Analyzing, Improving, and Testing a Small Program [Draft]

This article analyzes a simple program and discusses how it can be improved and tested. The idea is to walk through several iterations of a program to see how it evolves, which mirrors iterations of refactoring I have performed during my own work.

The program calls a web service to retrieve a list of builds and prints out the highest ranking build, where the ranking of a build is the length of its build number (the higher the better).

Builds are queried by their id using the Visual Studio Team Services (VSTS) REST API through the use of a Nuget package VSORestAPI. The build ids to query are located in a text file called *buildsIds.txt* that exists on the Desktop directory of the host machine. The program parses *buildIds.txt* to retrieve the list of build ids to query, queries the builds, and then returns the highest ranking build.

Initial Version (Commentary)

Iteration 1 (Commentary)

Iteration 2 (Commentary)

Iteration 3 (Commentary)

Initial Version (No Unit Tests)

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using VSORestAPI;

public class Program
{
    public static void Main(string[] args)
    {
        var account = args[0];
        var project = args[1];
        var bestBuild = GetTopRankedBuild(account, project);
        Console.WriteLine($"The top ranked build is {bestBuild.BuildNumber}");
    }

    /// <summary>
    /// Retrieves the top ranked build.
    /// </summary>
    /// <param name="account">the VSTS account the builds were launched on</param>
    /// <param name="project">the VSTS project the builds were launched on</param>
    /// <returns>the top ranked build, or null if there are no builds</returns>
    internal static Build GetTopRankedBuild(string account, string project)
    {
        var windowsDir = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        var buildIds = File.ReadAllLines(Path.Combine(windowsDir, "buildIds.txt"));
```

```
        var builds = new List<Build>();

        foreach (var buildId in buildIds.Select(x => int.Parse(x)))
        {
            var build = BuildAPI.GetBuild(account, project, buildId);
            builds.Add(build);
        }

