Introducing Stubs

External Dependency – An external dependency is an object in the system that the code under test interacts with, and over which you have no control. (Common examples are file-systems, threads, memory, time, database, and so on.)

Stub – 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. **Stubs can't fail tests!**

Example:

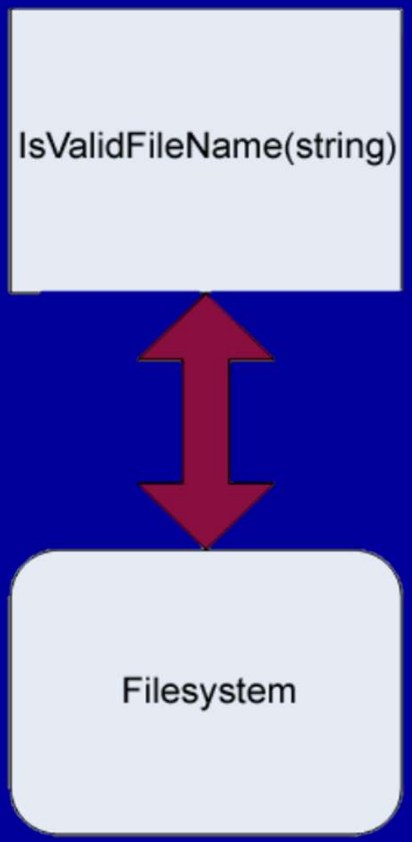
LogAnalyzer class application can be configured to handle multiple log filename extensions using a special adapter for each file. Let's assume that the allowed filenames are stored somewhere on disk as a configuration setting for the application.

```
public class LogAnalyzer
{
    5 references
    public bool IsValidLogFileName(string fileName)
    {
        if (!File.Exists(fileName))
        {
            throw new Exception("No log file with that name exists");
        }
        if(!fileName.ToLower().EndsWith(".slf"))
        {
            return false;
        }


        return true;
    }
}
```

Introducing Stubs (*Cont.*)

The problem that arises is that a test, that will be built, depends on the filesystem. As a result we're performing an integration test, and we have all the associated problems: integration tests are slower to run, they need configuration, they test multiple things, and so on.




```
public class LogAnalyzer
{
    5 references
    public bool IsValidLogFileName(string fileName)
    {
        if (!File.Exists(fileName))
        {
            throw new Exception("No log file with that name exists");
        }
        if(!fileName.ToLower().EndsWith(".slf"))
        {
            return false;
        }

        return true;
    }
}
```

This is the essence of test-inhibiting design: the code has some dependency on an external resource, which might break the test even though the code's logic is perfectly valid. In legacy systems, a single class or method might have many dependencies on external resources over which your test code has little, if any, control.

Back

Introducing Stubs (*Cont.*)

"There is no object-oriented problem that cannot be solved by adding a layer of indirection, except, of course, too many layers of indirection."

A lot of the "art" in the art of unit testing is about finding the right place to add or use a layer of indirection to test the code base.

The art also involves figuring out when a layer of indirection already exists instead of having to invent it, or knowing when not to use it because it complicates things too much.

There's a definite pattern for breaking the dependency:

1. **Find the interface or API that the object under test works against.**
2. **Replace the underlying implementation of that interface with something that you have control over.**

