittest04.html, and replace the `qunit-fixture` logic with the following:

```
<div id="test-input">
<form><input type="text" size="30" name="xlInput"
        id="xlInputTest" class="xlarge" value="Andy"></form>
</div> <!-- test input markup -->
```

This logic reuses the input text field from the application, changes the ID to `"xlInputTest,"` and adds a value attribute equal to `"Andy."` Next, let's modify the testing function as follows:

```
test("hello", function() {
   // Find element with id="xlInputTest"
   var testMessage = document.getElementById("xlInputTest");
   var testMessageresult = testMessage.value;
   ok(2 + 2 == 4, "two plus two equals four");
   ok(2 + 5 == 7, "two plus five equals seven");
   ok(5 + 5 == 10, "five plus five equals ten");
   equal(testMessageresult, "Andy", "text in div is correct");
});
```

Here, we modified the logic to get the HTML element associated with the input text field that was added, to get the value of that field, and to compare that value to the expected value "Andy." After adding these changes to the app, we see the test results in Figure 5.

Let's make one more change to set a div element based on the test input field value, see that value be displayed just as if it were in the actual text field, and perform a test to check the value. Save

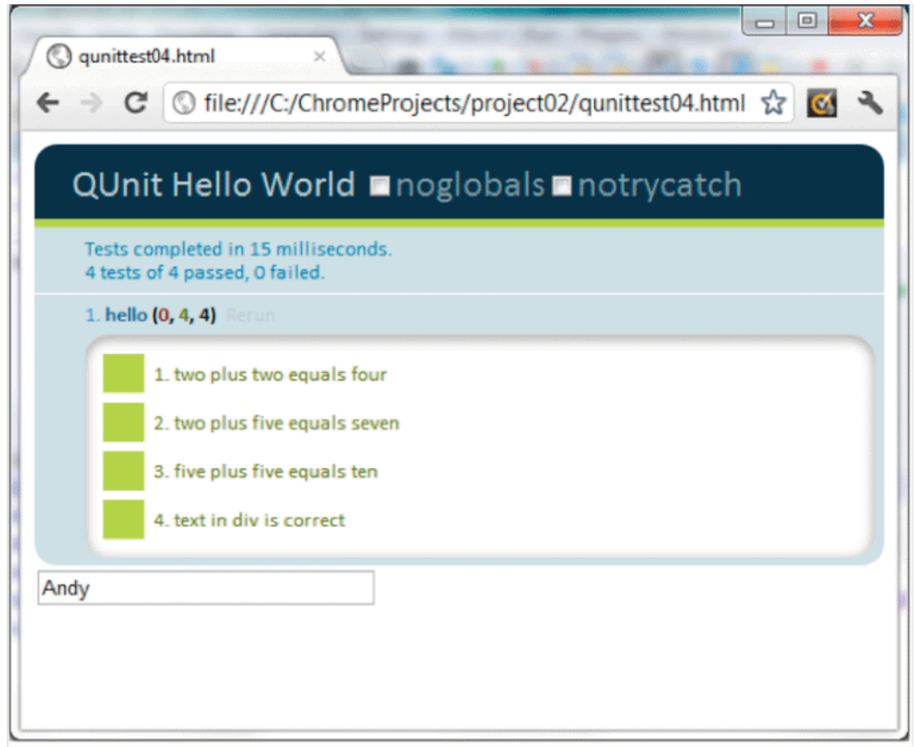

Figure 5.

qunittest04.html as qunittest05.html, then update the testing function as follows:

```
test("hello", function() {
   // Find element with id="xlInputTest"
   var testMessage = document.getElementById("xlInputTest");
   var testMessageresult = testMessage.value;
   // Find element with id="test"
   var showMessagetest = document.getElementById("test");
   showMessagetest.innerHTML = "Hello, " +
```

```
    testMessageresult; // Display message
  ok(2 + 2 == 4, "two plus two equals four");
  ok(2 + 5 == 7, "two plus five equals seven");
  ok(5 + 5 == 10, "five plus five equals ten");
  equal(testMessageresult, "Andy",
    "text in input field is correct");
  equal(showMessagetest.innerHTML, "Hello, Andy",
    "text in message is correct");
});
```

We will also add a `div` with an ID of `"test"` to the bottom of the file as follows:

```
<div id="test">
</div><!-- /test -->
```

When we add these changes to the application, we get the test results shown in Figure 6.

