Writing a Unit Test

To illustrate some unit testing techniques for an object-oriented language, let's start with an example of some code we wish to add unit tests for. In this example, we have a configuration class that contains all the startup options for an app we are writing. Normally it reads from a `.config` file, but we are having three problems with the current implementation:

1. There is a bug in the Configuration class, and we have no unit tests since it relies on reading a config file
2. We can't unit test any of the code that relies on the Configuration class reading a config file
3. In the future, we want to allow for configuration to be saved in the cloud and accessed via REST api.

The bug we are trying to fix is that if there are multiple empty lines in the configuration file, an IndexOutOfRangeException is being thrown. Our class currently looks like this:

```
using System.IO;
using System.Linq;

public class Configuration
{
    // Public getter properties from configuration object
    public string MyProperty { get; private set; }

    public void Initialize()
    {
        var configContents = File.ReadAllLines(".config");

        // Config is in the format: key=value
        var config = configContents.Select(l => l.Split('='))
                                   .ToDictionary(kv => kv[0], kv => kv[1]);

        // Assign all properties here
        this.MyProperty = config["myproperty"];
    }
}
```

Abstraction

In our example, we have a single dependency: the file system. Rather than just abstracting the file system entirely, let us think about why we need the file system and abstract the *concept* rather than the implementation. In this case, we are using the `File` class to read from the config file, and the config contents. The abstraction concept here is some form or configuration reader that returns each line of the configuration in a string array. We could call it `ConfigurationReader`, and it has a single method, `Read`, which returns the contents.

When creating abstractions, it can be good practice creating an interface for that abstraction, in languages that support it. In the example with C#, we can create an `IConfigurationReader` interface, and instead of just having a `ConfigurationReader` class we can be more specific and name if `FileConfigurationReader` to indicate that it reads from the file system:

```
// IConfigurationReader.cs
public interface IConfigurationReader
{
    string[] Read();
}

// FileConfigurationReader.cs
public class FileConfigurationReader : IConfigurationReader
{
    public string[] Read()
    {
        return File.ReadAllLines(".config");
    }
}
```

Now that the file dependency has been abstracted away, we need to update our Configuration class's Initialize method to use the new abstraction instead of calling `File.ReadAllLines` directly:

```
public void Initialize()
{
    var configContents = new FileConfigurationReader().Read();

    // Config is in the format: key=value
    var config = configContents.Select(l => l.Split('='))
                               .ToDictionary(kv => kv[0], kv => kv[1]);

    // Assign all properties here
    this.MyProperty = config["myproperty"];
}
```

As you can see, we still have a dependency on the file system, but that dependency has been abstracted out. We will need to use other techniques to break the dependency completely.

Dependency Injection

In the previous section, we abstracted the file access into a `FileConfigurationReader` but we still had a dependency on the file system in our function. We can use dependency injection to inject the right reader into our `Configuration` class:

```
using System.IO;
using System.Linq;

public class Configuration
{
    private readonly IConfigurationReader configReader;

    // Public getter properties from configuration object
    public string MyProperty { get; private set; }

    public Configuration(IConfigurationReader reader)
    {
        this.configReader = reader;
    }

    public void Initialize()
    {
        var configContents = configReader.Read();

        // Config is in the format: key=value
        var config = configContents.Select(l => l.Split('='))
                                   .ToDictionary(kv => kv[0], kv => kv[1]);

        // Assign all properties here
        this.MyProperty = config["myproperty"];
    }
}
```

Above, a technique was used called Constructor Injection. This uses the object's constructor to set what our dependencies will be, which means whichever object creates the `Configuration` object will control which reader needs to get passed in. This is an example of "inversion of control", previously the `Configuration` object controlled the dependency, but instead we pushed up the control to whatever component creates this object.

Note that we injected the interface `IConfigurationReader` and not the concrete class. This is what allows us to break the dependency; whereas originally we had a hard-coded dependency on the `File` class, now we only depend on an object that implements `IConfigurationReader`.

Writing our first unit tests

We started down this venture because we have a bug in the `Configuration` class that was not caught because we do not have unit tests. Let us write some unit tests that gives us full coverage of the `Configuration` class, including a test that tests the scenario described by the bug (if there are multiple empty lines in the configuration file, an IndexOutOfRangeException is being thrown).

However, we still have one problem, we only have a single implementation of `IConfigurationReader`, and it uses the file system, meaning any unit tests we write will still have a dependency on the file system! Luckily since we used dependency injection, all we need to do is create an implementation of `IConfigurationReader` that does not depend on the file system. We could create a mock here, but instead let's create a concrete implementation of the interface which simply returns the passed in string[] - we can call it `PassThroughConfigurationReader` (for more details on why this approach may be better than mocking, see the page on mocking)

```
public class PassThroughConfigurationReader : IConfigurationReader
{
    private readonly string[] contents;

    public PassThroughConfigurationReader(string[] contents)
    {
        this.contents = contents;
    }

    public string[] Read()
    {
        return this.contents;
    }
}
```

This simple class will be used in our unit tests, so we can create different states without requiring lots of file access. Now that we have this in place, we can go ahead and write our unit tests, starting with the tests that describe the current behavior:

```
public class ConfigurationTests
{
    [Fact]
    public void Initialize_EmptyConfig_Throws()
    {
        var reader = new PassThroughConfigurationReader(Array.Empty<string>());
        var config = new Configuration(reader);

        Assert.Throws<KeyNotFoundException>(() => config.Initialize());
    }

    [Fact]
    public void Initialize_CorrectFormat_SetsProperty()
    {
        var reader = new PassThroughConfigurationReader(new[] {
```

```
            "myproperty=myvalue"
        });
        var config = new Configuration(reader);

        config.Initialize();

        Assert.Equal("myvalue", config.MyProperty);
    }
}
```

Fixing the Bug

All our current tests pass, and give us 100% coverage, however as evidenced by the bug, we must not be covering all possible inputs and outputs. In the case of the bug, multiple empty lines would cause an issue. Additionally, `KeyNotFoundException` is not a very friendly exception and is an implementation detail, not something that makes sense when designing the Configuration API. Let's add some more tests and align the tests with how we think the `Configuration` class should behave:

```
public class ConfigurationTests
{
    [Fact]
    public void Initialize_EmptyConfig_Throws()
    {
        var reader = new PassThroughConfigurationReader(Array.Empty<string>());
        var config = new Configuration(reader);

        Assert.Throws<InvalidOperationException>(() => config.Initialize());
    }

    [Fact]
    public void Initialize_MalformedLine_Throws()
    {
        var reader = new PassThroughConfigurationReader(new[] {
            "myproperty",
        });
        var config = new Configuration(reader);

        Assert.Throws<InvalidOperationException>(() => config.Initialize());
    }

    [Fact]
    public void Initialize_MultipleEqualSigns_PropertyContainsNoEquals()
    {
        var reader = new PassThroughConfigurationReader(new[] {
            "myproperty=myval1=myval2",
        });
        var config = new Configuration(reader);
```

```
        config.Initialize();

        Assert.Equal("myval1=myval2", config.MyProperty);
    }

    [Fact]
    public void Initialize_WithBlankLines_Ignores()
    {
        var reader = new PassThroughConfigurationReader(new[] {
            "myproperty=myvalue",
            string.Empty,
        });
        var config = new Configuration(reader);

        config.Initialize();

        Assert.Equal("myvalue", config.MyProperty);
    }

    [Fact]
    public void Initialize_CorrectFormat_SetsProperty()
    {
        var reader = new PassThroughConfigurationReader(new[] {
            "myproperty=myvalue"
        });
        var config = new Configuration(reader);

        config.Initialize();

        Assert.Equal("myvalue", config.MyProperty);
    }
}
```

Now we have 4 failing tests and 1 passing test, but we have firmly established through the use of these tests how we expect callers to user the Configuration class and what is and isn't allowed as inputs. Now we just need to fix the `Configuration` class so that our tests pass:

```
public void Initialize()
{
    var configContents = configReader.Read();

    if (configContents.Length == 0)
    {
        throw new InvalidOperationException("Empty config");
    }

    // Config is in the format: key=value
    var config = configContents.Where(l => !string.IsNullOrWhiteSpace(l))
                               .Select(l =>
                               {
```

```
                                    var splitLine = l.Split('=', 2);
                                    if (splitLine.Length < 2)
                                    {
                                        throw new
InvalidOperationException("Malformed line");
                                    }
                                    return splitLine;
                                })
                                .ToDictionary(kv => kv[0], kv => kv[1]);

    // Assign all properties here
    this.MyProperty = config["myproperty"];
}
```

Now all our tests pass! We have fixed our bug, added unit tests to the `Configuration` class, and have much higher confidence in future changes.

Untestable Code

As described in the abstraction section, not all code can be properly unit tested. In our case we have a single class that has 0% test coverage: `FileConfigurationReader`. This is expected; in this case we kept `FileConfigurationReader` as light as possible with no additional logic other than calling into the third-party dependency. `FileConfigurationReader` is an example of the facade design pattern.

Testable Design and Future Improvements

One of our original problems described in this example is that in the future we expect to load the configuration from a web API. By doing all the work of abstracting the way we load the configuration text and breaking the dependency on the file system, we have already done all the hard work to enable this future scenario! All that needs to be done next is to create a `WebApiConfigurationReader` implementation and use that the construct the `Configuration` object, and it should just work.

That is one of the benefits of testable design, in the process of writing our tests in a safe way, a side effect of that is that we already have our dependencies that might change abstracted, and will require minimal changes to implement.

Another added benefit is we have multiple possibilities opened by this testable design. For example, we can have a cascading configuration set up now using all 3 `IConfigurationReader` implementations, including the one we wrote only for our tests! We can first check if internet access is available and if so use `WebApiConfigurationReader`. If no internet is available, we can

fall back to the local config file on the current system using `FileConfigurationReader`. If for some reason the config file does not exist, we can use the `PassThroughConfigurationReader` as a hard-coded default configuration somewhere in the code. We have full flexibility to do whatever we may need to do in the future!

Last update: August 26, 2024