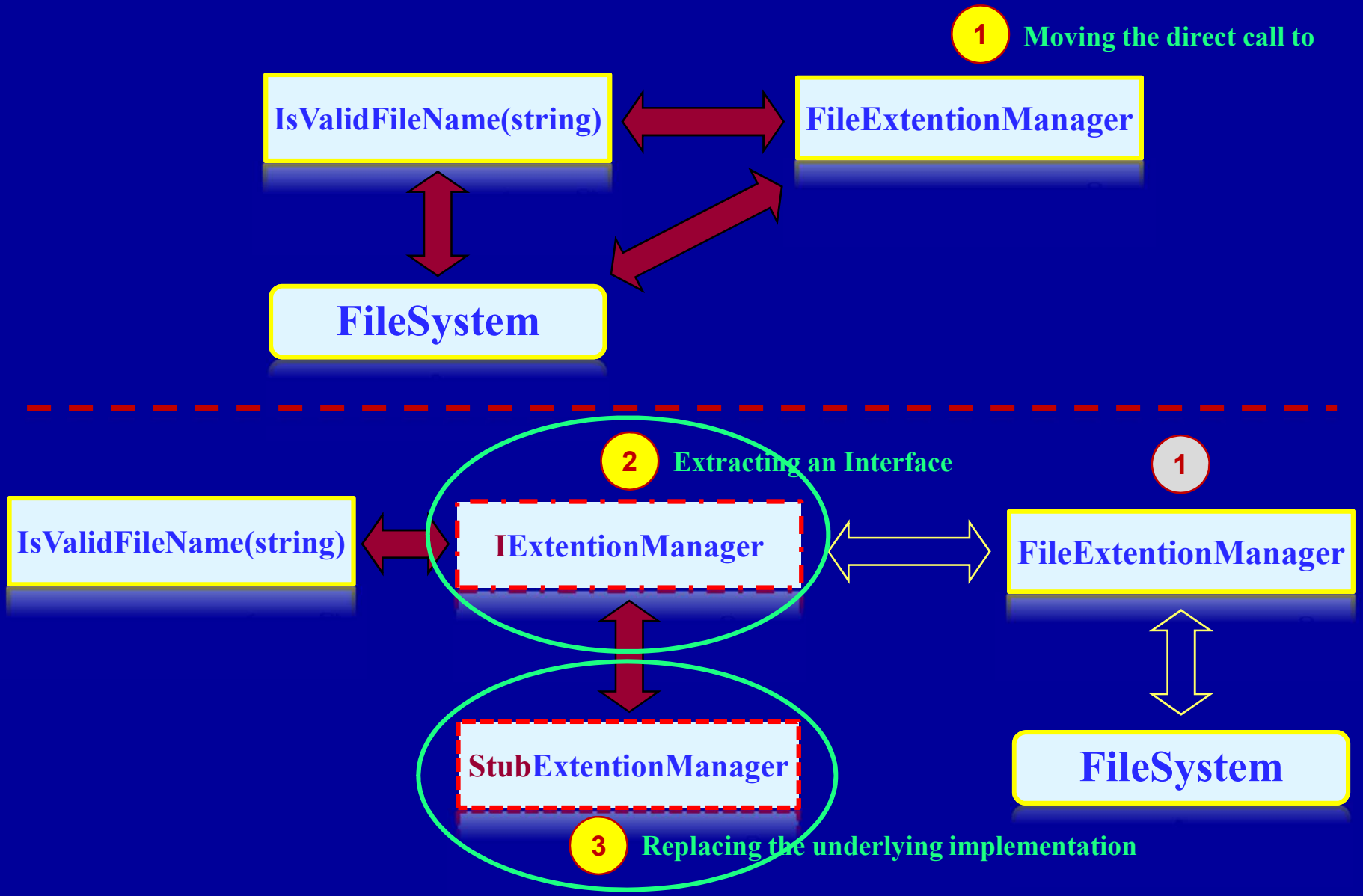
Notes: **All these technics are also called "Dependency Injection"**

Introducing Stubs (*Cont.*)

Bit more details about pattern for breaking the dependency:

1. **Find the interface that the method under test works against. (In this case, "interface" isn't used in the pure object-oriented sense; it refers to the defined method or class being collaborated with.) In our *LogAnalyzer* project, this is the filesystem configuration file.**
2. **If the interface is directly connected to our method under test (as in this case - we're calling directly into the filesystem), make the code testable by adding a level of indirection to the interface. In our example, moving the direct call to the filesystem to a separate class (such as for example *FileExtensionManager*) would be one way to add a level of indirection.**
3. **Replace the underlying implementation of that interactive interface with something that you have control over. In our case, we'll replace the instance of the class that our method calls (*FileExtensionManager*) with a stub class that we can control (*StubExtensionManager*), giving our test code control over external dependencies.**

Introducing a Stubs to break the dependency



Refactoring Design to be more Testable

Refactoring – A Refactoring means changing a piece of code without changing its functionality. If you've ever renamed a method, you've done refactoring. If you've ever split a large method into multiple smaller method calls, you've refactored your code. The code still does the same thing, but it becomes easier to maintain, read, debug, and change.

Seams – Seams are places in a code where you can plug in different functionality, such as stub classes.

If we want to break the dependency between a code under test and the filesystem, we can use common design patterns, refactorings, and techniques, and introduce one or more seams into the code. It's just needed to make sure that the resulting code does exactly the same thing. Below 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.

Refactoring Design to be more Testable (*Cont.*)

- ***Extract an interface to allow replacing underlying implementation***

  In this technique, we need to break out the code that touches the filesystem into a separate class. That way we can easily distinguish it and later replace the call to that class from our tested function. Listing below shows the places where we need to change the code.

```
public bool IsValidLogFileName(string fileName)
{
    FileExtensionManager mgr =
              new FileExtensionManager();
    return mgr.IsValid(fileName);
}

class FileExtensionManager
    {
        public bool IsValid(string fileName)
        {
        //read some file here
        }
    }
```

Uses the extracted class

Defines the extracted class

Refactoring Design to be more Testable (*Cont.*)

Next, we can tell our class under test that, instead of using the concrete *FileExtensionManager* class, it will deal with some form of *ExtensionManager*, without knowing its concrete implementation. In *.NET*, this could be accomplished by either using a base class or an interface that *FileExtensionManager* would extend.

```
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

Refactoring Design to be more Testable (*Cont.*)

We've simply created an interface with one *IsValid(string)* method, and made *FileExtensionManager* implement that interface. It still works exactly the same way, only now we can replace the "real" manager with our own "stub" manager to support our test.
We still haven't created the stub extension manager, so let's create that right now.

Implements the IExtensionManager

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

This stub extension manager will always return true, no matter what the file extension is. We can use it in our tests to make sure that no test will ever have a dependency on the filesystem.

Refactoring Design to be more Testable (*Cont.*)

Now we have an interface and two classes implementing it, but our method under test still calls the real implementation directly:

```
public bool IsValidLogFileName(string fileName)
        {
            IExtensionManager mgr = new FileExtensionManager();
            return mgr. IsValid (fileName);
        }
```

Somehow it needs to tell our method to talk to our implementation rather than the original implementation of *IExtensionManager*. We need to introduce a seam into the code, where we can plug in our stub.

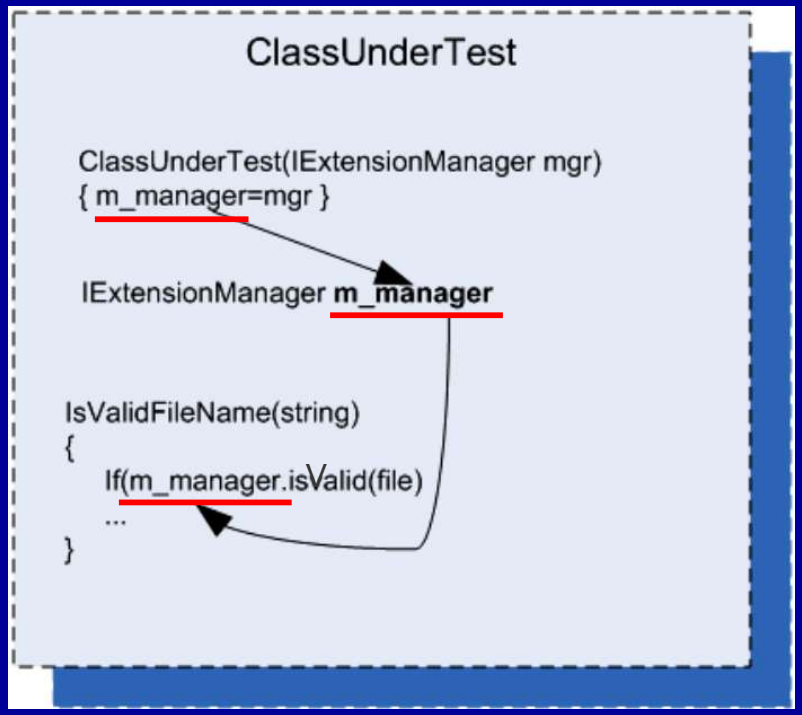
Inject stub implementation into a class under test

There are several proven ways to create interface-based seams in our code-places where we can inject an implementation of an interface into a class to be used in its methods. Here are some of the most notable ways:

- ✓ Receive an interface at the constructor level and save it in a field for later use.
- ✓ Receive an interface as a property "get" or "set" and save it in a field for later use.
- ✓ Receive an interface just before the call in the method under test using:
  - ❑ a parameter to the method (parameter injection).
  - ❑ a factory class.
  - ❑ a local factory method.
  - ❑ variations on the preceding techniques.

Receive an interface at the constructor level (constructor injection)

In this scenario, we add a new constructor (or a new parameter to an existing constructor) that will accept an object of the interface type we extracted earlier (*IExtensionManager*). The constructor then sets a local field of the interface type in the class for later use by our method or any other.



Flow of injection via a constructor

Receive an interface at the constructor level (constructor injection)

```
public class LogAnalyzer  // <====== Defines production code
{
    private IExtensionManager manager;
    0 references
    public LogAnalyzer()  // <====== Creates object in production code
    {
        manager = new FileExtensionManager();
    }
    1 reference
    public LogAnalyzer(IExtensionManager mgr)  // <====== Defines constructor that can be called by tests
    {
        manager = mgr;
    }
    1 reference
    public bool IsValidLogFileName(string fileName)
    {
        return manager.IsValid(fileName);
    }
}
4 references
public interface IExtensionManager
{
    2 references
    bool IsValid(string fileName);
}
```

Problems with constructor injection

Problems can arise from using constructors to inject implementations. If your code under test requires more than one stub to work correctly without dependencies, adding more and more constructors (or more and more constructor parameters) becomes a hassle, and it can even make the code less readable and less maintainable.

Example: suppose *LogAnalyzer* also had a dependency on a web service and a logging service in addition to the file extension manager. The constructor might look like this:

```
public LogAnalyzer(IExtensionManager mgr, ILog logger, IWebService
  service)
 {
    //    this constructor can be called by tests
           manager = mgr;
        log= logger;
        svc= service;
   }
```

Problems with constructor injection (*Cont.*)

One solution to these problems is to create a special class that contains all the values needed to initialize a class, and to have only one parameter to the method: that class type. That way, you only pass around one object with all the relevant dependencies. (This is also known as a *parameter object refactoring*.)

Another possible solution to these problems is using *Inversion of Control* (*IoC*) containers. You can think of *IoC* containers as "*smart factories*" for your objects (although they are much more than that).

Notes: such containers provide special factory methods that take in the type of object you'd like to create and any dependencies that it needs, and then initialize the object using special configurable rules such as what constructor to call, what properties to set in what order, and so on.

Now, imagine that you have 50 tests against your constructor, and you find another dependency you had not considered, such as a factory service for creating special objects that works against a database. You'd have to create an interface for that dependency and add it as a parameter to the current constructor, and you'd also have to change the call in 50 other tests that initialize the code. At this point, your constructor could use a facelift. Fortunately, using *property getters* and *setters* can solve this problem easily.

Inversion of Control

IoC – Inversion of Control, or IoC, is an abstract principle describing an aspect of some software architecture designs in which the flow of control of a system is inverted in comparison to procedural programming.

That means is that in procedural programming a chunk of code that uses, or consumes, another chunk of code is in control of the process.
It knows exactly what piece of code, what method in what class, it uses. And in doing so it is also quite likely that it knows about some implementation details in the code it uses.

Example:

Questions:

- Does "X" really need to know that it uses "Y"?
- Isn't it enough that "X" knows that it uses something that has the behavior, the methods, properties etc., of "Y" without knowing who actually implements the behavior?

Inversion of Control (*Cont.*)

By extracting an abstract definition of the behavior used by "X" in "Y", illustrated as "I" below, and letting the consumer "X" to use an instance of "I" instead of "Y", it can continue to do what it does without having to know the specifics about "Y".

What benefits can we get from this operation?

- First - "X" is not dependent on "Y" anymore and it's therefore less likely that we'll have to make changes to "X" when we make changes in "Y" as we're less likely to write code in "X" that relies on implementation details in "Y".
- Second - "X" is now much more flexible since we can change an implementation of "I" without changing anything in "X".
- It's possible to isolate the code when creating unit tests!
- In the setup phase of tests, it's possible to chose which implementation of "I ", "X" will use.

About using 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.

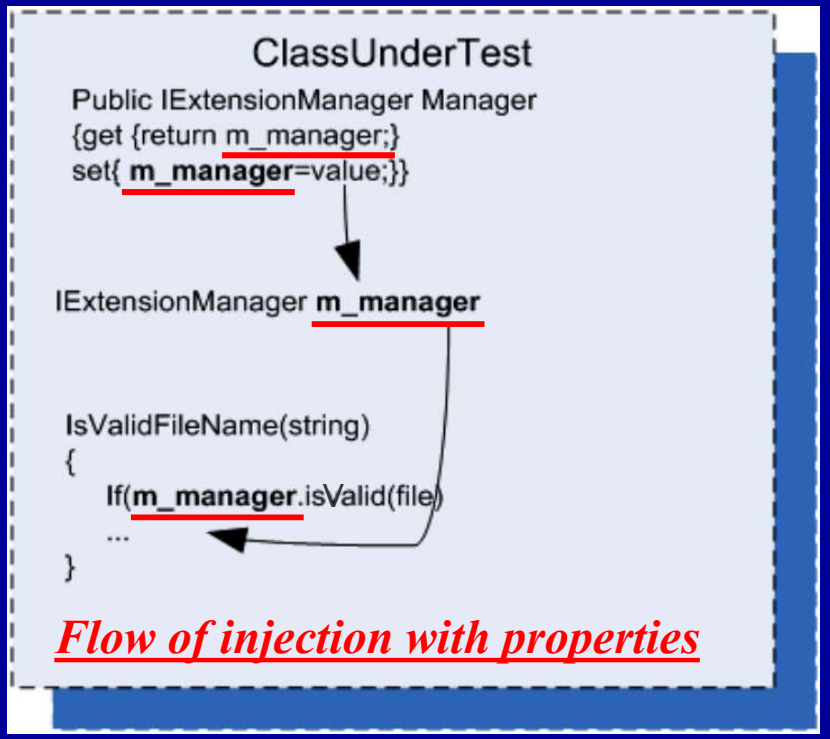
If you choose to use constructor injection, you'll probably also want to use *IoC* containers. This would be a great solution if all code in the world were using *IoC* containers, but most people don't know what the Inversion of Control principle is.

Refactoring Design to be more Testable

❖ *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.

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



Flow of injection with properties

Using this technique (also called *dependency injection*, a term that can also be used to describe the other techniques), our test code would look quite similar to that we considered before, and which used constructor injection. But the code, shown in next slide, is more readable and simpler to achieve.

Refactoring Design to be more Testable (*Cont.*)

Let's look at the code listing below:

```
public class LogAnalyzer
{
    private IExtensionManager manager;

    1 reference
    public LogAnalyzer()
    {
        manager = new FileExtensionManager();
    }

    0 references
    public IExtensionManager ExtensionManager // <====== Allows setting dependency via a property
    {
        get { return manager; } // <====== Allows setting dependency via a property
        set { manager = value; } // <====== Allows setting dependency via a property
    }

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

Like constructor injection, property injection has an effect on the API design in terms of defining which dependencies are required and which aren't. By using properties, you're effectively saying, "This dependency isn't required to operate this type."

Constructor & Properties Injection

Flow of injection via a constructor

```
public class LogAnalyzer  // <====== Defines product
{
    private IExtensionManager manager;
    0 references
    public LogAnalyzer()  // <====== Creates object
    {
        manager = new FileExtensionManager();
    }
    1 reference
    public LogAnalyzer(IExtensionManager mgr)  // <=
    {
        manager = mgr;
    }
    1 reference
    public bool IsValidLogFileName(string fileName)
    {
        return manager.IsValid(fileName);
    }
}
4 references
public interface IExtensionManager
{
    2 references
    bool IsValid(string fileName);
}
```

Flow of injection with properties

```
public class LogAnalyzer
{
    private IExtensionManager manager;

    1 reference
    public LogAnalyzer()
    {
        manager = new FileExtensionManager();
    }

    0 references
    public IExtensionManager ExtensionManager //  <=
    {
        get { return manager; } // <====== Allows se
        set { manager = value; } // <====== Allows
    }

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

Notes: Use this technique when you want to signify that a dependency of the class under test is optional, or if the dependency has a default instance that doesn't create any problems during the test.

Introducing Mocks

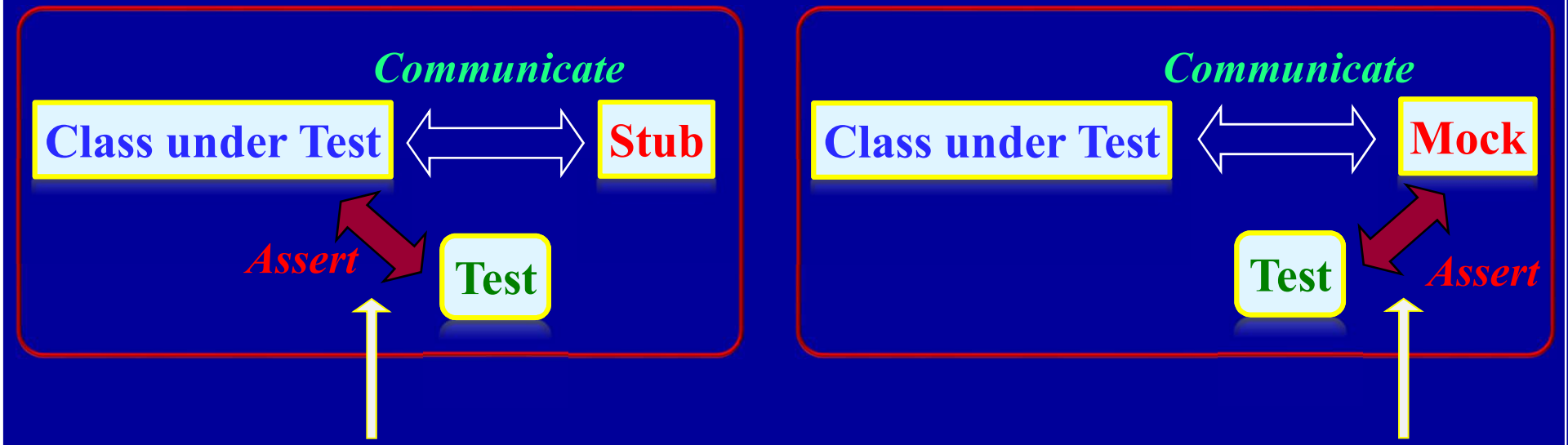
Interaction testing is testing how an object sends input to or receives input from other objects - how that object interacts with other objects.

A mock object is a fake object in the system that decides whether the unit test has passed or failed. It does so by verifying whether the object under test interacted as expected with the fake object. There's usually no more than one mock per test.

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.

Introducing Mocks (*Cont.*)

The basic difference between Stubs and Mocks is that stubs can't fail tests, and mocks can.

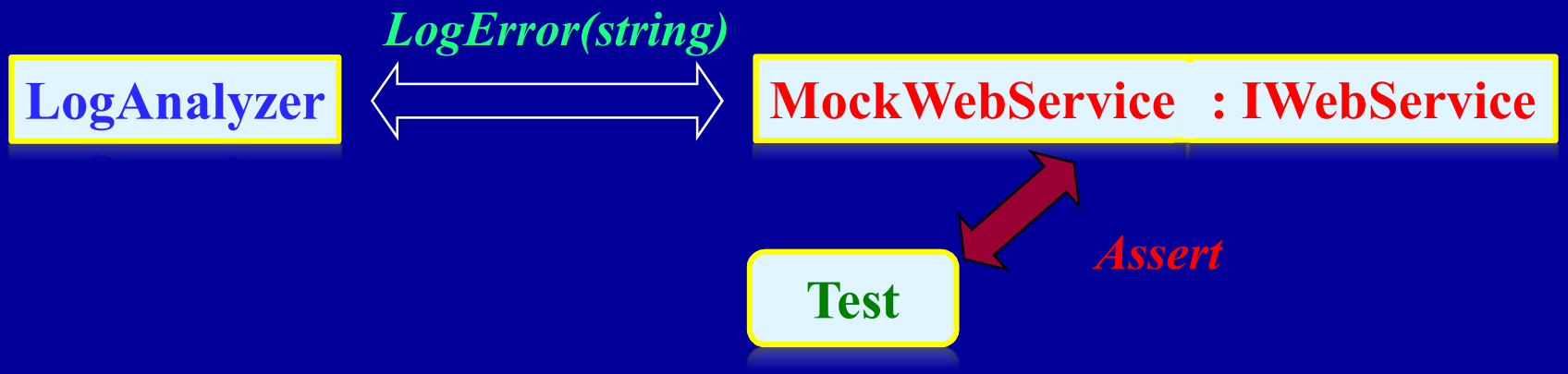


By using a stub, the assert is performed on the class under test. On the other hand, the test will use a mock object to verify whether the test failed or not.

The easiest way to tell we're dealing with a stub is to notice that the stub can never fail the test. The asserts, the test uses, are always against the class under test.

A Simple Manual Mock Example

Let's add a new requirement to our *LogAnalyzer* class. This time, it will have to interact with an external web service that will receive an error message whenever the *LogAnalyzer* encounters a filename whose length is too short.



First off, let's extract a simple interface that we can use in our code under test, instead of talking directly to the web service. It'll serve us when we want to create stubs as well as mocks and will let us avoid an external dependency we have no control over:

```
public interface IWebService
{
    void LogError(string message);
}
```

A Simple Manual Mock Example (*Cont.*)

Next, let's create the mock object itself. It may look like a stub, but it contains one extra bit of code that makes it a mock object.

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

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

Our mock implements an interface, as a stub does, but it saves some state for later, so that our test can then assert and verify that our mock was called correctly. This is actually called a Test Spy.

A Simple Manual Mock Example (*Cont.*)

The test might look like the listing below:

```
[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);
}
```

Asserts against mock object

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

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

```
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);
        }
    }
}
```

Notes:

- The assert is performed against the mock object, and not against the ***LogAnalyzer*** class.
- The tests aren't writing directly inside the mock object code.