Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Testing Complex Systems

Good design practices can save your system and your sanity

November 05, 2012
URL:http://www.drdobbs.com/testing/testing-complex-systems/240008913

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

Developing 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.

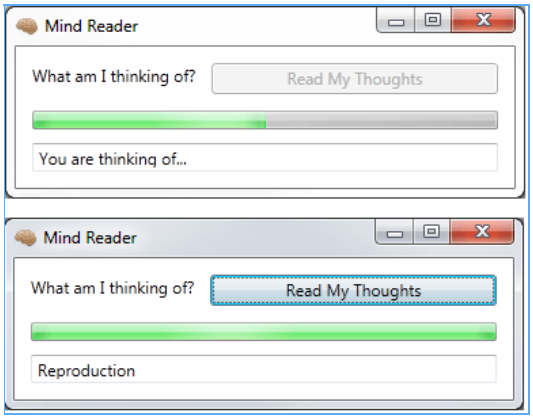

Figure 1.

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 allows 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 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` interfaces 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 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 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.

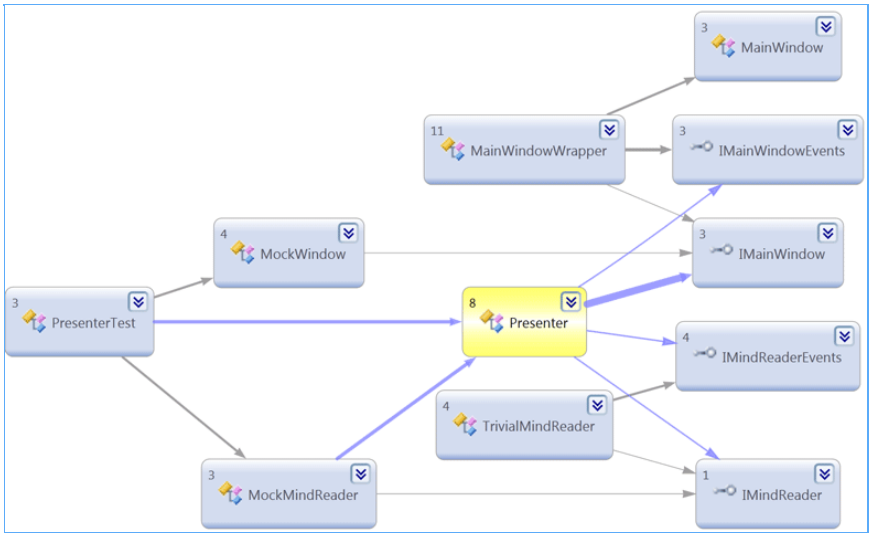

Figure 2.

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:

1. 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.
2. When progress reports arrive via `OnProgress()`, it should update the progress bar.
3. 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() { 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() { "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() { 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:<percent>*" (for example. `"progress:25"`), and finally, it sends the thought itself as *thought: <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. Some times 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:

1. Run your code locally and connect to the test environment. This option is usually good for testing client code.
2. 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:

1. The DB schema or file formats used for persistence
2. 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:

1. Callback-based APIs
2. Future-based APIs
3. 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.

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 fall-back. 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 cohesive 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.

Related Article

Testing Complex C++ Systems

Testing Python and C# Code