        return builds.OrderByDescending(x => x.BuildNumber.Length).FirstOrDefault();
    }
}
```

Commentary On Initial Version

Even the smallest of programs can fail in a surprisingly large number of ways. Look through the code and try to identify assumptions that lead to failure.

Points of Failure

The program can fail in many ways:

- If the program is passed less than two arguments, the index into args throws an IndexOutOfBoundsException
- The arguments to GetTopRankedBuild are not validated
- If *buildIds.txt* does not exist, File.ReadAllLines throws a FileNotFoundException
- The content of *buildIds.txt* may not contain the list integers the program is expecting, causing the call to int.Parse to throw a FormatException
- The call to vsts.GetBuild may throw an exception. Without reading the documentation of VSORestAPI, we have no way of knowing which exceptions it may throw
- If the build object returned by vsts.GetBuild is null or its BuildNumber property is null, the call to builds.OrderByDescending throws a NullReferenceException
- The path specified by Environment.SpecialFolder.Desktop may not be reliable. For example, can this location depend on the logged on user?
- If GetTopRankedBuild returns null, Main will crash trying to print the build number of the best build

Dealing with failure scenarios is difficult and frustrating. Unlike the happy path, how to handle a failure scenario is often undefined or ambigious. For example, if *buildIds.txt* gets corrupted, how should the program recover, if at all? For a small program such as this, letting it crash and burn is often okay, but when writing code other programmers will take depend on, it is important to define and handle each failure scenario.

Other common assumptions that lead to failures include:

- *Concurreny assumption*: my code is executed on a single thread

- *Globalization assumption*: my code is executed in the same timezone
- *Environment assumption*: my code is executed on the same hardware and software stack
- *Localization assumption*: my code is executed by English users only

Performance Concerns

There are also serious performance concerns:

- How large is *buildIds.txt*? For each build id, the program makes an HTTP request to retrieve the build
- How long does an HTTP request to VSTS take?
    - What happens if the service is offline or is taking a long time to reply?

Unit Testing Impediments

- Testing the program depends on making an HTTP call (or several) to VSTS, thus any unit test requires an Internet connection and retrieving real, persistent, data from VSTS

Iteration 1

Program (Iteration 1)

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using VSORestAPI;

public interface IVsts
{
    Build GetBuild(string account, string project, int buildId);
}

public class HttpVsts : IVsts
{
    public Build GetBuild(string account, string project, int buildId)
    {
        return BuildAPI.GetBuild(account, project, buildId);
    }
}

public class Program
{
    internal static IVsts vsts = new HttpVsts();

    public static void Main(string[] args)
    {
        var account = args[0];
        var project = args[1];
        var bestBuild = GetTopRankedBuild(account, project);
        if (bestBuild != null)
        {
```

```
            Console.WriteLine($"The top ranked build is {bestBuild.BuildNumber}");
        }
        else
        {
            Console.WriteLine("No builds on this machine");
        }
    }

    /// <summary>
    /// Retrieves the top ranked build.
    /// </summary>
    /// <param name="account">the VSTS account the builds were launched on</param>
    /// <param name="project">the VSTS project the builds were launched on</param>
    /// <returns>the top ranked build, or null if there are no builds</returns>
    internal static Build GetTopRankedBuild(string account, string project)
    {
        if (string.IsNullOrWhiteSpace(account)) throw new ArgumentException(nameof(account));
        if (string.IsNullOrWhiteSpace(project)) throw new ArgumentException(nameof(project));

        var windowsDir = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        var buildIdsFile = Path.Combine(windowsDir, "buildIds.txt");
        var buildIds = Enumerable.Empty<int>();

        if (File.Exists(buildIdsFile))
        {
            var buildIdsContent = File.ReadAllLines(buildIdsFile);
            try
            {
                buildIds = buildIdsContent.Select(x => int.Parse(x));
            }
            catch (FormatException e)
            {
                var content = string.Join(Environment.NewLine, buildIdsContent);
                throw new InvalidOperationException(
                    $"Malformed {buildIdsFile}: {content}", e);
            }
        }

        var builds = new List<Build>();
        foreach (var buildId in buildIds)
        {
            var build = vsts.GetBuild(account, project, buildId);
            builds.Add(build);
        }

        return builds
            .OrderByDescending(x => x?.BuildNumber == null ? 0 : x.BuildNumber.Length)
            .FirstOrDefault();
    }
}
```

Commentary On Iteration 1

The first iteration addresses a several failure scenarios and adds a unit test suite.

Failure Scenario Improvements

First, the GetTopRankedBuild method validates that parameters account and project are non-null and non-empty. Next, the existence of *buildIds.txt* is verified before trying to read it. If a FormatException is thrown while parsing the content of *buildIds.txt*, it is caught, and an InvalidOperation is thrown that includes the path and content of *buildsIds.txt* in the exception message. A possible NullReferenceException is avoiding when ranking builds by assigning a rank of zero to builds that are null or have a null BuildNumber property. Finally, Main includes a conditional statement checking if the object returned by GetTopRankedBuild is null.

Unit Tests

The following changes allow the program to be more easily unit tested:

1. The VSTS service is exposed via the IVsts interface
2. The VSTS service used by program is a static field

During a test run, the TestInitialize function sets the IVsts service to an instance of TestVsts, so the service can be mocked. A tenet of good testing is mocking (read: controlling) all parts of the program, except for the portion (a.k.a. unit) you want to test.