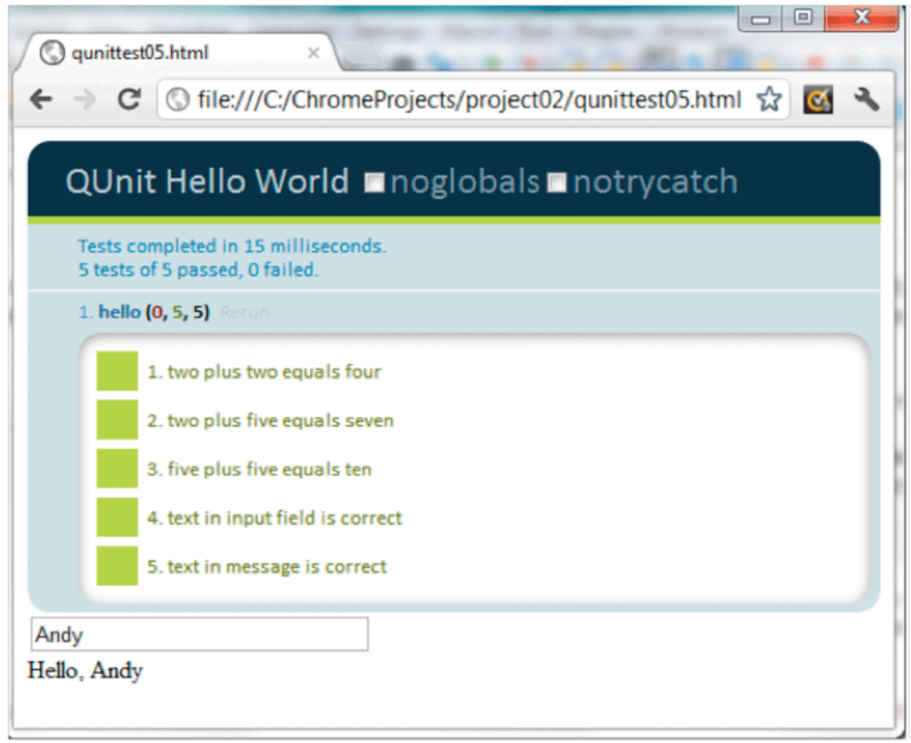

Figure 6.

Conclusion

This article has given a quick overview of how easy it is to test browser apps and have the results show in the browser itself. Several additional resources are available to continue your exploration. The NetTuts tutorial (http://is.gd/FWQR20) gives an excellent overview of QUnit with simple examples. The article "Automating JavaScript Testing with QUnit" on the Microsoft Developer Network (http://msdn.microsoft.com/en-us/scriptjunkie/gg749824.aspx) goes into more detail on testing DOM elements and asynchronous testing. It is a good follow-on piece to the NetTuts tutorial. Finally, the presentation by Ben Alman on QUnit (http://benalman.com/talks/unit-testing-qunit.html) has links to some good example code, and is also a neat HTML-based presentation (press the space bar to move to the next slide).

Enjoy!

— Andy Sylvester is an author and software developer. This article was adapted from his book, **Building Browser Apps with Google Chrome** *(http://www.buildbrowser-apps.com/chromeappbook), which is available for free in HTML and can be purchased inexpensively in other formats.*

Comment

From the Vault

Perl Testing

Testing is the cornerstone of robust software development, and Perl supports a flexible and powerful testing capability.

