Using Stub to break dependency in xUnit testing

External Dependency in a CUT

- **DEFINITION** An external dependency is an object in your system that your code under test interacts with, and over which you have no control. (Common examples are filesystems, threads, memory, time, and so on.)

A STUB

- **DEFINITION** A stub is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without dealing with the dependency directly.

An example CUT

```
public class LogAnalyzer
  {
     public bool IsValidLogFileName(string fileName)
     {
       // read through the configuration file
       read("url:validfileextension.config"); // this file will be uploaded
                                              // by some components
                                              // dynamically
       parsing the entries in the configuration file
       .....
        // return true if cor                 extensi
     }
  }
```

An external dependency to external resource

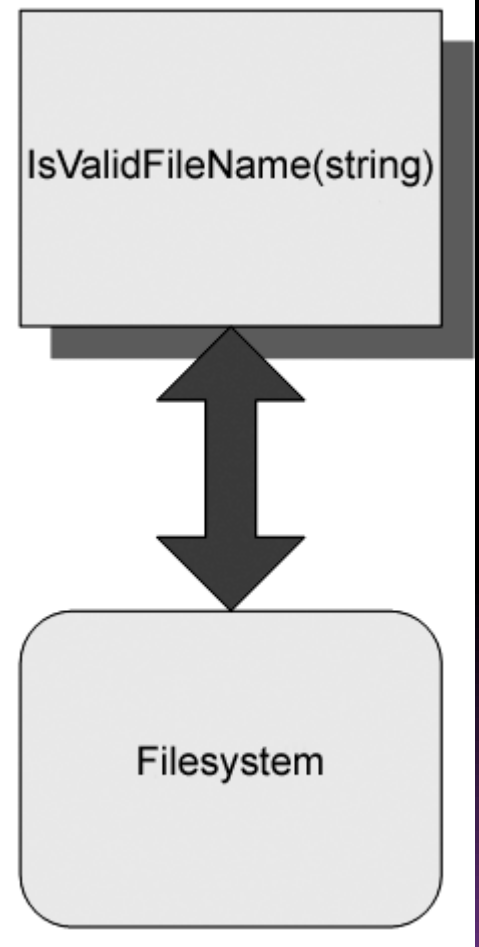
The sample Test code

```
[TestFixture]
public class LogAnalyzerTests
{
   private LogAnalyzer m_analyzer=null;

   [SetUp]
   public void Setup()
   {
      m_analyzer = new LogAnalyzer();
   }
   [Test]
   [Ignore("This test is broken")]
   public void IsValidFileName_validFileLowerCased_ReturnsTrue()
   {
      bool result = m_analyzer.IsValidLogFileName("whatever.slf");
      Assert.IsTrue(result, "filename should be valid!");
   }
   [Test]
   public void IsValidFileName_validFileUpperCased_ReturnsTrue()
   {
      bool result = m_analyzer.IsValidLogFileName("whatever.SLF");
      Assert.IsTrue(result, "filename should be valid!");
   }
   [Test]
   [ExpectedException(typeof(Exception),"No log file with that name exists")]
   public void IsValidFileName_nunintvalidFileUpperCased_ReturnsTrue()
   {
      bool result = m_analyzer.IsValidLogFileName("whatever.SLF");

      Assert.IsTrue(result, "filename should be valid!");
   }
```

The direct external dependency



Old tricks!! – Add abstraction

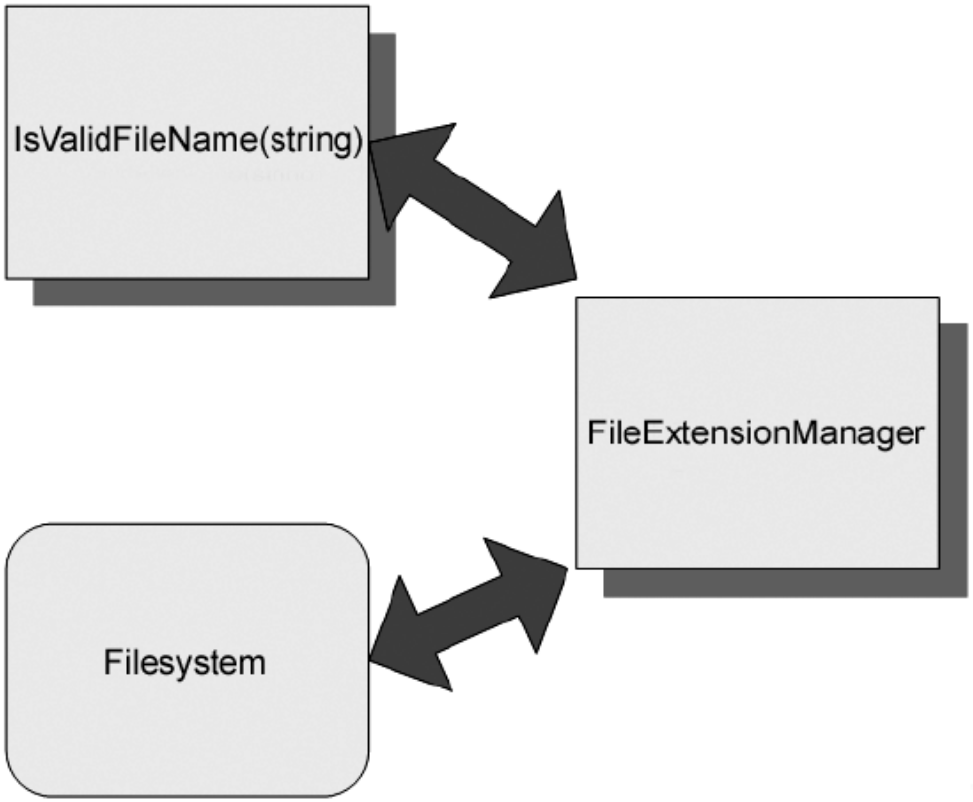
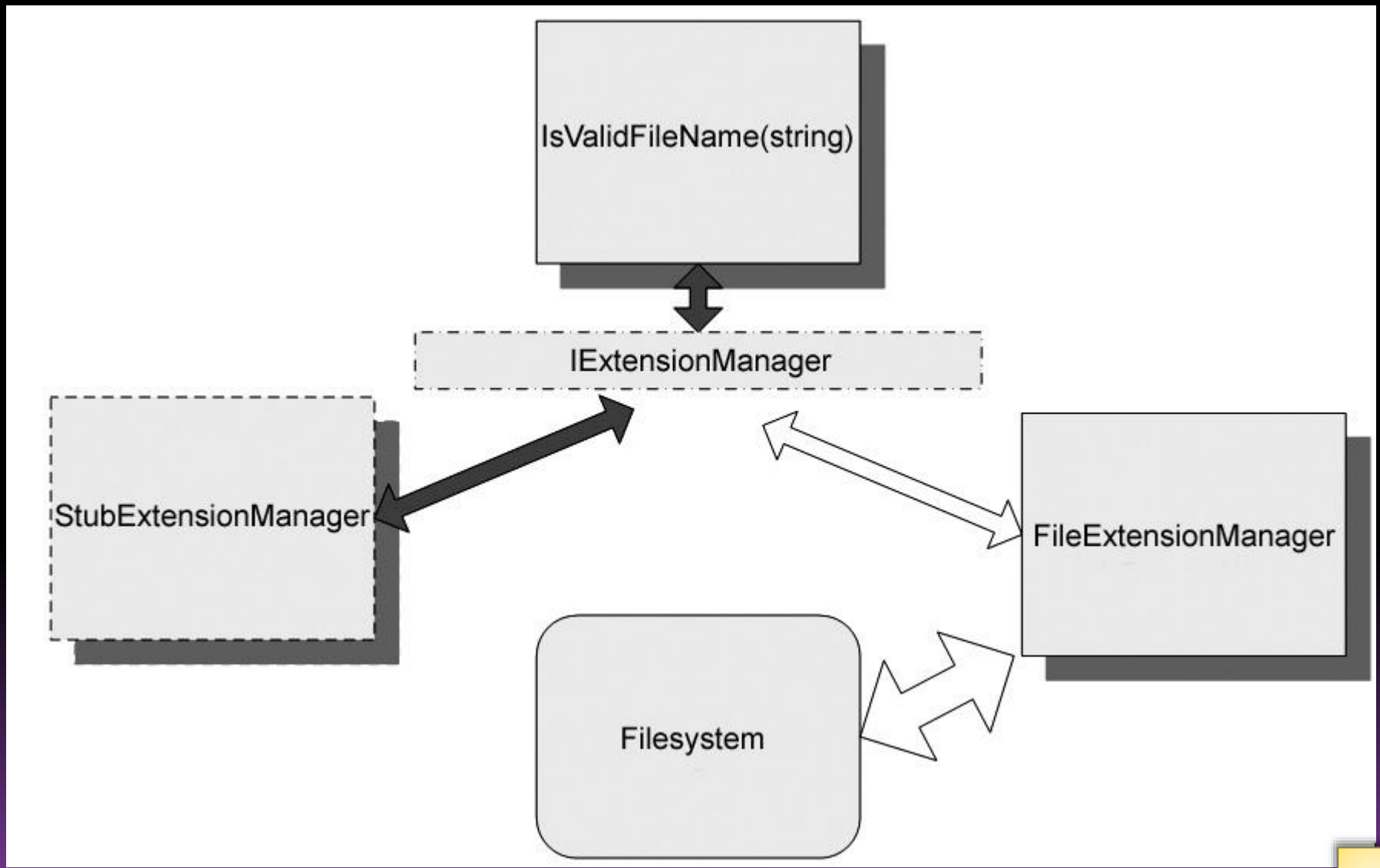


Figure 3.3 Introducing a layer of indirection to avoid a direct dependency on the filesystem. The code that calls the filesystem is separated into a `FileExtensionManager` class, which will later be replaced with a stub in our test.

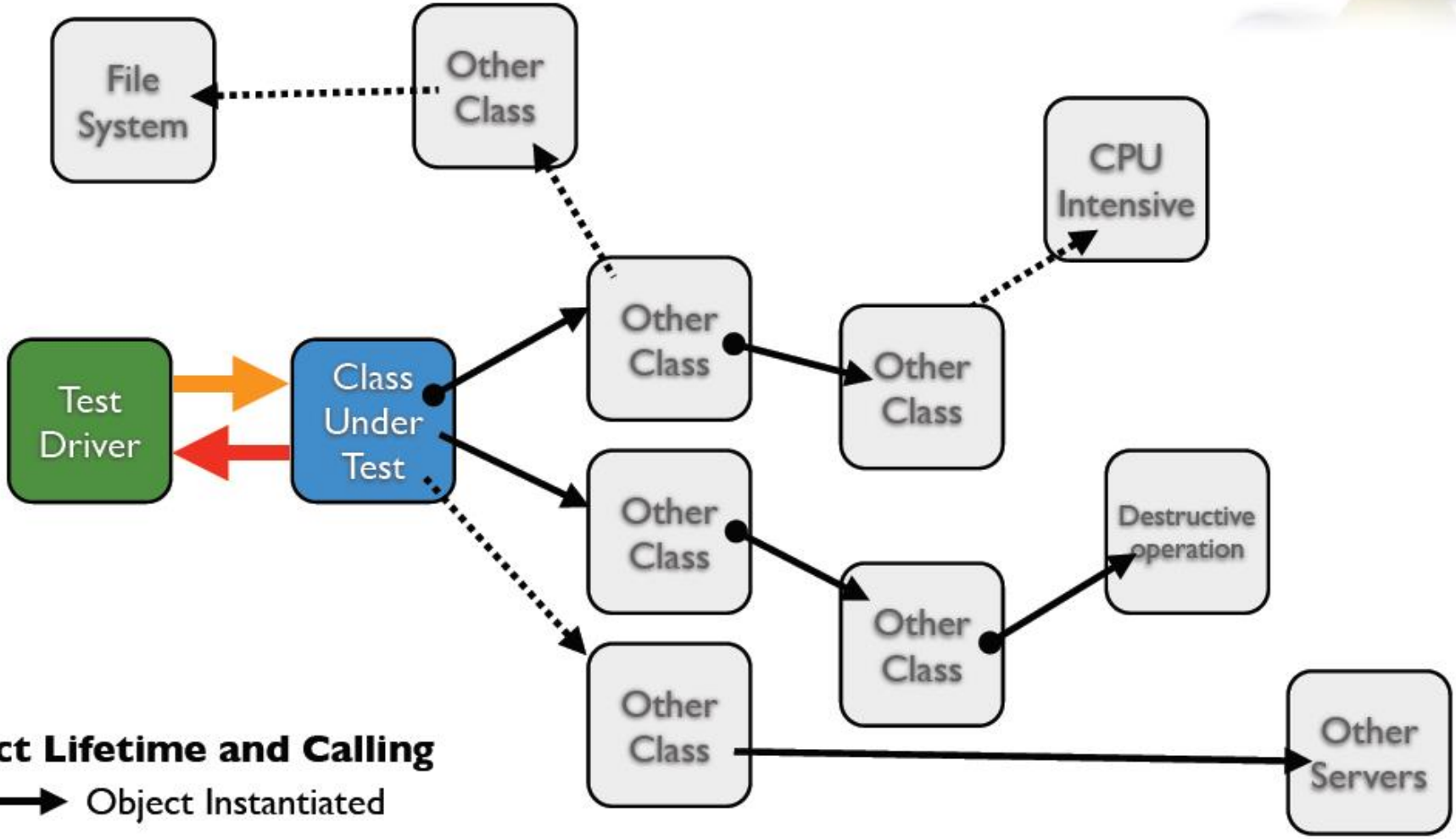
Keep going with the tricks learned in this class



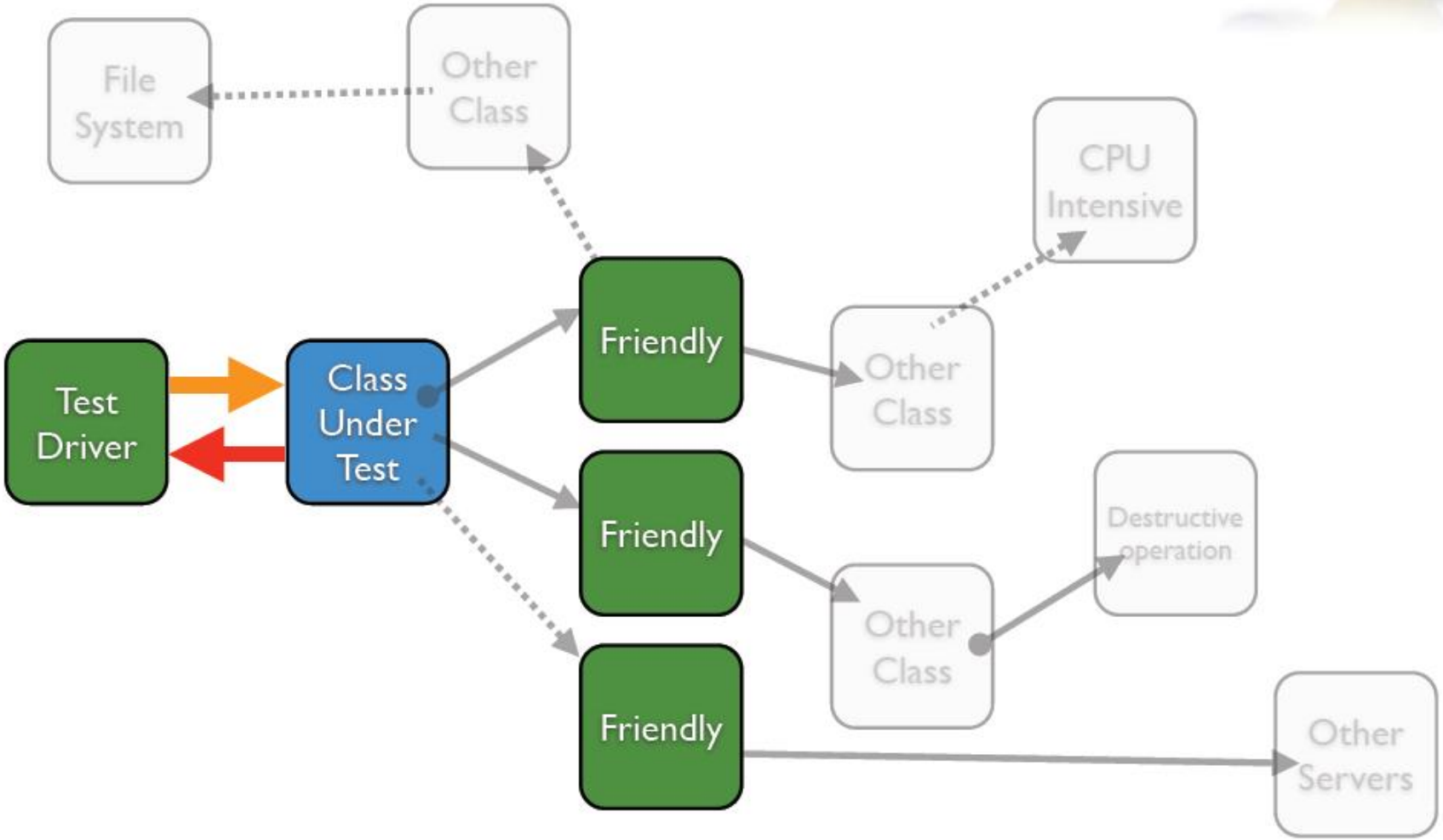
Testability

- In the above example, We've looked at one way of introducing testability into our code base—by creating a new interface.
- DEFINITION: Seams are places in your code where you can plug in different functionality, such as stub classes.

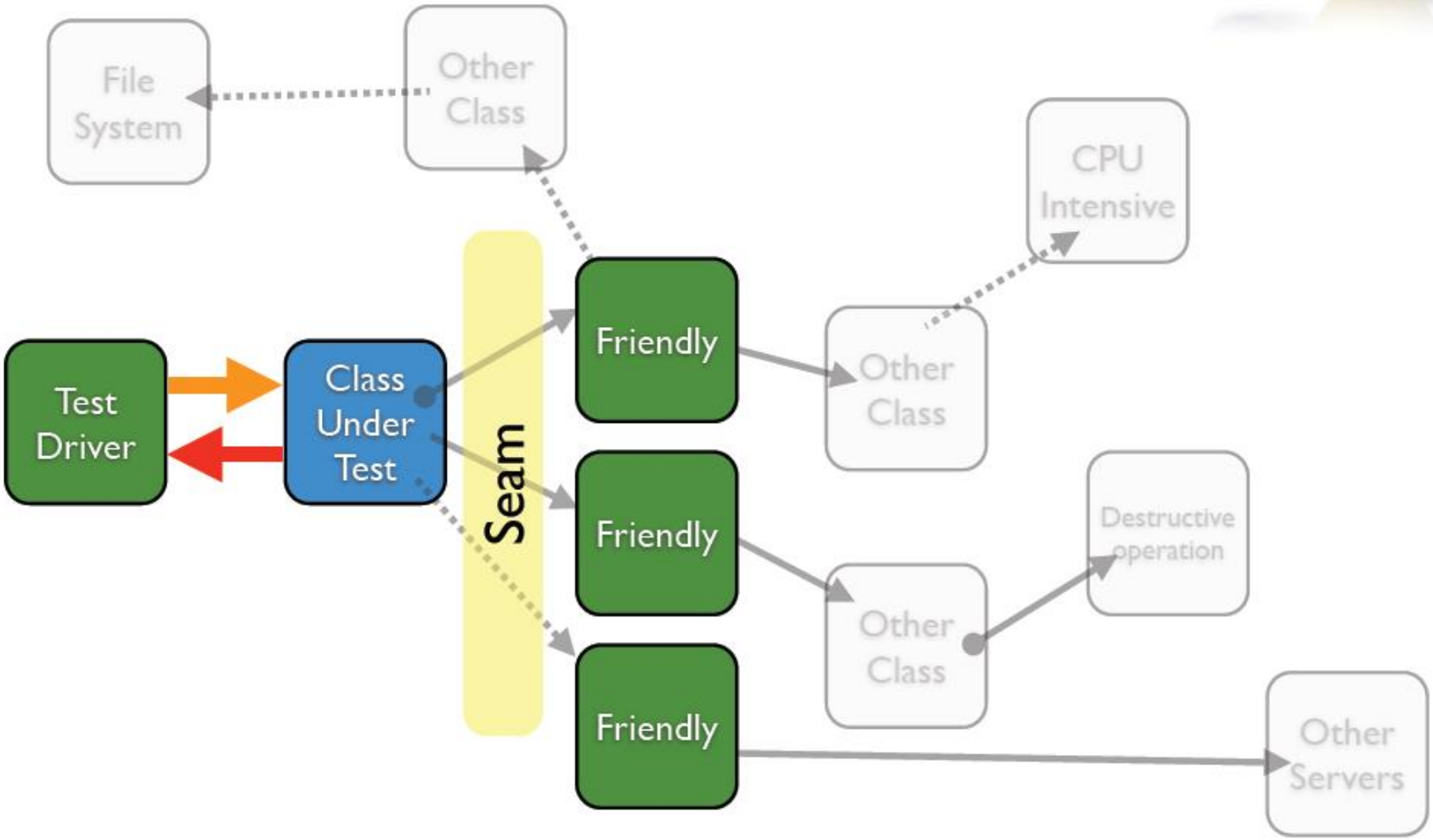
Unit Testing a Class



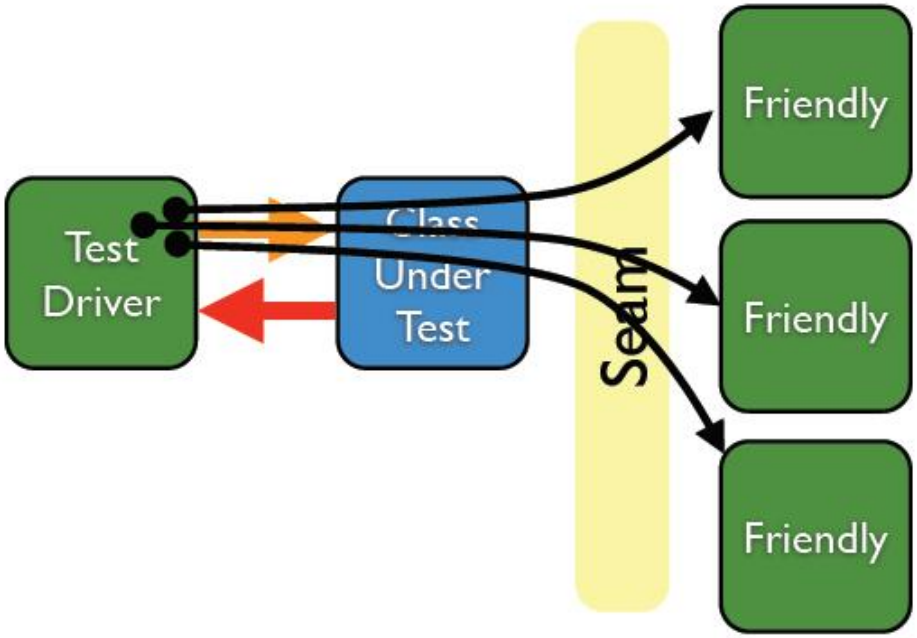
Unit Testing a Class



Unit Testing a Class



Unit Testing a Class



Object Lifetime and Calling

- Object Instantiated
- Object Passed In
- Global Object

- If we want to break the dependency between our code under test and the filesystem, we can use common design patterns, refactorings, and techniques, and introduce one or more seams into the code. We just need to make sure that the resulting code does exactly the same thing.
- Here are some techniques for breaking dependencies:
  - Extract an interface to allow replacing underlying implementation.
  - Inject stub implementation into a class under test.
  - Receive an interface at the constructor level.
  - Receive an interface as a property get or set.
  - Get a stub just before a method call.

Listing 3.2 Extracting an interface from a known class

```
public class FileExtensionManager : IExtensionManager
    {
        public bool IsValid(string fileName)
        {
        ...
        }
    }
public interface IExtensionManager
    {
        bool IsValid (string fileName);
    }

//the method under test:
public bool IsValidLogFileName(string fileName)


   {
       IExtensionManager mgr =
                new FileExtensionManager();
       return mgr.IsValid(fileName);
   }
```

Implements the interface

Defines the new interface

Defines variable as the type of the interface

Listing 3.3 Simple stub code that always returns `true`

```
public class StubExtensionManager:IExtensionManager
    {
        public bool IsValid(string fileName)
        {
            return true;
        }
    }
```

Implements `IExtensionManager`

Receive an interface at the constructor level.

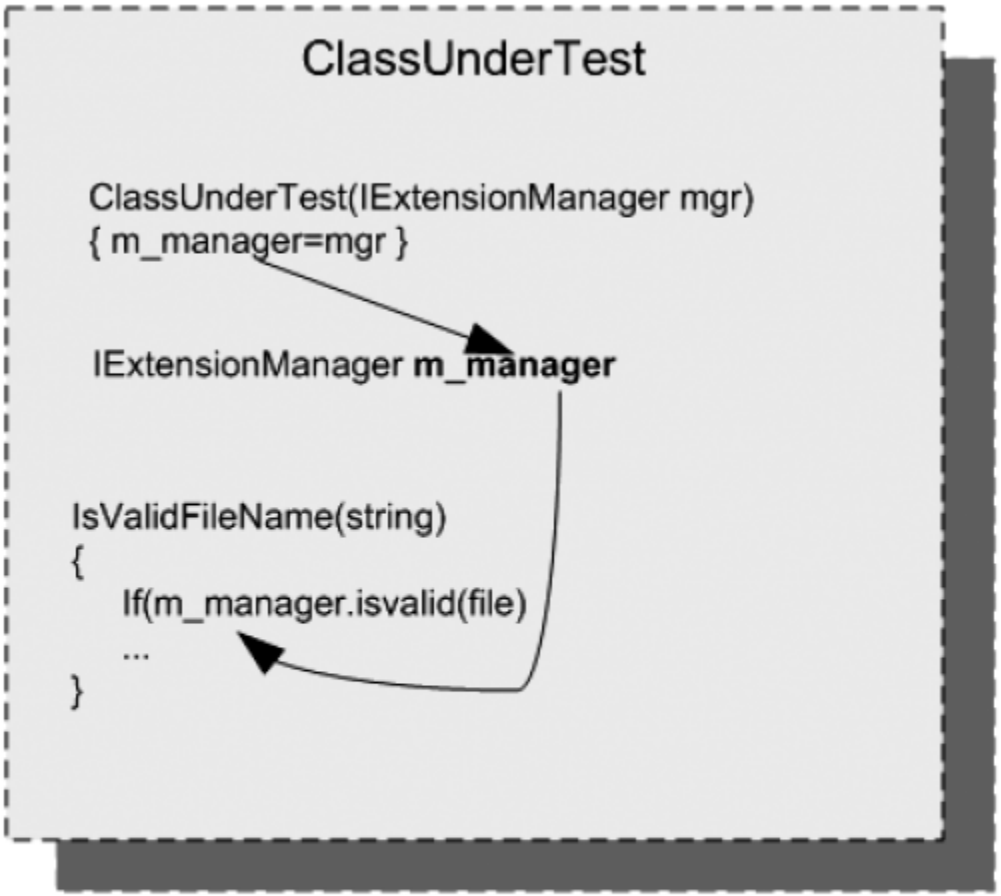


Figure 3.5 Flow of injection via a constructor

Listing 3.4 shows how we could write a test for our `LogAnalyzer` class using a constructor injection technique.

Listing 3.4 Injecting our stub using constructor injection

```
public class LogAnalyzer
    {
        private IExtensionManager manager;

        public LogAnalyzer ()
        {
            manager = new FileExtensionManager();
        }
        public LogAnalyzer(IExtensionManager mgr)
        {
            manager = mgr;
        }

        public bool IsValidLogFileName(string fileName)
        {
            return manager.IsValid(fileName);
        }
    }
```

Defines production code

Creates object in production code

Defines constructor that can be called by tests

```
[TestFixture]                                   <-- Defines test code
public class LogAnalyzerTests
{
    [Test]
    public void
IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
    {
        StubExtensionManager myFakeManager =
                    new StubExtensionManager();
        myFakeManager.ShouldExtensionBeValid
                                        = true;

        //create analyzer and inject stub
        LogAnalyzer log =
            new LogAnalyzer (myFakeManager);    <-- Sends in stub

        //Assert logic assuming extension is supported
        bool result = log.IsValidLogFileName("short.ext");
        Assert.IsFalse(result,
              "File name with less than 5 chars should have failed
               the method, even if the extension is supported");
    }
}

internal class StubExtensionManager : IExtensionManager
{
    public bool ShouldExtensionBeValid;

    public bool IsValid(string fileName)
    {
        return ShouldExtensionBeValid;
    }
}
```

Sets up stub to return true

Defines stub that uses simplest mechanism possible

When you should use constructor injection

- using constructor arguments to initialize objects can make your testing code more cumbersome unless you're using helper frameworks such as IoC containers for object creation. Every time you add another dependency to the class under test, you have to create a new constructor that takes all the other arguments plus a new one, make sure it calls the other constructors correctly, and make sure other users of this class initialize it with the new constructor.
- On the other hand, using parameters in constructors is a great way to signify to the user of your API that these parameters are non-optional. They have to be sent in when creating the object.

Receive an interface as property get or set

3.4.4 Receive an interface as a property get or set

In this scenario, we add a property get and set for each dependency we'd like to inject. We then use this dependency when we need it in our code under test. Figure 3.6 shows the flow of injection with properties.

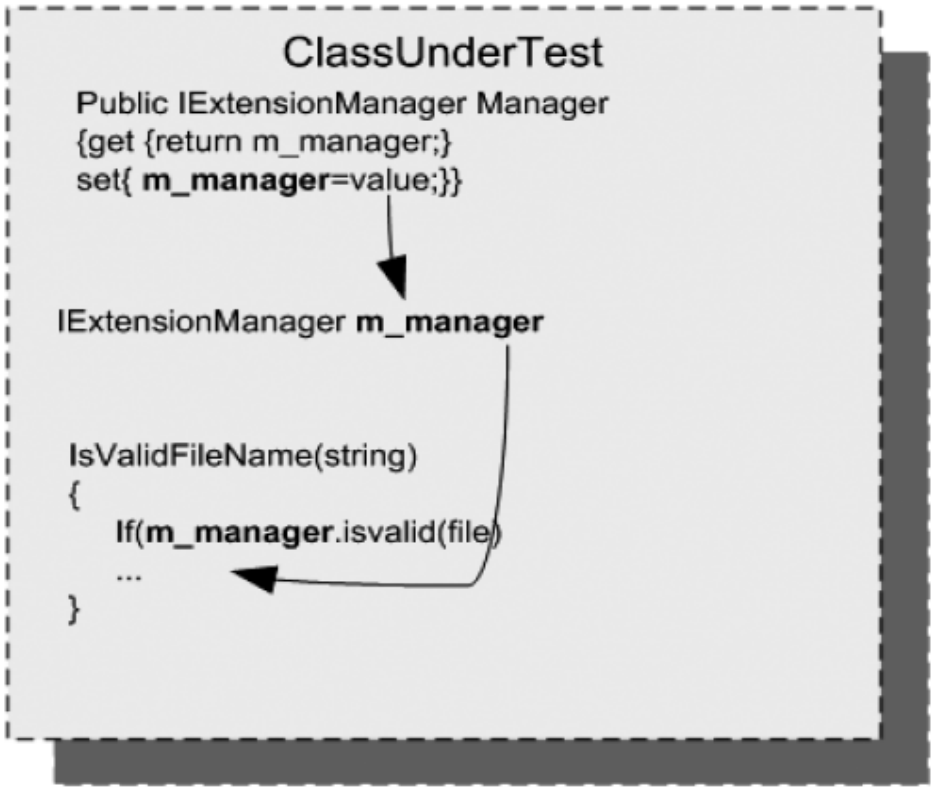


Figure 3.6 Using properties to inject dependencies. This is much simpler than using a constructor because each test can set only the properties that it needs to get the test underway.

Get a stub before a method call using factory pattern

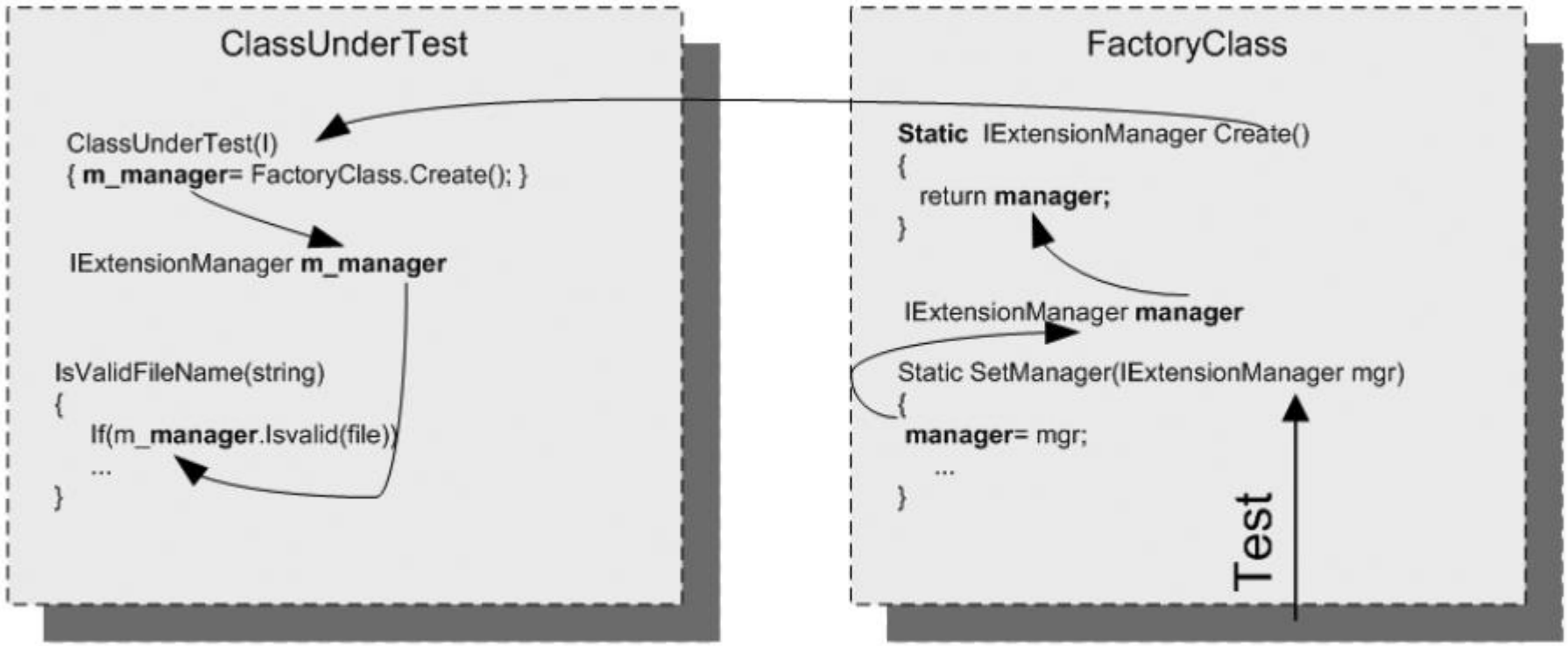


Figure 3.7 A test configures the factory class to return a stub object. The class under test uses the factory class to get that instance, which in production code would return an object that isn't a stub.

Listing 3.6 Setting a factory class to return a stub when the test is running

```
public class LogAnalyzer
    {
        private IExtensionManager manager;

        public LogAnalyzer ()
        {
            manager = ExtensionManagerFactory.Create();
        }

        public bool IsValidLogFileName(string fileName)
        {
            return manager.IsValid(fileName)
            && Path.GetFileNameWithoutExtension(fileName).Length>5;
        }
    }

        [Test]
         Public void
IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
         {
             //set up the stub to use, make sure it returns true
             ...
```

Uses factory in production code

```
            ExtensionManagerFactory
                                .SetManager(myFakeManager);
            //create analyzer and inject stub
            LogAnalyzer log =
                new LogAnalyzer ();

            //Assert logic assuming extension is supported
            ...
        }
    }
```

Sets stub into factory class for this test

```
Class ExtensionManagerFactory
{
    Private IExtensionManager customManager=null;
    Public IExtensionManager Create()
    {
        If(customManager!=null)
return customManager;
        Return new FileExtensionManager();
    }
    Public void SetManager(IExtensionManager mgr)
    {
        customManager = mgr;
    }
```

Defines factory that can use and return custom manager

Hiding seams in release mode

What if you don't want the seams to be visible in release mode? There are several ways to achieve that. In .NET, for example, you can put the seam statements (the added constructor, setter, or factory setter) under a conditional compilation argument, like this:

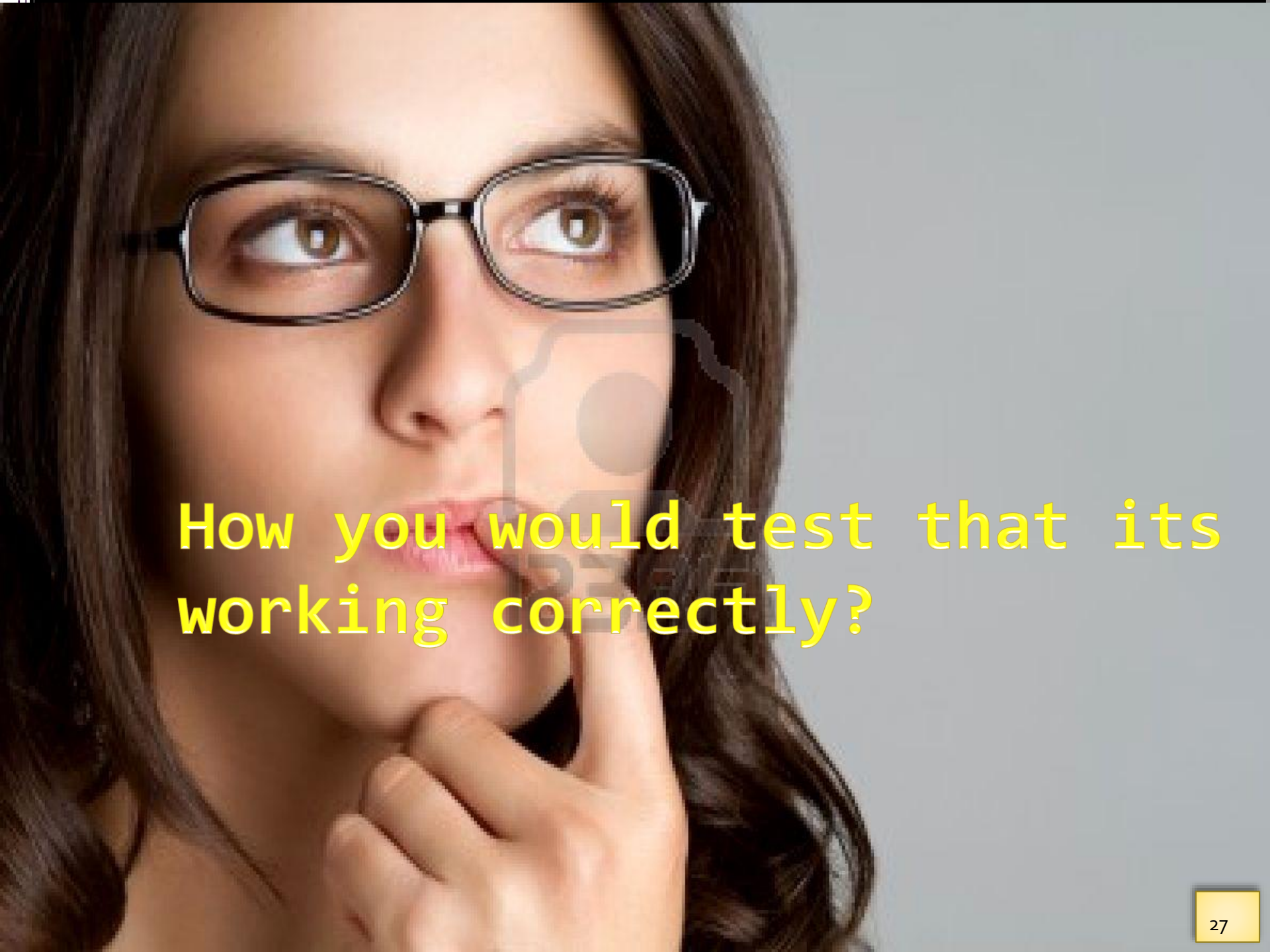
```
#if DEBUG
    MyConstructor(IExtensionManager mgr)
    {...}
#endif
```

There's also a special attribute in .NET that can be used for these purposes:

```
    [Conditional("DEBUG")]
    MyConstructor(IExtensionManager mgr)
    {...}
```

State-based vs Interaction (behavior) testing

- So, you have a watering system and you have given your system specific instructions on when to water the tree in your yard: how many times a day and what quantity of water each time.


How you would test that its working correctly?

State-based testing

- Run the system for 12 hours
- At the end of that time,
  - Check the state of the tree being irrigated
  - Is land moist enough?
  - Is its tree leaves green?
- Is it a good solution? How do you think?

Interaction testing

- At the end of the irrigation hose, set up a device that records when irrigation starts and stops and how much water flows through the device.
- At the end of each day, check that the device has been called with the right number of times, with the correct quantity of water….
- => do not worry about checking the tree

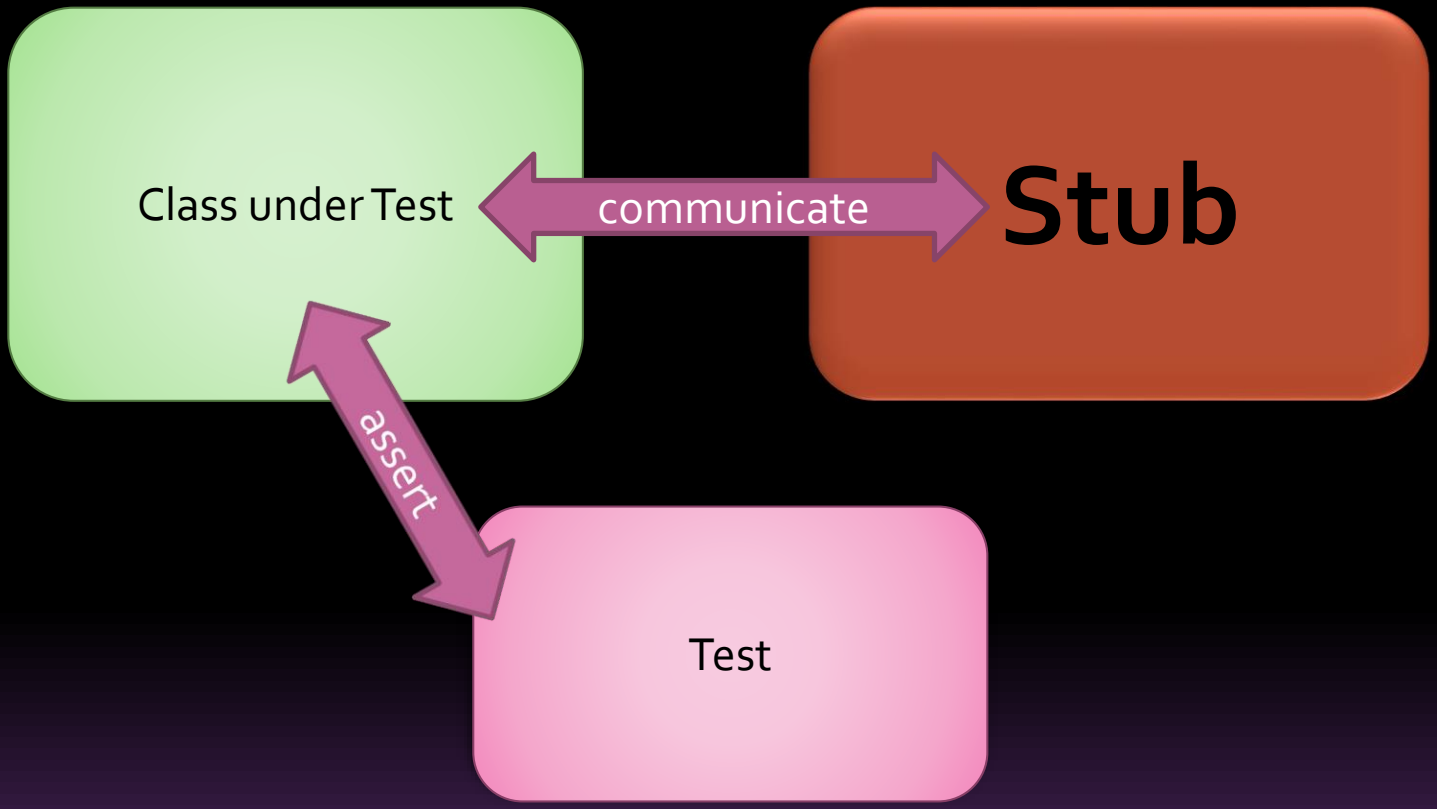
The Role of the device

- So, what is that device that records the irrigation information?
  - It is a fake water hose, or a stub
  - More precisely, a stub that records the calls made to it
- DEFINITION: A **mock** object is a fake object in the system that decides whether the unit test has passed or failed, by verifying whether the OUT (Object Under Test) interacted as expected with the fake object.

The Difference between a stub and a mock object

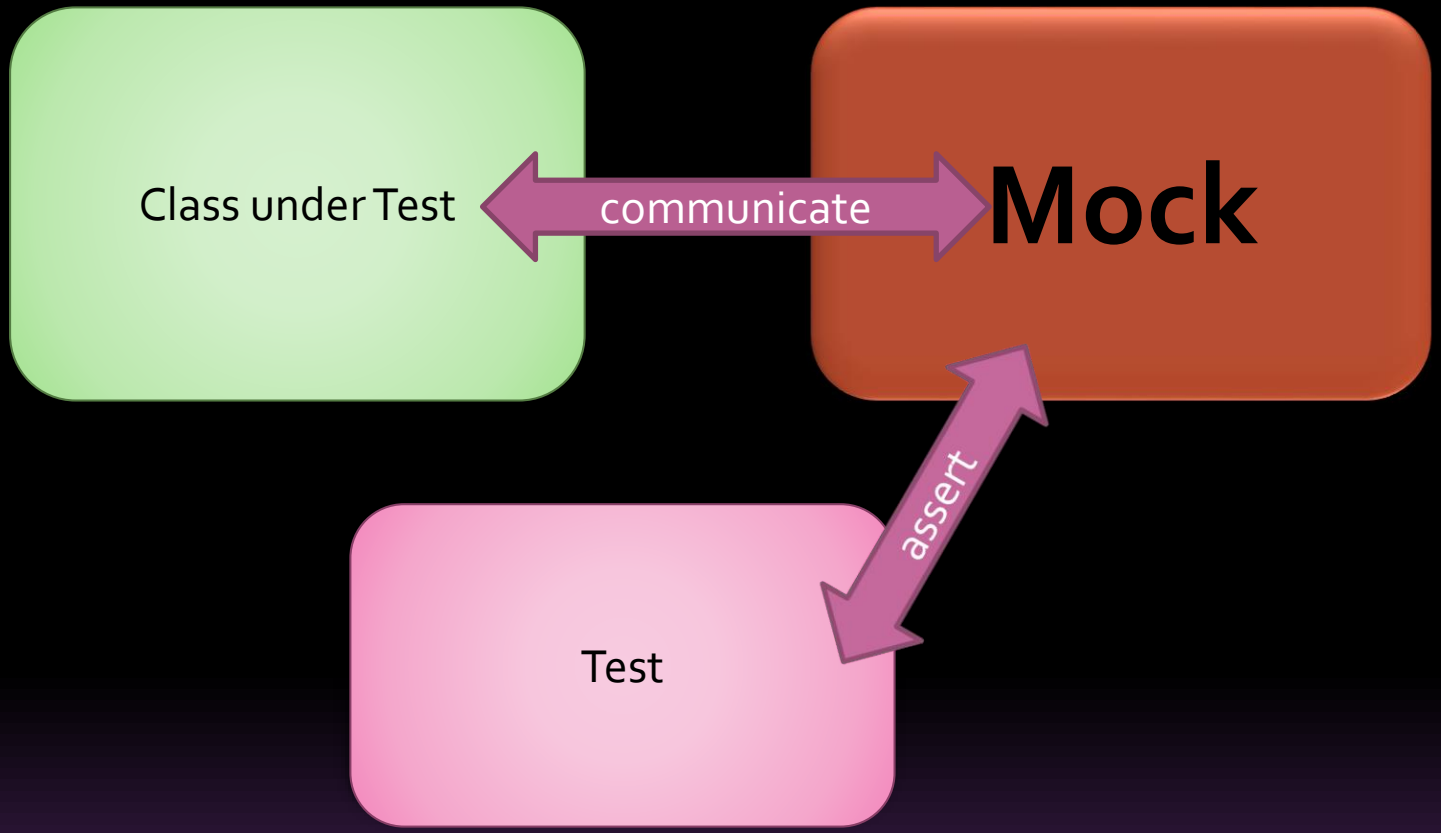
- A stub and a mock can be easily confused.

A stub



A stub can never fail the test

A Mock object



The CUT communicates with the mock object and all communication is recorded In the mock. The test uses the mock object to verify that the test passes

Continue with the LogAnalyzer example

```
namespace AOUT.CH4.LogAn
{
   public class LogAnalyzer
   {
      private IWebService service;

      public LogAnalyzer(IWebService service)
      {
         this.service = service;
      }

      public void Analyze(string fileName)
      {
         if(fileName.Length<8)
         {
            service.LogError("Filename too short:" + fileName);
         }
      }
   }
}
```

Test 1/2

```
namespace AOUT.Ch4.LogAn.Test
{
   [TestFixture]
   public class LogAnalyzerTests
   {
      [Test]
      public void Analyze_TooShortFileName_CallsWebService()
      {
         MockService mockService = new MockService();
         LogAnalyzer log = new LogAnalyzer(mockService);
         string tooShortFileName="abc.ext";
         log.Analyze(tooShortFileName);

         Assert.AreEqual("Filename too short:abc.ext",mockService.LastError);
      }
   }
```

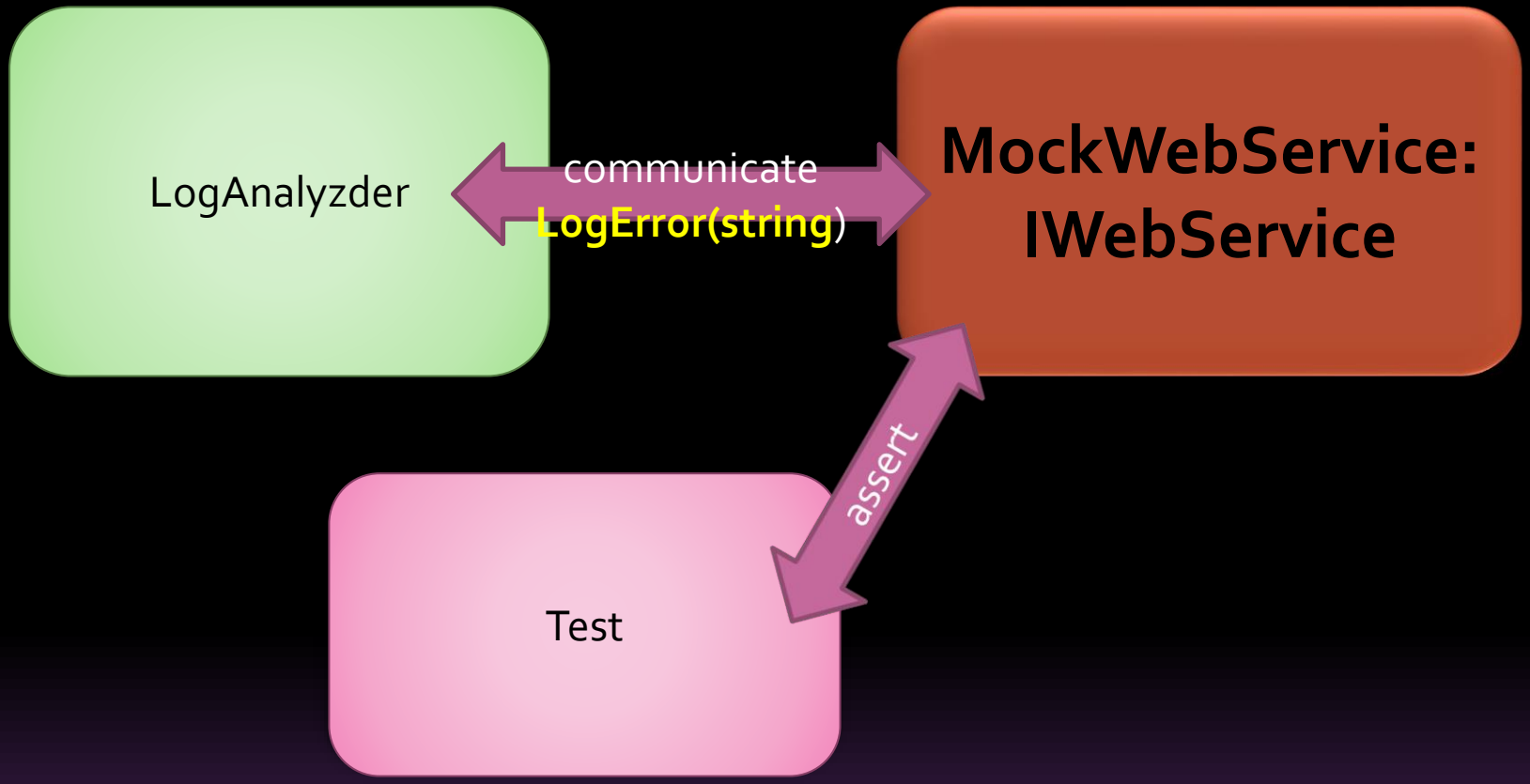
Test 2/2

```
public class MockService:IWebService
{
   public string LastError;

   public void LogError(string message)
   {
      LastError = message;
   }
}
```

Mock object keeps the history for later to be asserted by test

LogAnalyzer with a Mock



The CUT communicates with the mock object and all communication is recorded In the mock. The test uses the mock object to verify that the test passes

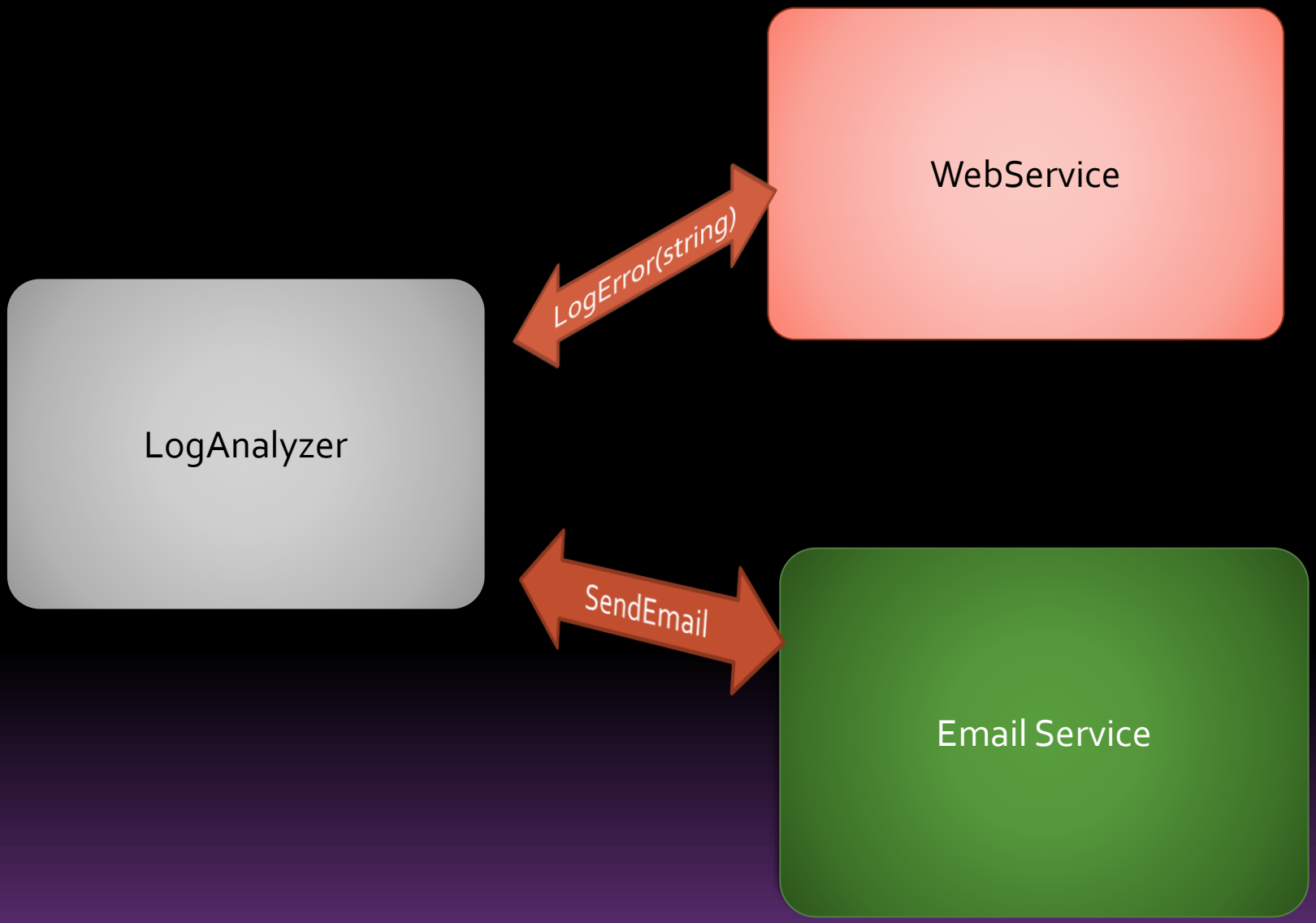
When stub and mock works together

- An example – suppose the logic of logAnalyzer is more complicated

```
public void Analyze(string fileName)
    {
      if(fileName.Length<8)
      {
        try
        {
          service.LogError("Filename too short:" + fileName);
        }
        catch (Exception e)
        {
          email.SendEmail("a","subject",e.Message);
        }
      }
    }
```

You tell the webservice to log an error. However, the web service may throw an exception

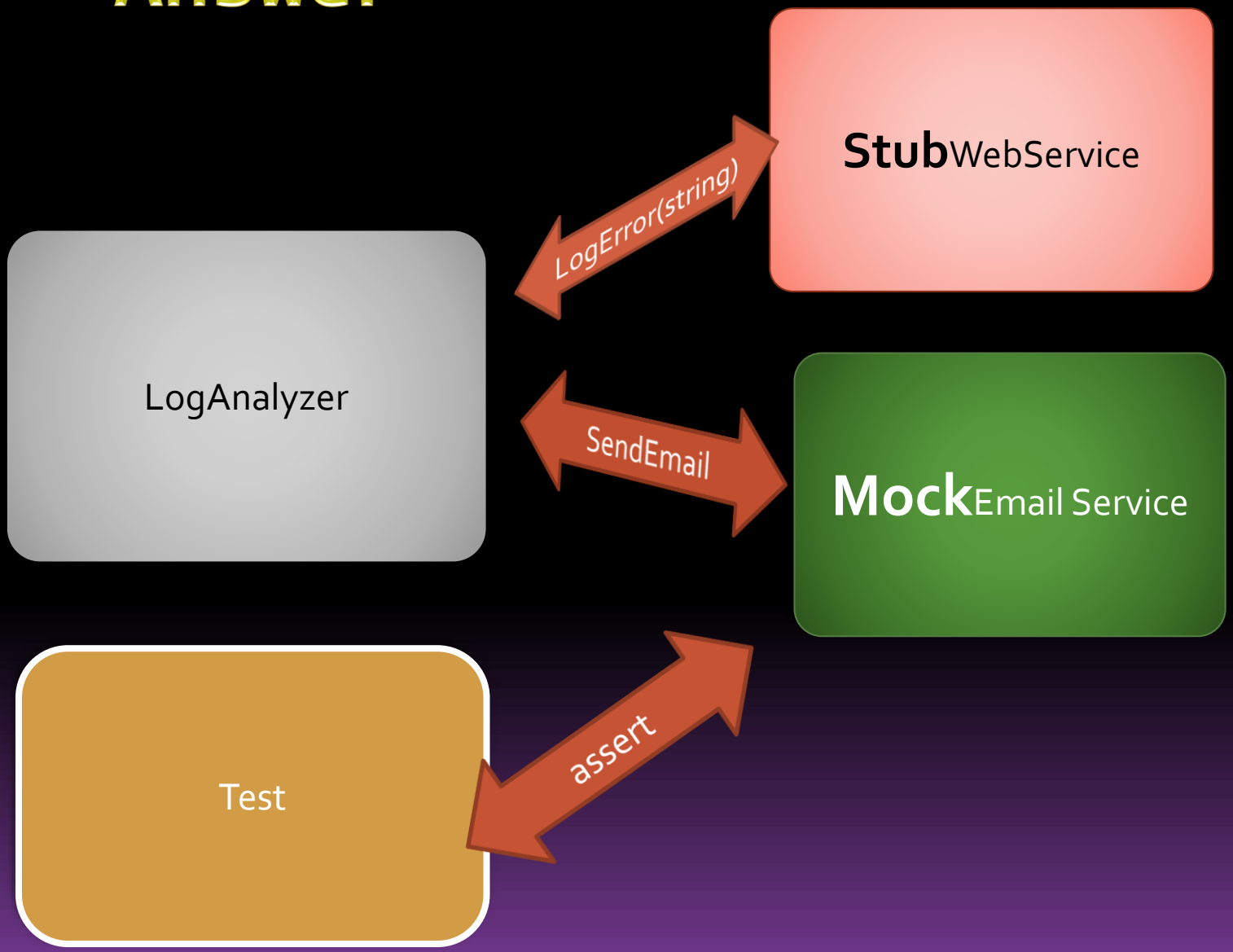
When the webservice is down we need to send an email to someone


WebService
LogError(string)
LogAnalyzer
SendEmail
Email Service

Questions

1. How can we replace the web service
2. How can we simulate an exception from the web service so that we can test the call to the email service
3. How will we know that the email service was called correctly or at all?

Answer


StubWebService
LogError(string)
LogAnalyzer
SendEmail
MockEmail Service
Test
assert

Notes

- In the above scenario, stubwebservice is a stub because test does not assert
- In the above scenario, emailservice is a Mock, because test assert the history it collects.

The code

```
public class LogAnalyzer2
  {
    private IWebService service;
    private IEmailService email;

    public IWebService Service
    {
      get { return service; }
      set { service = value; }
    }

    public IEmailService Email
    {
      get { return email; }
      set { email = value; }
    }
```

```
public void Analyze(string fileName)
    {
      if(fileName.Length<8)
      {
        try
        {
          service.LogError("Filename too
short:" + fileName);
        }
        catch (Exception e)
        {
    email.SendEmail("a","subject",e.Message);
        }
      }
    }
```

Problems in practice

1. It takes time to write mock and stubs
2. It is difficult to write stubs and mocks for classes and interfaces that have many methods, properties, and events
3. To save state for multiple calls of a mock method, you need to write a lot of boilerplate code to save the data
4. If you want to verify all parameters on a method call, you need to write multiple asserts. If the first assert fails, the others will never run because a failed assert throw an exception
5. It is hard to reuse mock and stub code for other tests.