Unit Test Shortcomings

There are still several issues with this iteration of the unit tests:

- The file system is not mocked, so the unit tests must create *buildIds.txt* on the desktop of the test machine. **Any data retrieval outside main memory should be viewed as an external service**
- The program relies on global state, which is a bad programming practice. Each unit test must set global memory, and then possibly reset it upon completion
- A change to the IVsts interface requires a change to the testing implementation (TestVsts), in addition to the actual implementation (HttpVsts), even though an addition to the interface should not affect any tests
- The testing implementation (TestVsts) itself is code to write and maintain, and may can contain bugs
- If there are many interfaces, writing test implementations of all the program's interface is burdensome work
    - Lots of boilerplate code may be required. For example, to test that an interface method is called with a set of parameters, the testing implementation must record each time it is called and its parameter values
- The current unit tests only test the case where the program behaves as expected, not when it fails

The program and its tests could be improved if we:

- Introduce an additional interface that exposes the file system as a service
- Introduce a mocking framework to avoid writing test implementations of service interfaces

- In this example, the Moq framework will be used

Iteration 2

Program (Iteration 2)

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using VSORestAPI;

public interface IVsts
{
    Build GetBuild(string account, string project, int buildId);
}

public class HttpVsts : IVsts
{
    public Build GetBuild(string account, string project, int buildId)
    {
        return BuildAPI.GetBuild(account, project, buildId);
    }
}

public interface IFileSystem
{
    string[] ReadAllLines(string path);
    bool FileExists(string path);
}

public class FileSytem : IFileSystem
{
    public string[] ReadAllLines(string path)
    {
        return File.ReadAllLines(path);
    }

    public bool FileExists(string path)
    {
        return File.Exists(path);
    }
}

public class Program
{
    internal static IVsts vsts;
    internal static IFileSystem fileSystem;

    private static void CreateServices()
    {
        vsts = new HttpVsts();
        fileSystem = new FileSytem();
    }

    public static void Main(string[] args)
```

```
    {
        CreateServices();
        var account = args[0];
        var project = args[1];
        var bestBuild = GetTopRankedBuild(account, project);
        Console.WriteLine($"The top ranked build is {bestBuild.BuildNumber}");
    }

    /// <summary>
    /// Retrieves the top ranked build.
    /// </summary>
    /// <param name="account">the VSTS account the builds were launched on</param>
    /// <param name="project">the VSTS project the builds were launched on</param>
    /// <returns>the top ranked build, or null if there are no builds</returns>
    internal static Build GetTopRankedBuild(string account, string project)
    {
        if (string.IsNullOrWhiteSpace(account)) throw new ArgumentException(nameof(account));
        if (string.IsNullOrWhiteSpace(project)) throw new ArgumentException(nameof(project));

        var windowsDir = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        var buildIdsFile = Path.Combine(windowsDir, "buildIds.txt");
        var buildIds = Enumerable.Empty<int>();

        if (fileSystem.FileExists(buildIdsFile))
        {
            var buildIdsContent = fileSystem.ReadAllLines(buildIdsFile);
            try
            {
                buildIds = buildIdsContent.Select(x => int.Parse(x));
            }
            catch (FormatException e)
            {
                var content = string.Join(Environment.NewLine, buildIdsContent);
                throw new InvalidOperationException(
                    $"Malformed {buildIdsFile}: {content}", e);
            }
        }

        var builds = new List<Build>();
        foreach (var buildId in buildIds)
        {
            var build = vsts.GetBuild(account, project, buildId);
            builds.Add(build);
        }

        return builds
            .OrderByDescending(x => x?.BuildNumber == null ? 0 : x.BuildNumber.Length)
            .FirstOrDefault();
    }
}
```

Commentary On Iteration 2

The file system is now mocked, so the unit tests no longer must create *buildIds.txt* on the Desktop of the test machine. Instead, the Moq framework mocks the IFileSystem interface to fake the existence of *buildIds.txt* and its content. The TestVsts class is removed and is replaced by a Moq implementation.

The following two lines of code is an example of how mock objects are controlled using Moq:

```
vsts = new Mock<IVsts>();
vsts.Setup(x => x.GetBuild(It.IsAny<string>(), It.IsAny<string>(), 1)).Returns(new
Build() { Id = 1, BuildNumber = "firstBuild" });
```

This is telling Moq that any call to IVsts.GetBuild where the build id is 1 (the third parameter) should return a Build object whose id is 1 and BuildNumber is “firstBuild”. The IFileSystem is mocked the same way.

Similiarly, the following line verifies that IVsts.GetBuild was called one time when a build id of 1 was passed to it:

```
vsts.Verify(x => x.GetBuild(It.IsAny<string>(), It.IsAny<string>(), 1), Times.Once);
```

This functionality completely replaces functionality that was implemented in TestVsts. Pretty nice, huh?

An issue with this iteration of the program is the unit tests must still set the global state of the program to run the tests.

Iteration 3

Program (Iteration 3)

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using VSORestAPI;

public interface IVsts
{
    Build GetBuild(string account, string project, int buildId);
}

public class HttpVsts : IVsts
{
    public Build GetBuild(string account, string project, int buildId)
    {
        return BuildAPI.GetBuild(account, project, buildId);
    }
}

public interface IFileSystem
{
    string[] ReadAllLines(string path);
    bool FileExists(string path);
```

```
}

public class FileSytem : IFileSystem
{
    public string[] ReadAllLines(string path)
    {
        return File.ReadAllLines(path);
    }

    public bool FileExists(string path)
    {
        return File.Exists(path);
    }
}

public class ServiceProvider
{
    public IVsts vsts = new HttpVsts();
    public IFileSystem fileSystem = new FileSytem();
}

public class Program
{
    public static void Main(string[] args)
    {
        var account = args[0];
        var project = args[1];
        var bestBuild = GetTopRankedBuild(new ServiceProvider(), account, project);
        Console.WriteLine($"The top ranked build is {bestBuild.BuildNumber}");
    }

    /// <summary>
    /// Retrieves the top ranked build.
    /// </summary>
    /// <param name="sp">the service provider</param>
    /// <param name="account">the VSTS account the builds were launched on</param>
    /// <param name="project">the VSTS project the builds were launched on</param>
    /// <returns>the top ranked build, or null if there are no builds</returns>
    internal static Build GetTopRankedBuild(ServiceProvider sp, string account, string project)
    {
        if (string.IsNullOrWhiteSpace(account)) throw new ArgumentException(nameof(account));
        if (string.IsNullOrWhiteSpace(project)) throw new ArgumentException(nameof(project));

        var windowsDir = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        var buildIdsFile = Path.Combine(windowsDir, "buildIds.txt");
        var buildIds = Enumerable.Empty<int>();

        if (sp.fileSystem.FileExists(buildIdsFile))
        {
            var buildIdsContent = sp.fileSystem.ReadAllLines(buildIdsFile);
            try
            {
                buildIds = buildIdsContent.Select(x => int.Parse(x));
            }
            catch (FormatException e)
            {
                var content = string.Join(Environment.NewLine, buildIdsContent);
                throw new InvalidOperationException(
```

```
                    $"Malformed {buildIdsFile}: {content}", e);
            }
        }

        var builds = new List<Build>();
        foreach (var buildId in buildIds)
        {
            var build = sp.vsts.GetBuild(account, project, buildId);
            builds.Add(build);
        }

        return builds
            .OrderByDescending(x => x?.BuildNumber == null ? 0 : x.BuildNumber.Length)
            .FirstOrDefault();
    }
}
```

Commentary On Iteration 3

The ServiceProvider class is introduced as a container of services; it encapsulates the IVsts and IFileSystem services. The GetTopRankedBuild is modified to accept an instance of ServiceProvider as its first parameter. The unit tests were modified to populate a ServiceProvider with mock IVsts and IFileSystem services. This removes the use of static fields entirely, and creates a scalable pattern for adding more services and passing services around the program as it grows.

Future Improvments

- Add unit tests for failure scenarios
- Use a generic service provider
    - For example, var fileSystem = serviceProvider.GetService<IFileSystem>()

Additional Reading

Programmers like to argue over terminology when it comes to unit testing. I use the verb “mock” to mean

1. Controlling the return values of services
2. Verifying services are called with the correct parameters

For more reading, see Mocks Aren’t Stubs, and a response to it, Mocks, Stubs, and Fakes: it’s a continuum