— *DDJ*

By Peter Scott

A crash is when your competitor's program dies. When your program dies, it is an 'idiosyncrasy.' Frequently, crashes are followed with a message like 'ID 02.' 'ID' is an abbreviation for idiosyncrasy and the number that follows indicates how many more months of testing the product should have had." — Guy Kawasaki

Testing is the cornerstone of robust software development, and Perl supports a flexible and powerful testing capability. In this article, I examine the testing mechanisms available to Perl programmers, focusing on the Extreme Programming (XP) Test First principle. Test First turns the traditional waterfall methodology on its head by saying that test writing precedes not only coding but even design, and that tests come straight out of requirements. In fact, the tests are a formal expression of the requirements.

This approach makes enormous sense. A requirement is not a real requirement unless it is testable. "The product shall be user friendly" is not testable, while "the system shall process 100 record updates within 500ms" is. So why interpose a huge hiatus of design and implementation between writing the requirements and finding out whether they're testable? Moreover, a full-fledged regression and unit test suite provides the safety net that lets you perform feats of derring-do on the high wire of development, safe in the knowledge that at every step you can instantly check whether the code passes the tests no matter what massive changes you may have just wrought upon it.

Clearly, the developers of Perl have embraced the XP testing philosophy. Between 1997 and 2004, the number of tests for the core Perl distribution itself mushroomed from 5000 to 70,000.

Simple to Get Started

Being lazy means never having to say you've got carpal tunnel syndrome. The test module designers applied the principle of Huffman coding to the API; that is, the most common functions you call should have the shortest names to optimize typing and reading. The most common test function is simply called `ok()`, which is vastly more appealing than, say, `TestSocketListenerConnectionFailure()`. If you're impatient (another fundamental Perl virtue), you can write a test just by creating a program containing, for instance:

```
use Test::Simple tests => 1;
ok(sqrt(16) == 4, "sqrt(16) works");
```

As this shows, the `ok()` function is exported by the `Test::Simple` module that comes with Perl (since Perl 5.8.0, and before that in CPAN) and so is close to hand.

All that the `ok()` function does is test its first argument and print out "ok" or "not ok," depending on whether it is true. You're probably thinking, "I hardly need anyone to write a module to do that." But from such little test acorns do mighty regression suite oaks sprout. The `ok()` function also outputs the test label in the second argument and a running count of tests it has run. This is where the tests argument to `Test::Simple` comes in — it specifies how many tests we intend to run, so that `Test::Simple` can warn you if it sees a different number of calls to `ok()`. The output of a typical test program run looks like Listing One.

Listing One

```
1..12
ok 1 - regex search
ok 2 - regex search
ok 3 - regex search
ok 4 - regex store lives
ok 5 - regex store
ok 6 - Store non-matching regex croaks
ok 7 - Keys on complex store
ok 8 - Hash on complex store
ok 9 - lol regex store lives
ok 10 - lol regex store
ok 11 - Chained assignment
ok 12 - Multiple match croaks
```

The first line shows the range of tests expected to be run; the significance of this will be seen later. Any error output from the program is passed through.

All this serves to show that the testing of Perl programs fits the same philosophy as programming Perl: Simple tasks should be simple to achieve. In practice, most testing uses the `Test::More` module, which exports a number of more useful functions, including:

- `is($expr, $value, $tag)` same as `ok($expr eq $value, $tag)`.
- `like($expr, $regex, $tag)`, same as `ok($expr =~ $regex, $tag)`.
- `is_deeply($struct1, $struct2, $tag)`, compares arbitrarily deep structures.
- `isa_ok($object, $class)`, verifies that an object is a member of a given class.
- `use_ok($module, @imports)`, test that a module can be loaded.

Note again the Huffman principle on names of commonly used functions. Even though several of the functions are syntactic sugar for vari-

ations on `ok()` (and all of them end up calling `ok()` anyway), any improvement in readability justifies a new function so that testing can be made as painless as possible. (There's even an `isnt()` function to avoid having to put a negation operator in an `is()` call, and, in true Perl tradition, there's an `isn't()` alias.)

`Test::More` lets you designate blocks of tests as conditional in two different ways. By naming a block `SKIP`, you can then call the `skip()` function inside the block if you want to skip the tests in it:

```
SKIP: {
  eval { require Acme::Garden;
         Acme::Garden->export;
       };
  $@ and skip "Don't have Acme::Garden installed", 2;
  is(pull_weed(), "kudzu", "It's taking over!");
  is(seed_depth("gladiolus"), 7, "Flower module ok");
}
```

The other way of marking tests as conditional is to put them in a block named "`TODO`." Failing tests in such a block will be specially annotated so as not to perturb aggregate statistics reported by `Test::Harness`.

The Perl test mechanism is not coupled to source code. (Perl 5 does not lend itself easily to reflective tools; this will be remedied in Perl 6.) On the other hand, its simplicity allows it to be used to write unit tests, integration tests, regression tests, or acceptance tests.

`Test::Simple` is, itself, built on a module, `Test::Builder`, which is designed for reuse by modules providing testing functionality. The introduction of `Test::Builder` spurred an explosion in the creation of testing modules; there are currently over 250 such modules on CPAN in nearly 100 distributions. One of the more useful ones is `Test::Exception`, which verifies that code dies or fails to die when it should. You can also find `Test::Pod`, which vets the syntax of inline documentation written in POD ("Plain Old Documentation").

Use In Practice

Many tools have been created and integrated with the Perl core to make testing easy. A basic test skeleton can be created with the all-purpose h2xs program that has come with all versions of Perl 5. (If you want to be on the leading edge, the CPAN module `ExtUtils::ModuleMaker` is better.) h2xs is best used when you're developing a module, because it creates the skeleton for that at the same time. The command:

```
h2xs -Xan Store::Inventory
```

creates a tree under a subdirectory `Store-Inventory` (or `Store/Inventory` prior to Perl 5.8.1) with a stub for the `Store::Inventory` module, and a subdirectory `t` with an initial test file. (Old versions of Perl created a `test.pl` test file instead.) In fact, as soon as you create a makefile from the skeleton instructions that h2xs made, via:

```
perl Makefile.PL
```

you can immediately type *make test* and see that first test run. You now have a test that passes (it verifies that the module skeleton loads) and you haven't even written any code yet.

The output of `make test` looks different from that of a standalone test file, however. That's because `make test` aggregates test files using another key core module, `Test::Harness`. Reading the output of all the `ok()` calls to tabulate their successes would be tedious. `Test::Harness` exports a function, `runtests()`, used by `make test`

to run all the test files in the `t` directory. `runtests()` intercepts their output and tallies the `ok/not ok` results, presenting a running count and a statistical summary. The output of `runtests()` looks familiar to

"With a little work, the end-to-end tests can be the actual requirements document for the application"

anyone who has built Perl from source, because the `make test` stage of building Perl uses the same testing mechanism. This is where the first line of output from an individual test (`"1..N"`) comes in. This tells `runtests()` how many tests it should expect to see output for.

You can now develop your module following the Test First principle. First, you write a unit test, then you write a method to satisfy the requirement of the test. With a little work, the end-to-end tests can be the actual requirements document for the application. Listings Two and Three are a sample test file and module file, respectively.

Listing Two

```
use Test::More tests => 11;
use Test::Exception;

BEGIN { use_ok('Store::Inventory') };

my $stock = Store::Inventory->new;

isa_ok($stock, 'Store::Inventory');

my $total;
while (<DATA>)
{
  my ($name, $price, $count) = split;
   my $item = Store::Item->new(name => $name, price =>
                               $price, count => $count);
  isa_ok($item, 'Store::Item');
  lives_ok { $stock->add_item($item) } 'Added item';
  is($stock->total_assets, $total += $count * $price);
}
lives_ok { $stock->sell('cornflakes') };
is($stock->total_assets, $total -= 2.99);
throws_ok { $stock->sell('hedgehogs') }
          qr/No hedgehogs in stock/,
          "Can't sell what we don't have";
__END__
cornflakes 2.99 250
bananas    0.39 400
```

Listing Three

```
package Store::Inventory;
use strict;
use warnings;

use Class::Struct;
use List::Util 'sum';

struct(items => '%');

sub add_item {
  my ($self, $item) = @_;
  $self->items->{$item->name} = $item;
}
sub total_assets {
  my $self = shift;
  return sum(map $_->count * $_->price => values %{
                                $self->items });
}
sub sell {
  my ($self, $name) = @_;
  my $item = $self->items->{$name}
    or die "No $name in stock\n";
  $item->sell;
}
package Store::Item;
use Class::Struct;
```

```
struct(name => '$', count => '$', price => '$');
sub sell {
  my $self = shift;
  $self->count($self->count - 1);
}
1;
```

Segregating tests into multiple files within the `t` directory can be used either to make the sizes of each file more manageable, or as a guide to partitioning the tests. You could define setup and teardown procedures in each file if you wanted to make each one effectively a single independent unit test; it's up to you. In true Perl fashion, you have enough rope either to hang yourself or to make a lasso.

For Further Exploration

Once you've mastered the basics of Perl testing, you may be interested in more advanced capabilities. For rigorous unit testing of legacy object-oriented applications, `Test::MockObject` provides a complete, if lengthy, solution. `Test::MockObject` lets you create objects that mock the behavior of the ones actually used in your application so they present apparently the same interface to the calling tests but under your complete control and without the side effects. (Testing a database is such a common need for mocking that the module `DBD::Mock` was created to help with it.) This is so-called "white box" testing: Instead of looking only at the interfaces a method presents to the outside world, you craft unit tests for it by inspecting its code to see where it calls functions or methods that may alter state. Then you mock those functions to track whether they are being called the way you expect at that point.

If you like to integrate code and its documentation, you might like to do the same with its tests, in which case the CPAN module `Test::Inline` lets you embed tests within code in an extension of the POD language. Just encapsulate them in POD-like blocks marked `"=begin testing" ... "=end testing"` and then extract the tests into a test file with the program pod2test that comes with `Test::Inline`. (This step can be automated in your module's `Makefile.PL` during the `make test` phase.)

"The value of tests is proportional to the proportion of possible equivalence classes that are exercised"

Test::Unit

If you're coming from a Java/JUnit background and want a familiar testing environment in Perl, the CPAN module `Test::Unit` is for you. You define `set_up()` and `tear_down()` procedures to prepare for and clean up after each test case so there can be no side effects between tests. `Test::Unit` even provides the green/red bar, courtesy of PerlTk.

Coverage Testing

The value of tests is proportional to the proportion of possible equivalence classes that are exercised. Generating the valid equivalence classes is still a manual task, but you can at least verify that all the statements in your code have been exercised during testing. Once again, this has been automated. Modules for measuring code coverage have existed for some time: `Devel::DProf` and `Devel::SmallProf` profile code and the parts that haven't been called can be inferred. `Devel::Coverage` shows them explicitly. But that's still too much work for the lazy. Therefore the modern module `Devel::Cover` has been extended to include coverage for tests. Now, simply by defining an environment variable, you can trigger analysis during `make test`

and generate a coverage database that can be analyzed with the cover program that comes with `Devel::Cover`. That program outputs HTML that provides a tabular representation of the lines of code that did not get exercised. There's even another module, `Test::Pod::Coverage`, that verifies that all the files in your distribution provide documentation.

Legacy Application Testing

Perl programmers are often faced with having to deal with legacy applications that lack unit tests or, for that matter, coherent units. Perl's testing framework at this point becomes useful for creating end-to-end regression tests to provide a safety net while the application is modified. This can even be extended to testing Web interfaces via the CPAN module `Test::WWW::Mechanize`. If you find the work of programming the necessary inputs to navigate a Web interface too tedious, the module `HTTP::Recorder` will do it for you.

Automated tests can be useful for other purposes than just application testing. With the Web-based testing tools just mentioned, you could write something like Listing Four to verify that your bank account is doing alright. (If you want to put it in a cron job, knowing that the exit code of a test run is the number of tests that failed may be useful.)

Listing Four

```
use strict;
use warnings;

use Test::WWW::Mechanize tests => 7;

my $mech = Test::WWW::Mechanize->new;

$mech->get_ok("http://www.mybank.example.com/",
    "My bank is still there");
$mech->content_contains("Login",
    "and it still has a login page");

my ($username, $password) = @ARGV;
$mech->set_visible($username, $password);

ok($mech->submit->is_success, "Login form submitted okay");
$mech->content_like(qr/Account data/, "Logged in okay");

my ($balance) = $mech->content =~ /Checking: \$([\d.]+)/;
ok(defined($balance), "Found checking account balance");

ok($balance < 1000, "Got enough for now");
```

Conclusion

Perl is a mature language with mature facilities for all flavors of testing. The beginner-friendly nature of Perl and the common word-of-mouth style of learning it often result in poorly structured development, but that's no excuse for mischaracterizing it as light on testing capabilities. In fact, Perl supports enough testing mechanisms to facilitate very large (tens or hundreds of thousands of lines of code, multiple developer, enterprise level mission critical) applications. In typical Perl community style, there are so many testing modules available that you can find something to support just about any preference, no matter how idiosyncratic.

— Peter Scott is an enterprise system engineering consultant and Perl author and trainer. He can be contacted at http://www.psdt.com/.

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

MEASURING COMPLEXITY CORRECTLY

Code complexity is an excellent predictor of defect probability. However, because the measure is poorly understood and badly applied, it does not deliver its most significant benefits.

http://www.drdobbs.com/240007928

WHAT'S NEW IN .NET FRAMEWORK 4.5

From arrays that can now exceed 2 GB to enhanced background garbage collection, changes in this release of .NET provide immediately useful capabilities.

http://www.drdobbs.com/windows/240008327

PARALLEL SHELLSORT

The strength of the Shellsort algorithm for fewer exchanges done over long distances to quickly place items closer to their final position is the concurrent algorithm's weakness.

http://www.drdobbs.com/parallel/240008640

STATE MACHINE COMPILATION

A lot of embedded systems really lend themselves to state machine descriptions.

http://www.drdobbs.com/embedded-systems/240008792

USING ASYNCHRONOUS METHODS IN ASP.NET 4.5 AND IN MVC 4

Asynchronous methods are an essential part of the new Windows 8 apps user experience. They also provide an easier way to manage operations in existing applications in WPF and ASP.NET MVC.

http://www.drdobbs.com/windows/240008768

HANDLING ERRORS IN IOS AND OS X WITH COCOA

Cocoa provides two classes for smooth handling of errors: `NSError` and `NSResponder`, which allow programs to dispatch an error object and respond to it from key points in the application. Only by knowing the underpinnings of each class is it possible to write truly user-friendly apps.

http://www.drdobbs.com/architecture-and-design/240007940

MICROSOFT VISUAL STUDIO 2012 REVIEWED

The new IDE from Microsoft is a must-have tool for Windows 8 development. For developers working on current platforms, it has many attractive new features — but you'll have to abide its peculiar user interface.

http://www.drdobbs.com/tools/240007128

IN THIS ISSUE

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

CONTRIBUTING EDITORS

Scott Ambler
Mike Riley
Herb Sutter

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

Dr.Dobb's Business Contacts

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

EVP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Salvatore Silletti
(212) 600-3327, ssilletti@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, aleemoll@techweb.com

Marketing Manager, Monique Kakegawa
(949) 223-3609, mluttrell@techweb.com

Program Manager, Diane Scala
516-562-5476, dscala@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, ssorhaindo@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Account Manager, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECH

Paul Miller CEO

John Dennehy CFO

David Michael CIO

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz EVP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, UBM Tech Media

Fritz Nelson VP, Guest Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

Eric Lundquist VP and Guest Editorial Analyst, InformationWeek Business Technology Network

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing