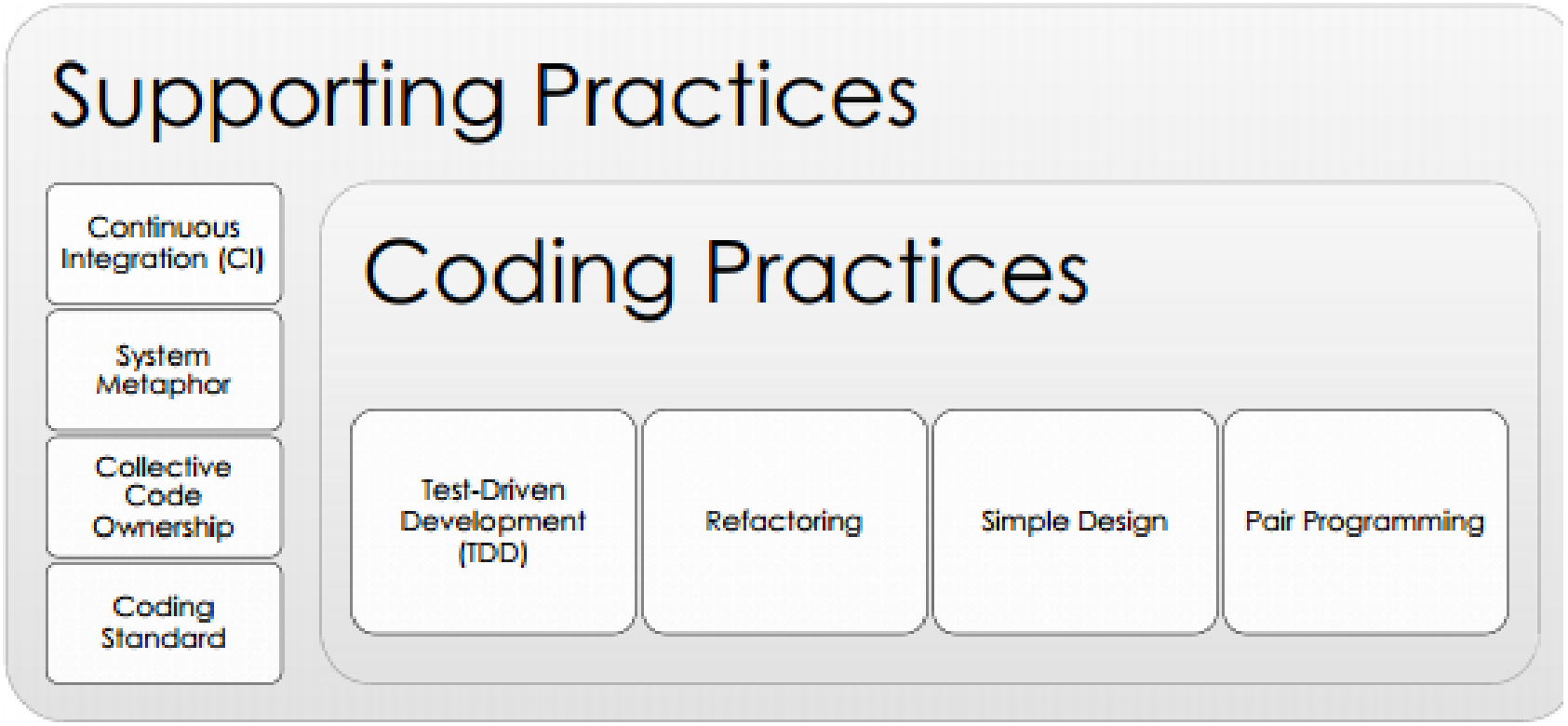
Introduction to Software Engineering
Unit Test Patterns

CMPS115 – Summer 2017
Richard Jullig

Acknowledgments

- Materials drawn from
  - Gerard Meszaros, *xUnit Test Patterns* (2007)
    - Ch. 11: Using Test Doubles
  - Roy Osherove, The Art of Unit Testing (2014, 2nd ed.)
- See also
  - Michael Feathers, *Working Effectively with Legacy Code* (2005)

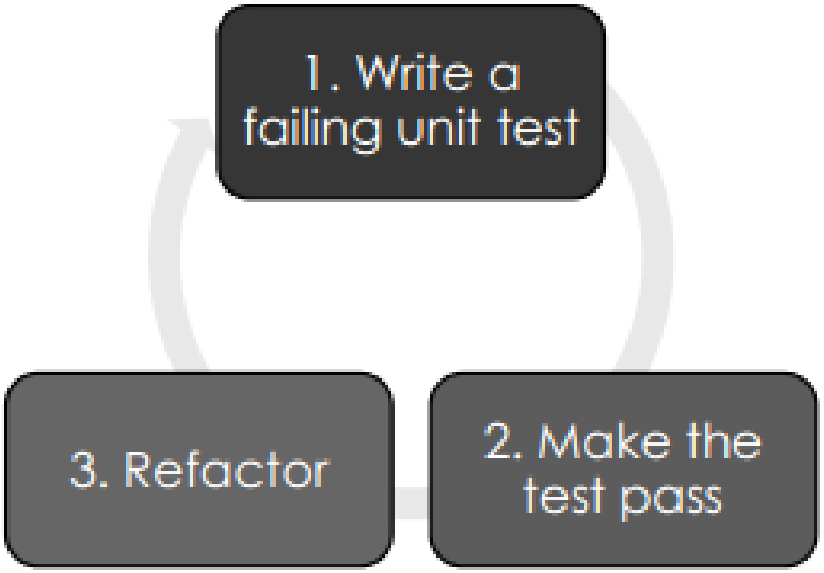
Technical Practices



- Unit Testing
  - Agile: essential part of TDD
  - General: testing of elementary components; performed by developers

Test-Driven Development (TDD)

"We only write new code when we have a test that doesn't work"



"TDD is primarily a design technique with a side effect of ensuring that your source code is thoroughly unit tested"
– Scott W. Ambler

- Turns testing into a design activity
  - Consumption awareness: test case code represents how you would like to access the functionality
  - Lead to programming by intention
- Provides continuous feedback
  - *"does it work?", "is it well structured?"*

Three Laws of TDD

Do not write any production code unless it is to make a failing unit test pass.

Do not write any more of a unit test than is sufficient to fail; and build failures are failures.

Do not write any more production code than is sufficient to pass the one failing unit test.

Desirable Test Characteristics (F.I.R.S.T.)

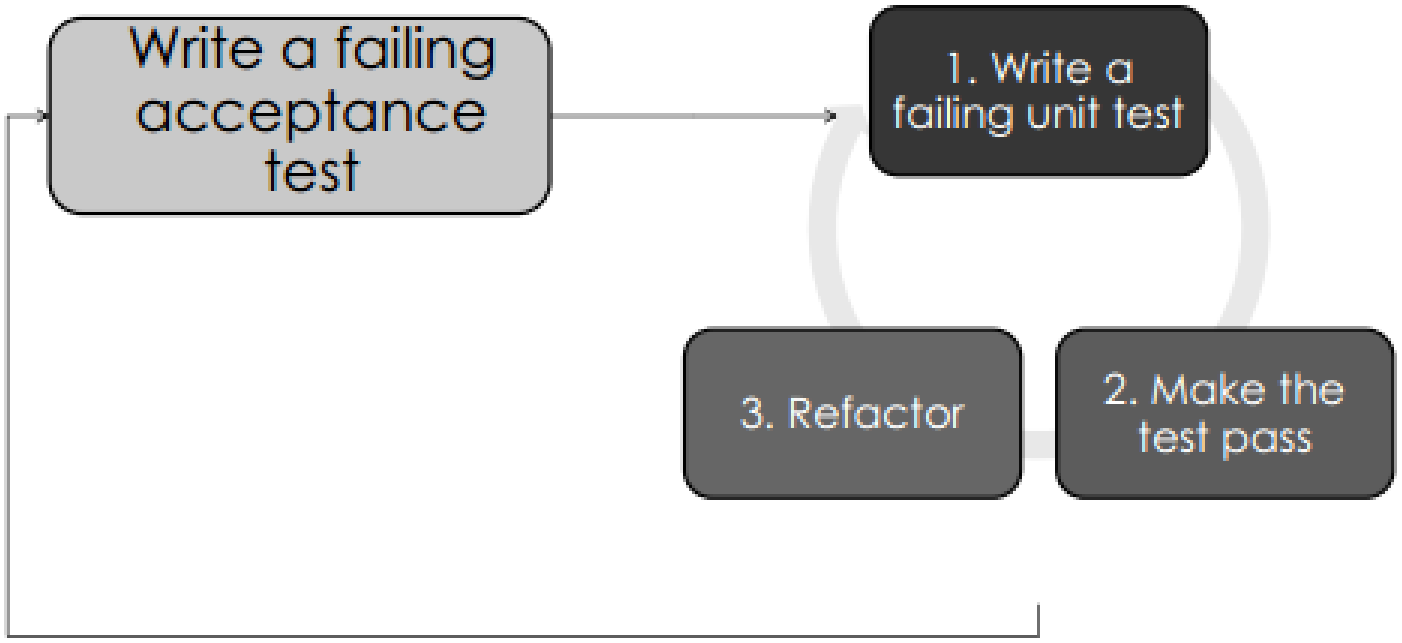
- Fast
  - tests will run frequently
  - Small and simple: test one concept at a time
- Independent
  - No dependencies between tests
  - Tests can run in any order
  - Simplifies failure analysis (debugging)
- Repeatable
  - Tests can run at any time, in any order
- Self-Validating
  - Test either pass or fail (Boolean result)
- Timely
  - Write the tests when you need them
  - In TDD: write test first, then code

Clean Tests

- Test code is code, too.
- Test code requires same care as production code
  - Readable
  - Understandable
  - Self-explanatory; clear intent

- Dirty test code
  - Same, or worse than, no tests
  - Ill-structured tests increasingly harder to change and grow

Bigger Picture: Acceptance TDD



- The code for passing the acceptance test is built incrementally based on several unit tests
- Acceptance TDD closely related to Behavior Driven Dev (BDD)

Need to isolate components for Unit testing

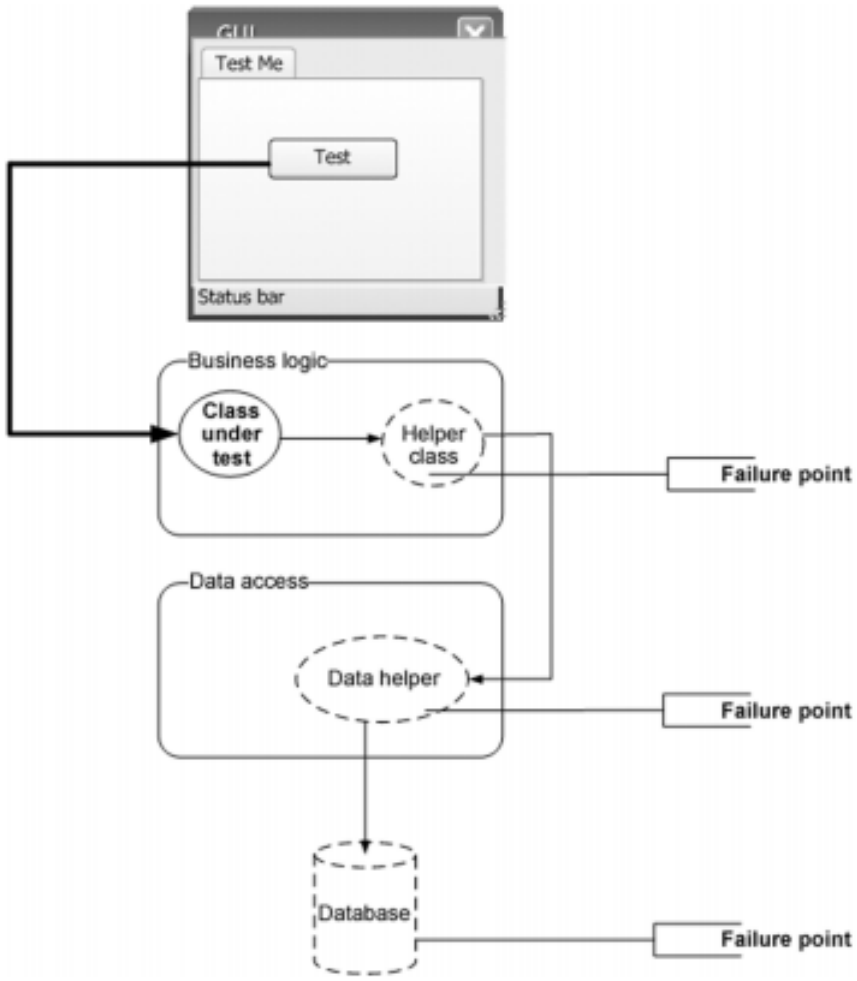
- Testing clusters of components (e.g. classes) is hard
  - many execution paths to cover
  - Test data to drive specific paths may be hard to generate
    - E.g. may be hard to trigger exceptions
- Integration testing
  - Testing clusters of components
  - Attempt only with Unit tested components

- In Unit tests: need to
  - Control indirect inputs from depended-on components (DoCs) to SUT (software/system under test, the dependent component)
  - Observe and verify indirect outputs from SUT to DoCs

Examples for Test Stubs, Spies, and Mocks

- Material from
  Osherove, *The art of unit testing* (2nd ed. 2014)
- Primarily Chapters 3 and 4

What makes integration testing difficult



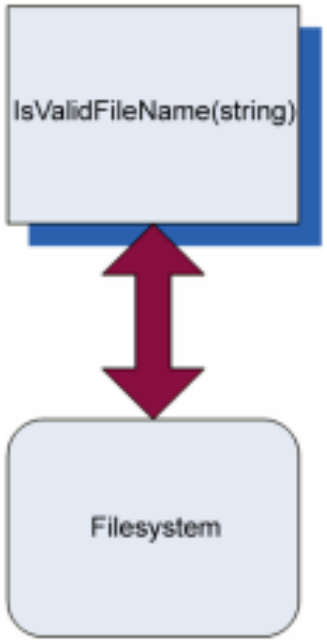
- Long execution paths
  - Through several service layers
  - Each call to service represents a failure point
- Failure analysis non-trivial
  - Not obvious where the failure occurred
    - Layer
    - Interaction between layers

Non scholae sed vitae discimus.

Sample Application: LogAnalyzer

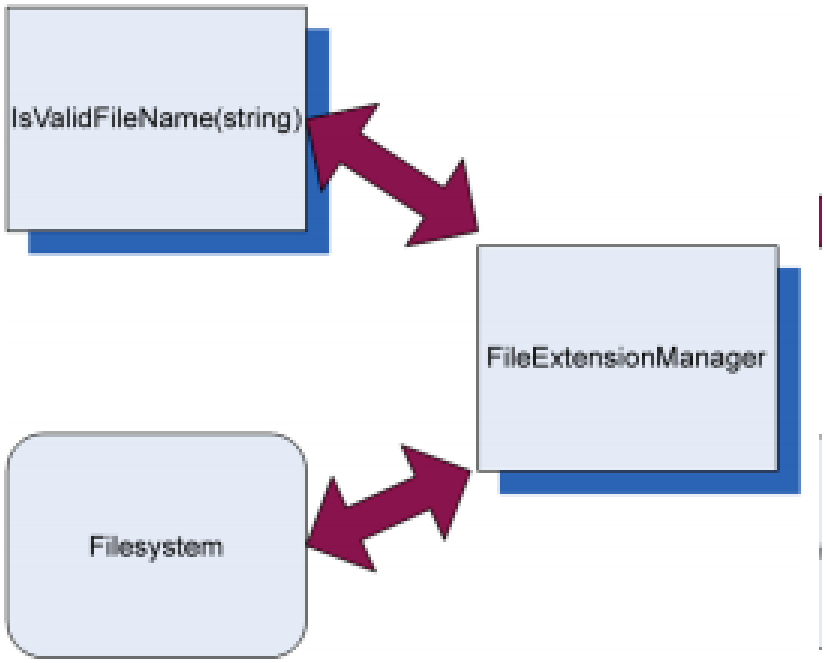
- Method IsValidFileName in class LogAnalyzer

```
public bool IsValidLogFileName(string fileName)
        {
        //read through the configuration file
        //return true if configuration says extension is supported.
        }
```



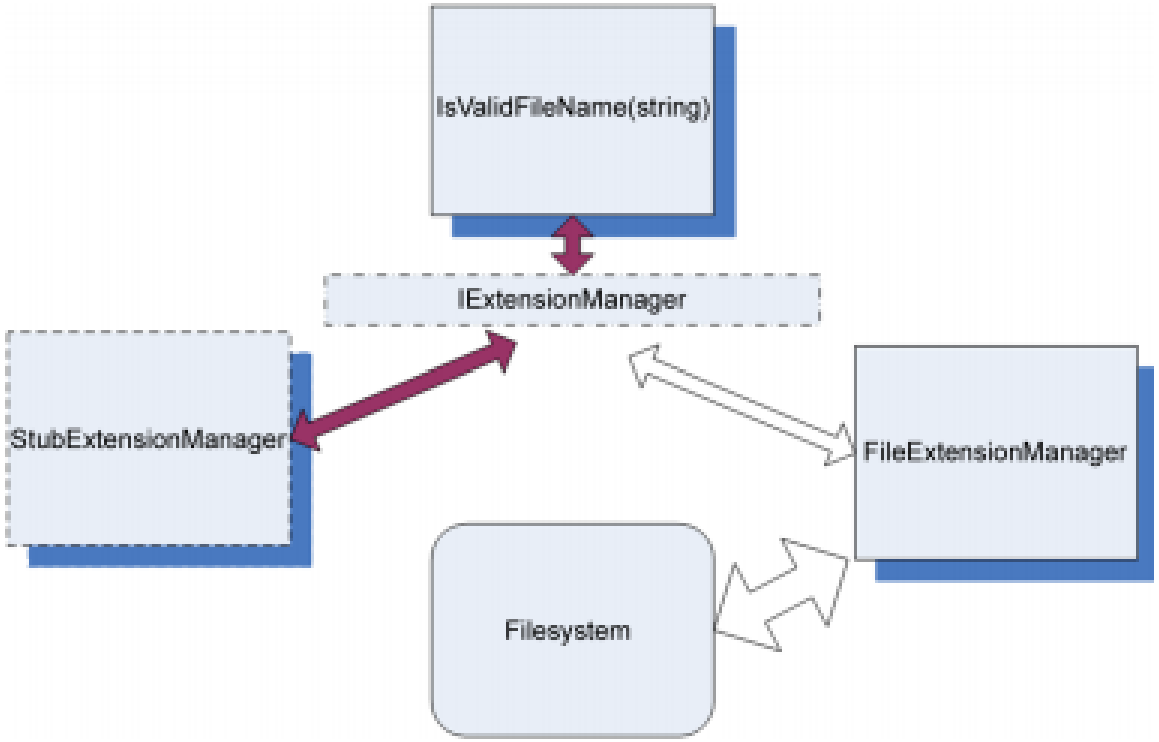
- SUT: IsValidFileName
- Has direct dependency on file system
- Not clear how to unit test
  - Requires integration test
- Test-inhibiting design
  - Dependency on external resource may make tests fail even though SUT works correctly
- Solution
  - Introduce abstraction layer
  - Between SUT and external resource(s)

Adding Abstraction Layer for Testability



- FileExtensionManager
  - Encapsulates access to file system
- IsValidFileName
  - No longer directly dependent on file system
- Isolate IsValidFileName for unit testing
  - Introduce ExtensionManager interface
  - Replace FileExtensionManager by stub

Introduce interface to make DoC replaceable



- ExtensionManager interface creates a seam.
- Refactoring does not change functionality of code
  - Keep the old version; confirm refactoring with regression tests
- For unit testing, replace FileExtensionManager by StubExtensionManager

Encapsulate Filesystem access – Extract class

```
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

- IsValidLogFileName still depends on concrete FileExtensionManager
- Remove this dependency by introducing ExtensionManager interface

Non scholae sed vitae discimus.

Introduce ExtensionManager Interface

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

//the unit of work under test:
public bool IsValidLogFileName(string fileName)
    {
        IExtensionManager mgr =
                new FileExtensionManager();
        return mgr.IsValid(fileName);
    }
```

Implements the interface

Defines the new interface

Defines a variable as the type of the interface

- Note:
  any object that implements the ExtensionManager interface can serve as mgr

Define Extensionmanager Stub

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

Implements IExtensionManager

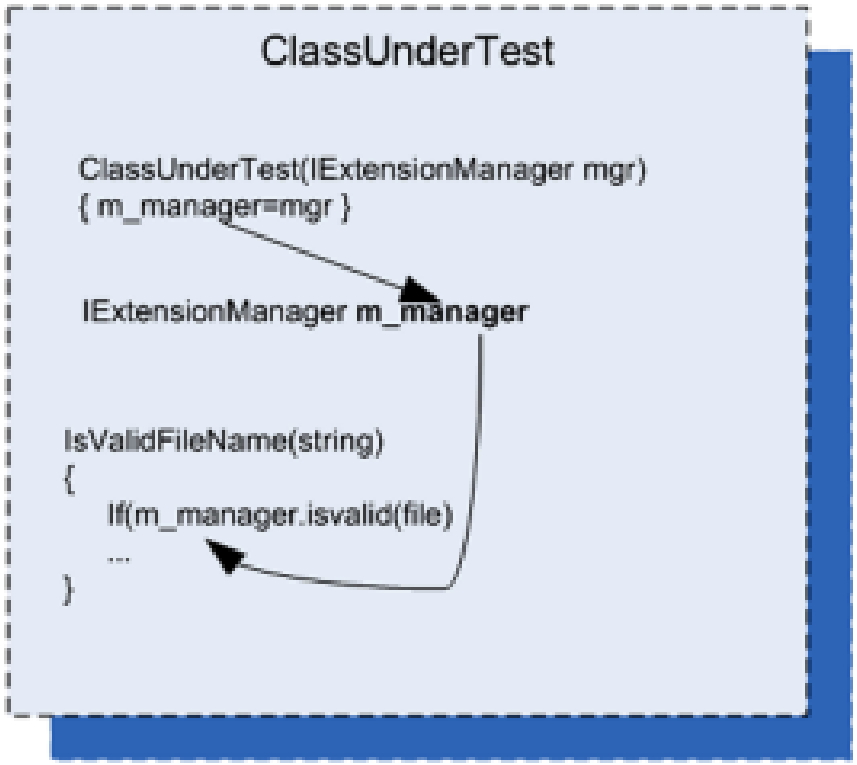
- ExtensionManager stub always returns true
- Still need to inject the stub into the SUT

- Terminology:
  - Osherove uses fake for an object that is a stub or a mock object
  - Meszaros uses fake for “cheap” replacement of an “expensive” DoC with same functionality

Dependency Injection: Stub or Mock into SUT

- Constructor injection
- Setter injection
- Injection before method call
- Extract and Override
  - See Osherove, Chapter 3
- Parameter injection
  - Pass DoC to method under test as parameter to method call

Constructor Injection



- Constructor of object on which method under test is called
  takes DoC as parameter
- DoC is provided to SUT by the client
  (i.e. component that calls the SUT)

Non scholae sed vitae discimus.

Constructor Injection: Example (1)

```
public class LogAnalyzer
{
        private IExtensionManager manager;

        public LogAnalyzer(IExtensionManager mgr)
        {
            manager = mgr;
        }
        public bool IsValidLogFileName(string fileName)
        {
            return manager.IsValid(fileName);
        }
}

public interface IExtensionManager
{
        bool IsValid(string fileName);
}
```

Defines production code

Defines constructor that can be called by tests

- When creating a LogAnalyzer object, an Extensionmanager must be provided to the constructor

Constructor Injection: Example (2)

```
public class LogAnalyzerTests
{
        [Test]
         public void
         IsValidFileName_NameSupportedExtension_ReturnsTrue()
         {
             FakeExtensionManager myFakeManager =
                         new FakeExtensionManager();
             myFakeManager.WillBeValid = true;

             LogAnalyzer log =
                 new LogAnalyzer (myFakeManager);

             bool result = log.IsValidLogFileName("short.ext");
             Assert.True(result);
         }
}

internal class FakeExtensionManager : IExtensionManager
{
         public bool WillBeValid = false;

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

Sets up stub to return true

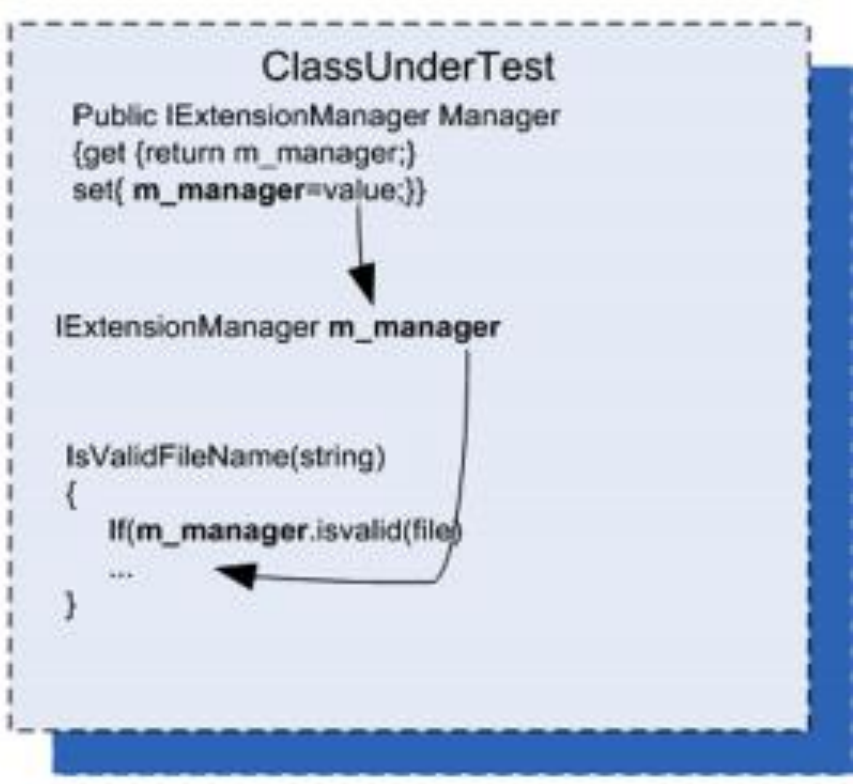
Sends in stub

Defines stub that uses simplest mechanism possible

- Configurable stub
- Configured in test setup

Non scholae sed vitae discimus.

Setter Injection



- Use property setter to configure object under test with suitable DoC, stub, or mock object
- Preferable to constructor injection when SUT has multiple possible dependencies
  not all of which may be needed in all contexts (by all tests)

Setter Injection: Example

```
public class LogAnalyzer
{
    private IExtensionManager manager;

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

    public IExtensionManager ExtensionManager
    {
        get { return manager; }
        set { manager = value; }
    }

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

Allows setting dependency via a property

- LogAnalyzer constructor initializes manager attribute to DoC

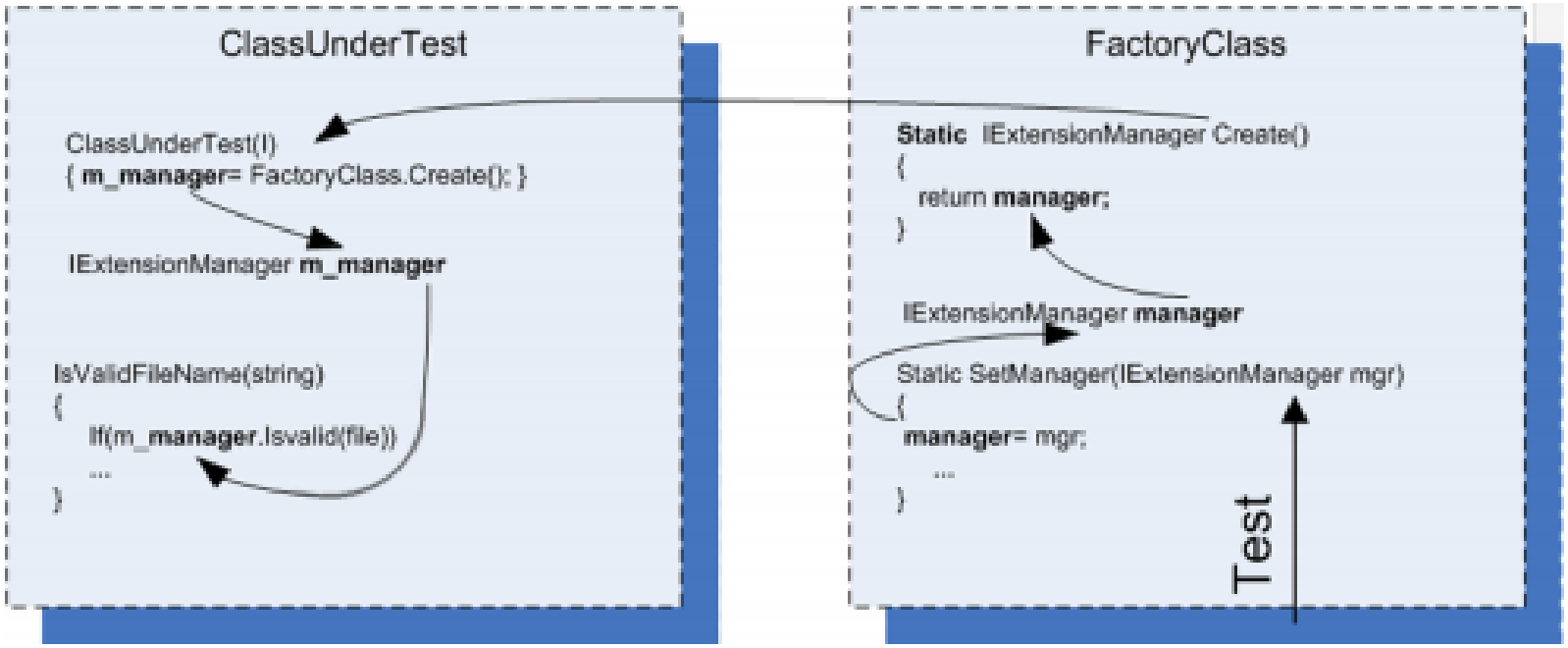
```
[Test]
Public void
IsValidFileName_SupportedExtension_ReturnsTrue()
{
    //set up the stub to use, make sure it returns true
    ...
    //create analyzer and inject stub
    LogAnalyzer log =
        new LogAnalyzer ();
    log.ExtensionManager=someFakeManagerCreatedEarlier;
    //Assert logic assuming extension is supported
    ...
}
```

Injects a stub

- Test setup sets ExtensionManager to stub

Non scholae sed vitae discimus.

Dependency Injection using Factory



- ClassUnderTest remains untouched by test setup
- Test setup manipulates FactoryClass to inject test stub

Dependency Injection using Factory: Example (1)

```
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;
        }
}
```

Uses factory in production code

- LogAnalyzer constructor calls Factory method to supply ExtensionManager

Dependency Injection using Factory: Example (2)

```
[Test]
public void
IsValidFileName_SupportedExtension_ReturnsTrue()
{
          //set up the stub to use, make sure it returns true
          ...
          ExtensionManagerFactory
            .SetManager(myFakeManager);
         //create analyzer and inject stub
         LogAnalyzer log =
             new LogAnalyzer ();
         //Assert logic assuming extension is supported
         ...
}
class ExtensionManagerFactory
{
    private IExtensionManager customManager=null;
    public IExtensionManager Create()
    {
        If(customManager!=null)
          return customManager;
        Return new FileExtensionManager();
    }
    public void SetManager(IExtensionManager mgr)
    {
        customManager = mgr;
    }
}
```

Sets stub into factory class for this test

Defines factory that can use and return custom manager

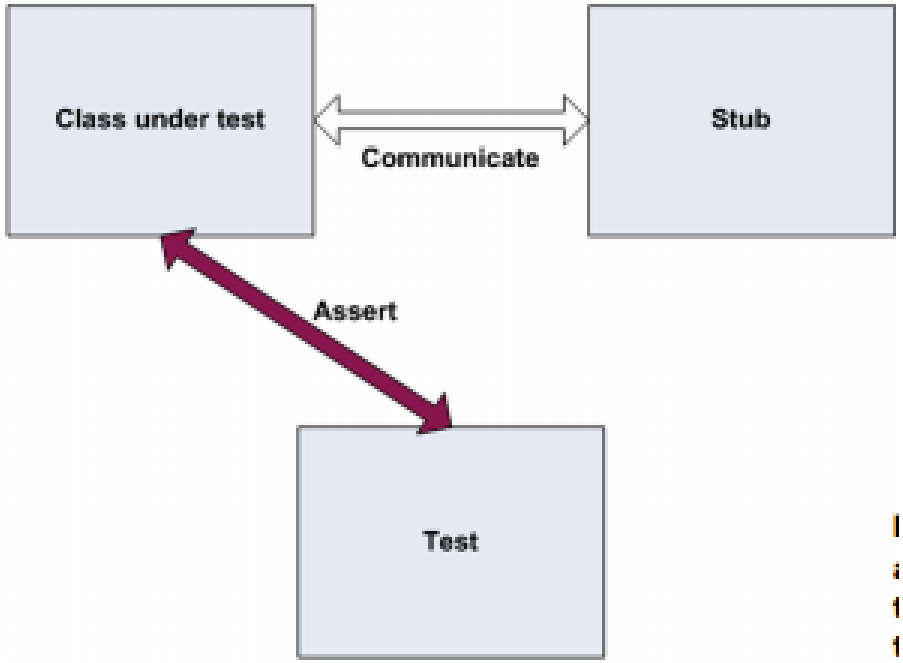
Non scholae sed vitae discimus.

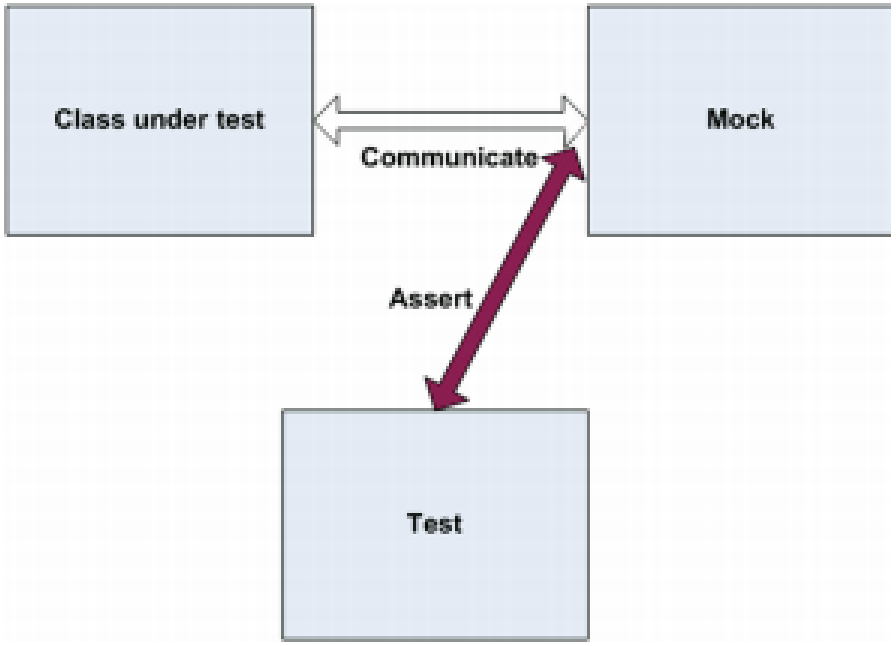
Interaction Testing Using Mock Objects

- Interaction Testing
  - Testing indirect outputs
  - Calls to DoCs
  - Effects not visible in SUT

- Mock objects
  - Doubles for DoCs
  - Perform test verification by comparing actual with expected behavior (i.e. calls from SUT)
  - Usually only one Mock object per test
    - Otherwise test may be doing too much

Stubs vs. Mocks





- Stubs control and supply indirect inputs
- Verification uses interface of class under test (to access results/state)

- Spies/Mocks observe and record indirect outputs
- Verification uses mock interface (to access interaction data)

Non scholae sed vitae discimus.

Test Spy: Example (1)

- Assume Loganalyzer sends message to Webservice
  when file name too short

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

public class FakeWebService:IWebService
{
    public string LastError;

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

- WebService interface
- Test Spy for WebService
- LastError: interface for assert method to access message sent

Non scholae sed vitae discimus.

Test Spy: Example (2)

```
[Test]
public void Analyze_TooShortFileName_CallsWebService()
 {
    FakeWebService mockService = new FakeWebService();
    LogAnalyzer log = new LogAnalyzer(mockService);
    string tooShortFileName="abc.ext";

    log.Analyze(tooShortFileName);

    StringAssert.Contains("Filename too short:abc.ext",
                          mockService.LastError);
}

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);
       }
   }
}
```

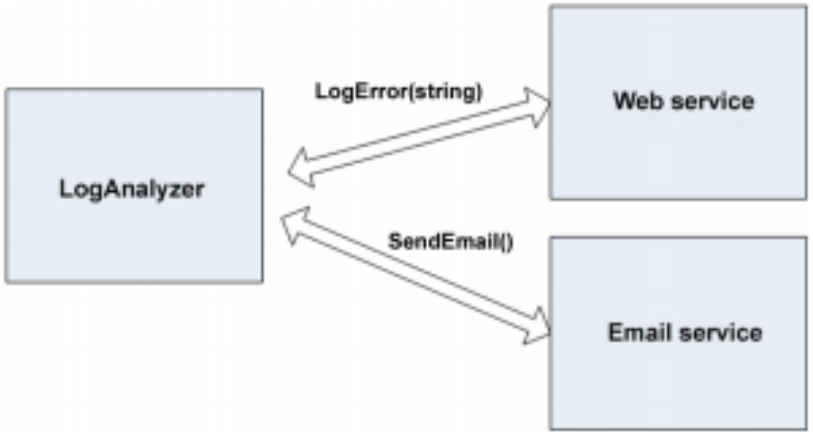
Asserts against a mock object

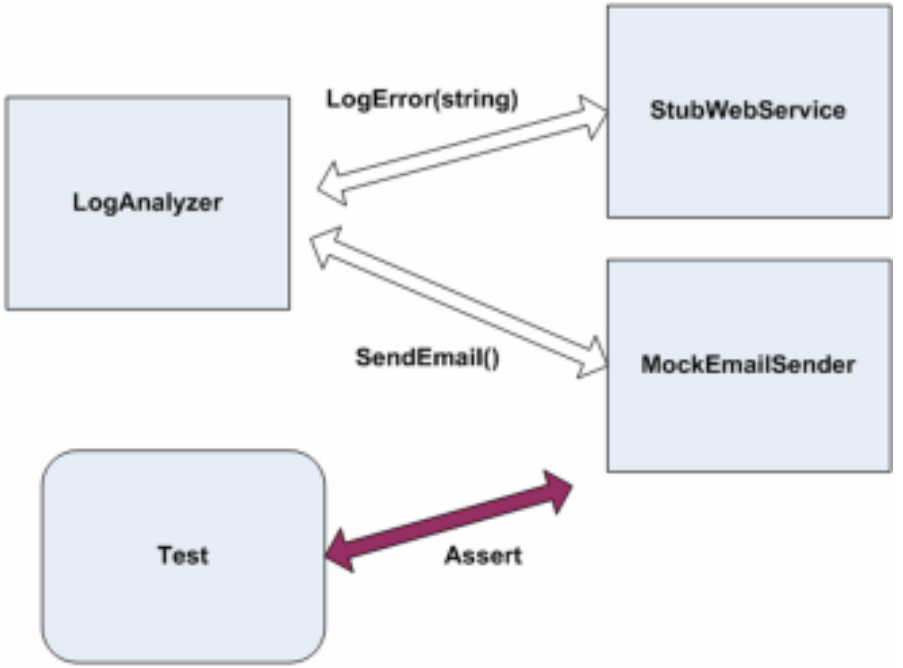
- Verification uses state of mock object,
- **not** state of SUT

Logs error in production code

Non scholae sed vitae discimus.

Using Mock and Stub together





- LogAnalyzer has two dependencies
- When LogError fails, WebService throws an exception,
  and LogAnalyzer sends email msg.

- To test interaction:
- Stub for WebService provides exception
- Mock for EmailService monitors SendEmail interaction

Non scholae sed vitae discimus.

Using Mock and Stub together: Example (1)


```
public interface IEmailService
    {
        void SendEmail(string to, string subject, string body);
    }

public class LogAnalyzer2
    {
       public LogAnalyzer2(IWebService service, IEmailService email)
        {
             Email = email,
             Service = service;
        }
        public IWebService Service
        {
            get ;
            set ;
        }

        public IEmailService Email
        {
            get ;
            set ;
        }

        public void Analyze(string fileName)
        {
            if(fileName.Length<8)
            {
                try
                {
                  Service.LogError("Filename too short:" + fileName);
                }
                catch (Exception e)
                {
                    Email.SendEmail("someone@somewhere.com",
                                    "can't log",e.Message);
                }
            }
        }
    }
```


Non scholae sed vitae discimus.

Using Mock and Stub together: Example (2)

- Note: [TestFixture] is a NUnit specific attribute.
- Better would be [TestClass]
- Do not confuse with standard meaning of test fixture

```
[TestFixture]
 public class LogAnalyzer2Tests
 {
     [Test]
      public void Analyze_WebServiceThrows_SendsEmail()
      {
          FakeWebService stubService = new FakeWebService();
          stubService.ToThrow=  new Exception("fake exception");

          FakeEmailService mockEmail = new FakeEmailService();

          LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);

          string tooShortFileName="abc.ext";
          log.Analyze(tooShortFileName);

          StringAssert.Contains("someone@somewhere.com",mockEmail.To);
          StringAssert.Contains("fake exception",mockEmail.Body);
          StringAssert.Contains("can't log",mockEmail.Subject);
      }
}
```

Using Mock and Stub together: Example (3)

```
public class FakeWebService:IWebService
{
    public Exception ToThrow;
    public void LogError(string message)
    {
        if(ToThrow!=null)
        {
            throw ToThrow;
        }
    }
}

public class FakeEmailService:IEmailService
{
    public string To;
    public string Subject;
    public string Body;

    public void SendEmail(string to,
           string subject,
           string body)
    {
        To = to;
        Subject = subject;
        Body = body;
    }
}
```

Testbed construction

- Objects collaborate with other objects in an application
- Collaboration results in dependencies
- How can we test individual objects/properties in isolation
  - Exactly what is to be tested?
  - What potential problems are the tests to address?
- How can we construct test that get around the dependencies?

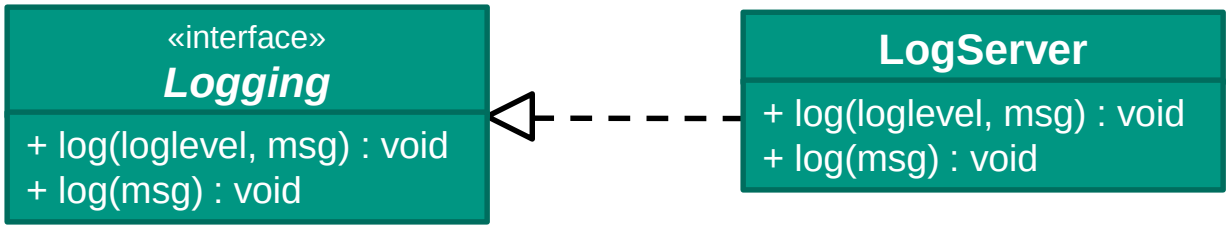
Test Helpers

- Use Test Doubles as approximations when there are dependencies on classes that
  - are not implemented yet
  - Inefficient/expensive to use
  - Difficult to observe
- Doubles as stand-ins for still missing implementations
  - Top-down or Outside-In development
  - Replace stubs and dummies by actual implementations

Types of Test Doubles (Meszaros)

- Test Stub
  - Supplies indirect inputs to SUT
- Test Spy
  - Observes and records calls to Depended-on Components (DoC)
  - Provides data for test assertions after SUT execution
  - May also provide indirect inputs
- Mock Objects
  - Compare during SUT execution actual invocation of DoC with expected invocation
  - May also provide indirect inputs
- Fake objects
  - “knock-offs” of DoC with same functionality but more efficient
  - E.g. hash table instead of relational DB

Example: Double (1)



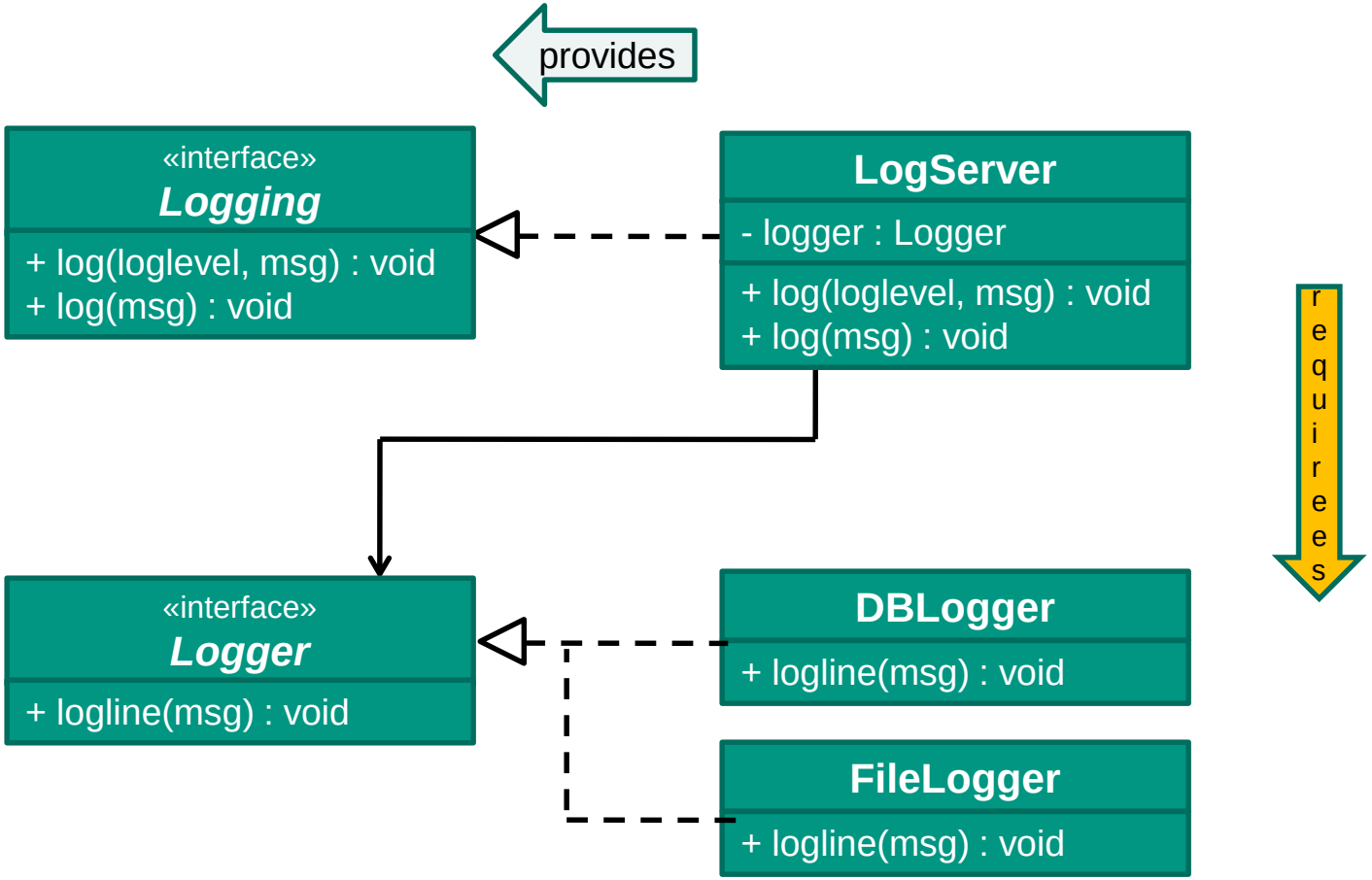
- System-wide log service
- `log(loglevel, msg)` records messages of any priority
- `log(msg)` records messages with standard priority 2

Example: Double (2)

- Problem:
  Where is the log being kept?
  - Initially file system may be sufficient
  - Relational DB may come later
  - Either option not ideal for testing LogServer

- Solution:
  hide log repository behind Logger interface

Example: Double (3)

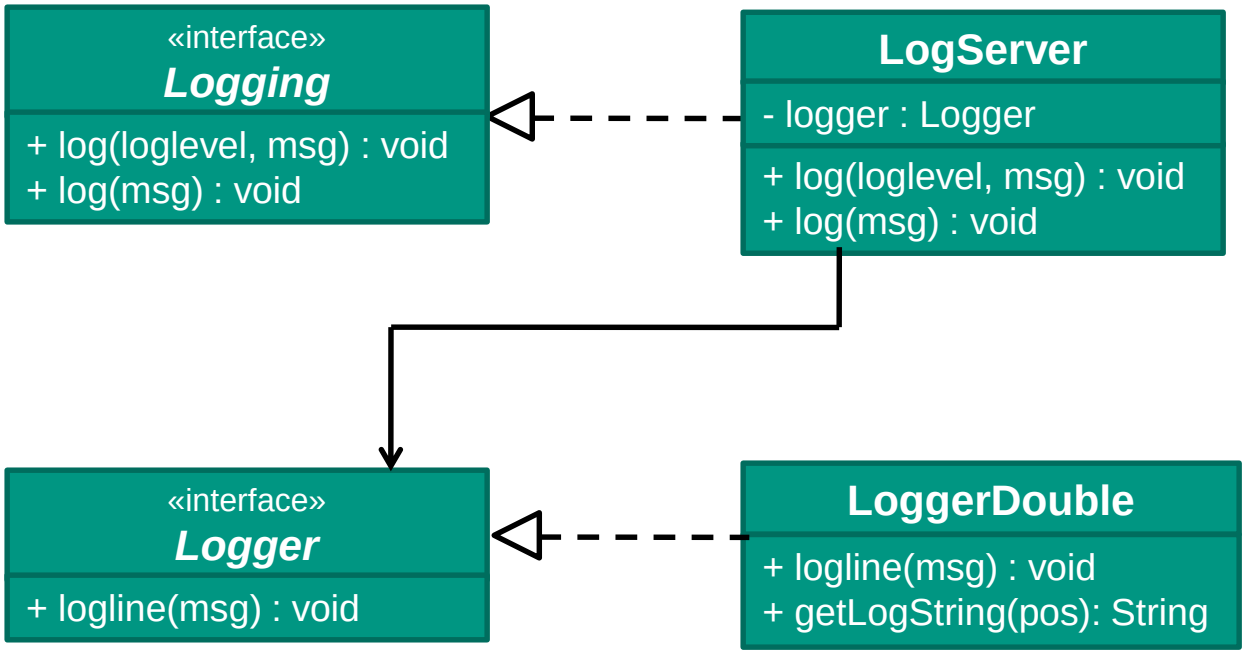


Non scholae sed vitae discimus.

Example: Double (4)

- Most important Logger property is now explicit:
  - Log is recorded line by line
- Target repository hidden behind interface: `DBLogger`, `FileLogger`
- `LogServer` contains instance of Logger.
- How does that help with testing?

Example: Double (5)



Example: Double (6) – Test Spy

- Double
  - Implements Logger interface for `LogServer`
  - Implements Retrieval interface
    - `getLogString` for the test
    - `getLogString(pos)` supplies the `pos`-th message
- Now we can write the test for LogServers:

```
public void testSimpleLogging() {
    LoggerDouble logger = new LoggerDouble();
    Logging logServer = new LogServer(logger);

   logServer.log(0, “First Line“);
   logServer.log(1, "Second Line");
   logServer.log("Third Line");

   assertEquals("(0): First Line",
        logger.getLogString(0));
   assertEquals("(1): Second Line",
        logger.getLogString(1));
   assertEquals("(2): Third Line",
        logger.getLogString(2));
}
```


Non scholae sed vitae discimus.

Example: Double (7)

- LogServer is independent of log repository
- Interface used for testing
- Checking of `LogServer` without exposing implementation

- Question:
  What can still be improved?

- Answer:
  migrate test code to dummy, turns into LoggerMockobject

Example: Test Mock

```
public class LoggerSimulation implements Logger {
    private List expectedLogs = new ArrayList();
    private List actualLogs = new ArrayList();

    public void addExpectedLine(String logString) {
      expectedLogs.add(logString);
    }
    public void logLine(String logLine) {
      actualLogs.add(logLine);
    }
    public void verify() {
      if (actualLogs.size() ! = expectedLogs.size()) {
        Assert.fail("Expected" + expectedLogs.size()
          +‘‘log entries but encountered‘‘+actualLogs.size());
      }
      for (int i = 0; i < expectedLogs.size(); i++){
        String expLine = (String) expectedLogs.get(i);
        String actLine = (String) actualLogs.get(i);
        Assert.assertEquals(expLine, actLine);
      }
    }
}
```

- Defines expected output
- called at beginning of test.

- Checks number of log entries and their contents.
- Called at end of test.

Example: Test Mock (2)

- Test code:

```
LoggerSimulation logger;
@Test
public void testSimpleLogging() {
  logger.addExpectedLine("(0): First Line");
  logger.addExpectedLine("(1): Second Line");
  logger.addExpectedLine("(2): Third Line");
  logServer.log(0, " First Line ");
  logServer.log(1, " Second Line ");
  logServer.log(" Third Line ");
  logger.verify();
}
```

- errors reported upon call of `verify()`
- for complex behaviors cause of error may be difficult to find

Example: Test Mock (3)

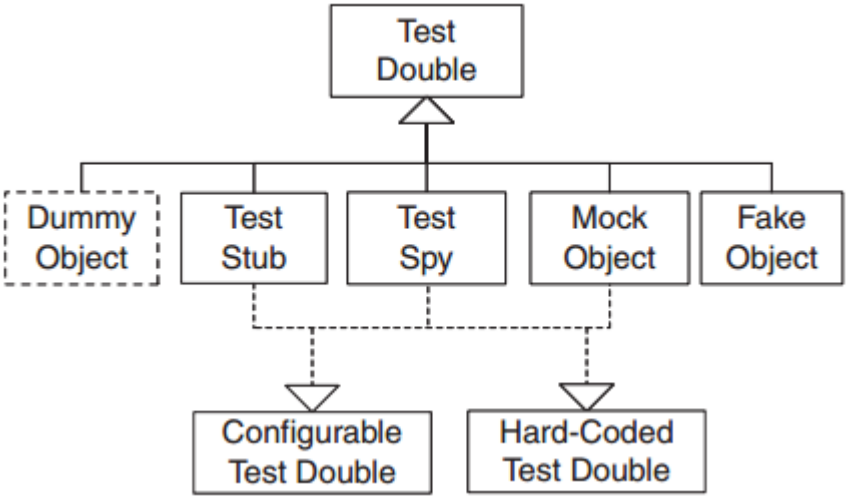
- Changed `LoggerMockobject`:

```
public void addExpectedLine(String logString) {
      expectedLogs.add(logString);
    }

public void logLine(String logLine) {
    Assert.assertNotNull(logLine);
    if (actualLogs.size() >= expectedLogs.size()) {
      Assert.fail("Too many log entries");
    }

    int currentIndex = actualLogs.size();
    String expectedLine =
      (String) expectedLogs.get(currentIndex);
    Assert.assertEquals(expectedLine, logLine);
    actualLogs.add(logLine);
}

public void verify() {
    if (actualLogs.size() < expectedLogs.size()) {
      Assert.fail("Expected" + expectedLogs.size()
       + "log entries but encountered" + actualLogs.size());
    }
}
```

Example: Mock object (4)

- The Mockobject
  - provides for each input a `setExpected` method; `addExpected` to accommodate multiple calls
- Contains the test code in `verify`
- Checks parameters and the order of method calls
- **Note**:
  Not the mock object is being tested
  but the use of it by the SUT

Test Double Flavors



- Dummy Object
  - Type-correct value needed by SUT but never used
- Fake Object
  - Functionally equivalent replacement of DoC
  - Cheaper, faster, simpler; local
  - Often not scalable
  - E.g. hashtable faking relational DB

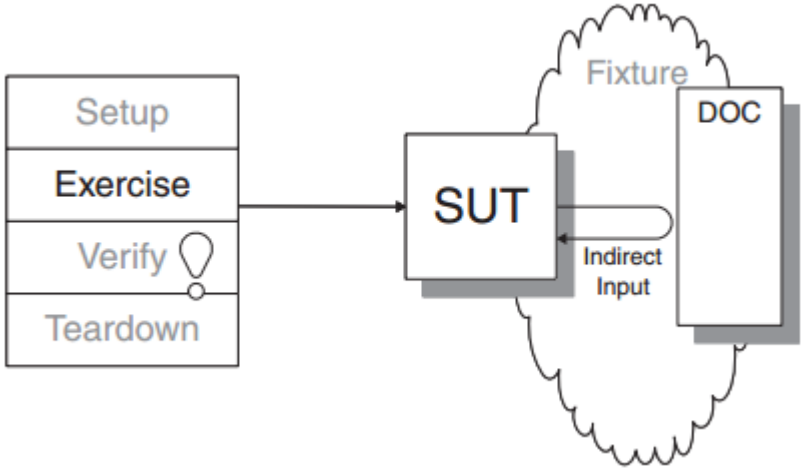
- Test Stub
  - Control point for indirect inputs
- Test Spy
  - Observation point for indirect outputs
- Mock Object
  - Observes and verifies indirect outputs
- Configurable (typical case) or hard-coded

Non scholae sed vitae discimus.

Test Double Flavors (Meszaros) (again)

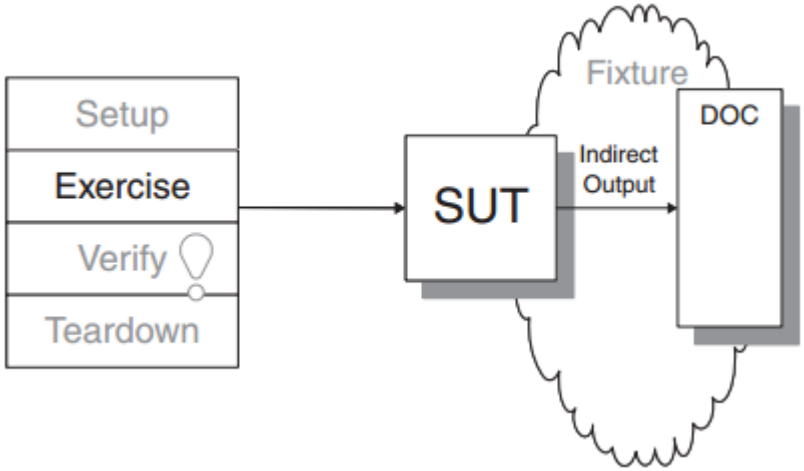
- Test Stub
  - Supplies indirect inputs to SUT
- Test Spy
  - Observes and records calls to Depended-on Components (DoC)
  - Provides data for test assertions after SUT execution
  - May also provide indirect inputs
- Mock Objects
  - Compare during SUT execution actual invocation of DoC with expected invocation
  - May also provide indirect inputs
- Fake objects
  - “knock-offs” of DoC with same functionality but more efficient
  - E.g. hash table instead of relational DB

Indirect Inputs



- Indirect inputs
  - Return values
  - Updated parameters/state
  - exceptions
- Control point
  - Point of injecting an indirect input into execution of SUT
  - Allows testing of execution paths requiring specific conditions
    - Including exception handling paths

Indirect Output

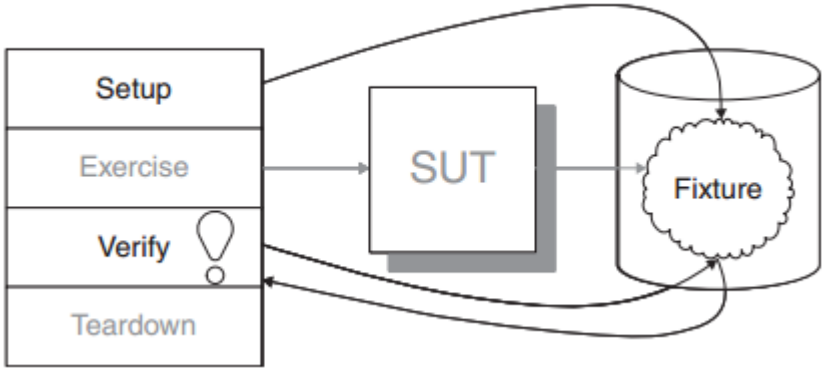


- Indirect Outputs
  - Effect of calling Doc may not be visible to SUT
  - Output may be sent to other components
  - E.g.
    - writing to log;
    - sending email,
    - storing in DB

- Observation Point
  - Means of seeing/accessing indirect outputs
  - Used in indirect output test conditions (to determine whether test passed or failed)

Non scholae sed vitae discimus.

Controlling indirect inputs: Back Door Setup

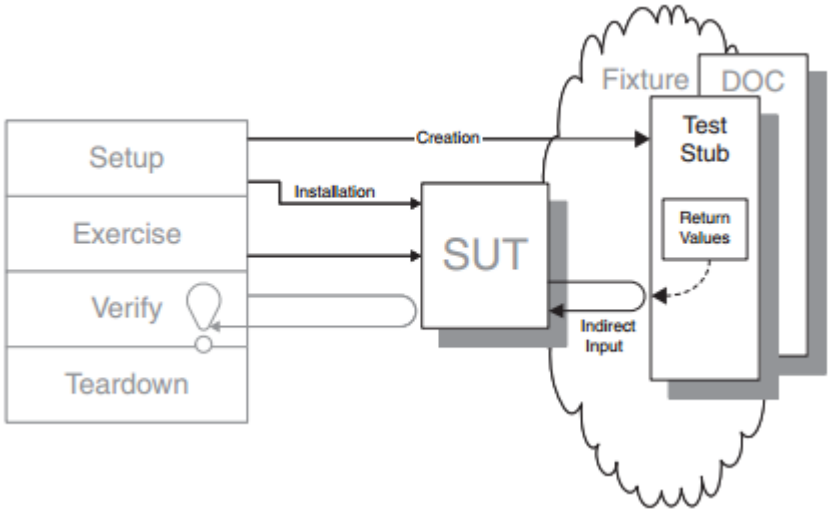


- Back Door Setup
  - Configure DoC in test set up to produce specific response
  - E.g. load data into DB so data look up will produce desired number of values (no item found, one item found, >1 items found)
- DoC itself serves as control point in Back Door Setup

- Back Door Setup
  - Not practical when manipulation of DoC
    - Is expensive
    - Has undesired side effects
  - Not possible when DoC
    - Cannot be manipulated for desired effect
    - Does not exist yet

Non scholae sed vitae discimus.

Test Stub as Control Point for Indirect Inputs

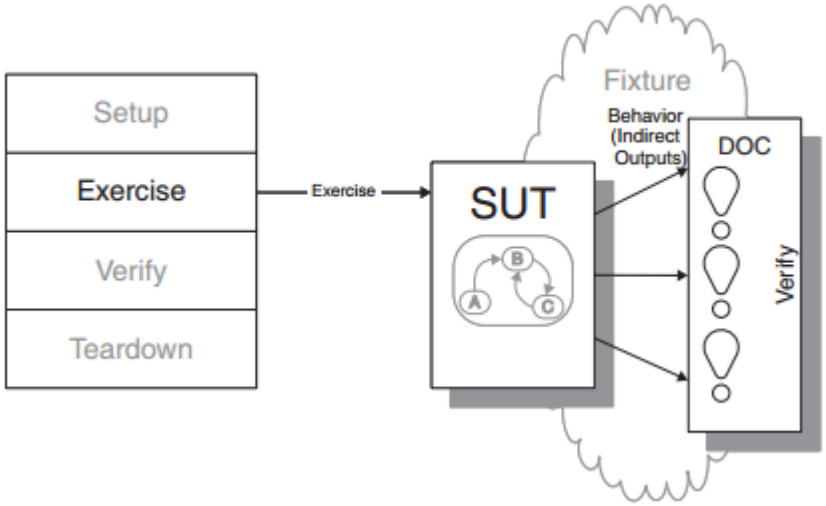


- Test Stub
  - DoC double
  - Control point of indirect inputs

- Setup phase
  - Create Test Stub object
  - Configure to return desired values to SUT
  - Install Test Stub object into SUT
- Exercise phase
  - Test Stub handles all calls to DoC

Non scholae sed vitae discimus.

Behavior (Indirect Output) Verification

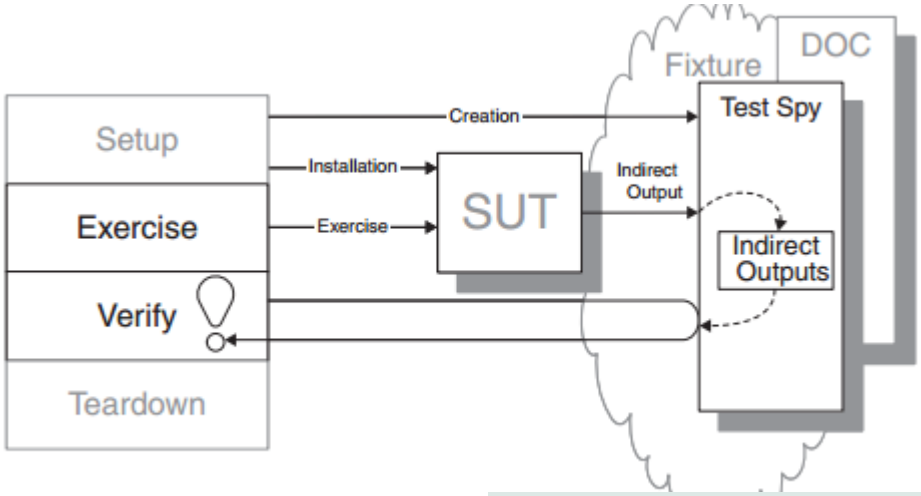


- To test indirect outputs
  - Need to observe calls of SUT to API of DoC
  - May also need to supply indirect inputs

- Back Door Verification
  - use DoC as observation point when DoC allows suitable queries, e.g.
    - Check file written with expected contents
    - Query DB for expected contents
- Not possible or practical when
  - Suitable DoC queries not possible, too expensive, or unacceptable side effects
  - DoC not available for use

Non scholae sed vitae discimus.

Test Spy for Procedural Behavior Verification

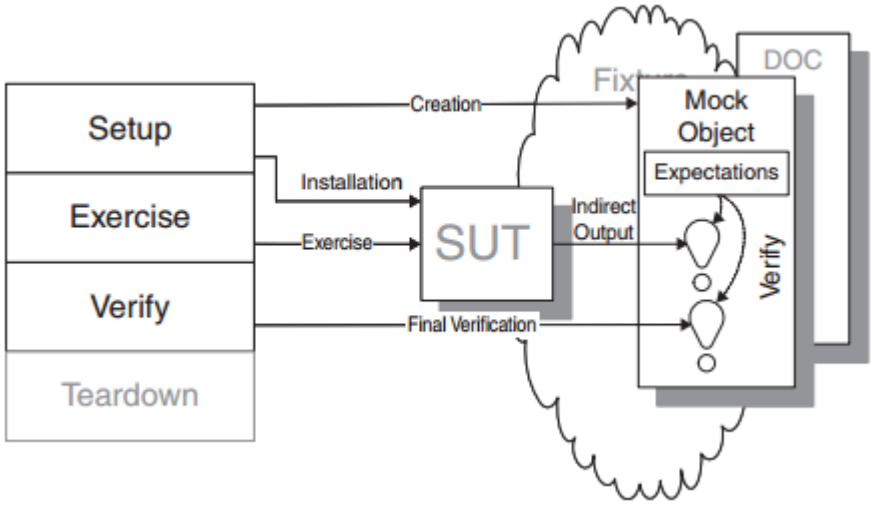


- Test Spy
  - Records indirect outputs from SUT
  - Implements DoC interface
  - Implements retrieval interface for use by test method during Verify phase

- Setup phase
  - Create Test Spy object
  - Install Test Spy object in SUT (instead of DoC)
- Exercise phase
  - Record indirect outputs from SUT in Test Spy object
- Verify phase
  - Test method accesses indirect outputs through retrieval interface

Non scholae sed vitae discimus.

Mock Objects with Expected Behavior for Indirect Output Verification



- Mock Object
  - Configured with expected use by SUT
  - Implements DoC interface
  - Compares actual use with expected use during SUT execution
  - Implements final verification method
    - E.g. to check that all expected calls were received

- Setup
  - Create Mock object
  - Configured with expected use by SUT
  - Install in SUT
- Exercise
  - Mock object verifies each call from SUT
  - Fails test when first call fails to verify
- Verify
  - Mock object performs final verification

Non scholae sed vitae discimus.

Providing the Test Double

- Hand-built test doubles
  - Hardcoded, or
  - Configurable
  - Simplify using *pseudo objects*
- Generated test doubles
  - Always configurable
  - Dynamically generated
    - Jmock toolkit
  - Statically generated
    - EasyMock toolkit
    - Compiled just like hand-built doubles

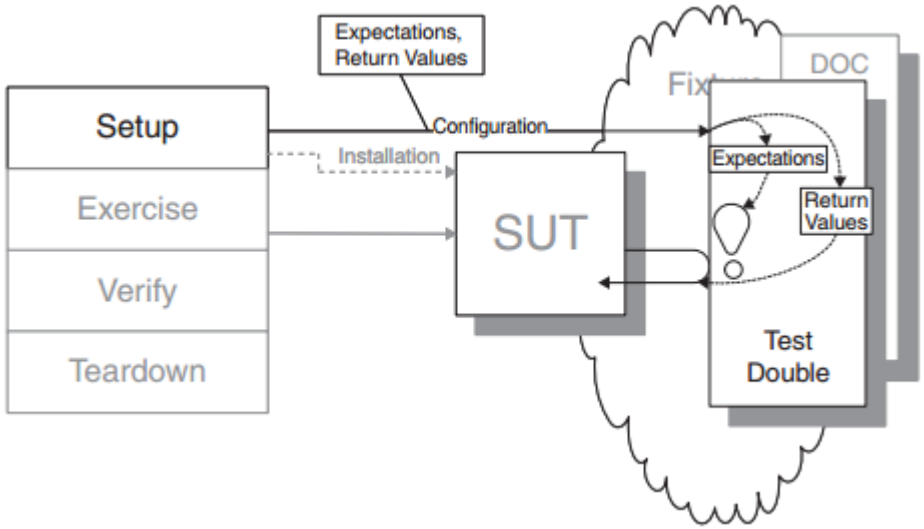
- Hand-built inner test stub →
  - Hardcoded
  - Defined within test method

```
public void testDisplayCurrentTime_AtMidnight_PS()
      throws Exception {
   // Fixture setup
   //    Define and instantiate Test Stub
   TimeProvider testStub = new PseudoTimeProvider()
   { // Anonymous inner stub
      public Calendar getTime(String timeZone) {
         Calendar myTime = new GregorianCalendar();
         myTime.set(Calendar.MINUTE, 0);
         myTime.set(Calendar.HOUR_OF_DAY, 0);
         return myTime;
      }
   };
   //   Instantiate SUT
   TimeDisplay sut = new TimeDisplay();
   //   Inject Test Stub into SUT
   sut.setTimeProvider(testStub);
   // Exercise SUT
   String result = sut.getCurrentTimeAsHtmlFragment();
   // Verify direct output
   String expectedTimeString =
           "<span class=\"tinyBoldText\">Midnight</span>";
   assertEquals("Midnight", expectedTimeString, result);
}
```


Non scholae sed vitae discimus.

Configuring Test Doubles

- Test stubs
  - Need return values
- Mock objects
  - Need expected interactions



- Hard-coded doubles
  - Values/expected interactions defined at design time
- Configurable doubles
  - Values/expected interactions defined at runtime by test method

- Configurable doubles
  - Provide configuration interface
    - E.g. attribute setters
  - Reusable across tests
  - Make tests more understandable
    - Configuration data visible in test method
  - Configuration takes place in setup phase

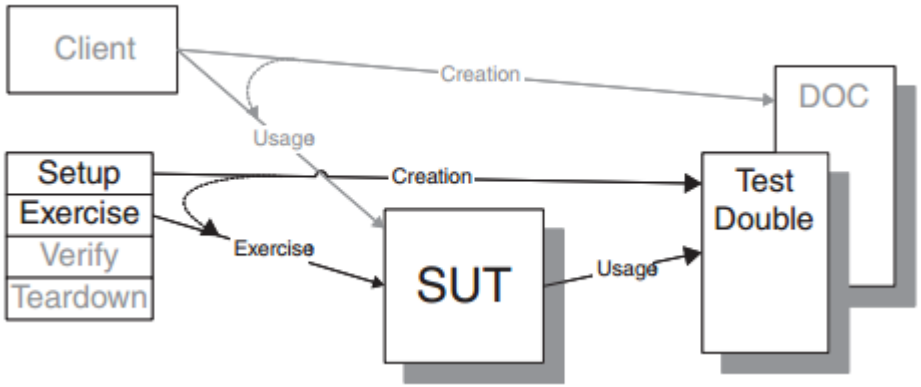
Non scholae sed vitae discimus.

Installing the Test Double

- Installing the test double
  - Connecting the SUT to the test double (instead of the production DoC)

- Installation options
  - Dependency Injection
    - Client passes DoC to dependent object
  - Dependency Lookup
    - Construction/Selection of DoC delegated to another object
  - Test hook
    - Conditional call to components within SUT
  - Inversion of Control frameworks
    - Language specific
    - Automate substitution of dependencies

Dependency Injection

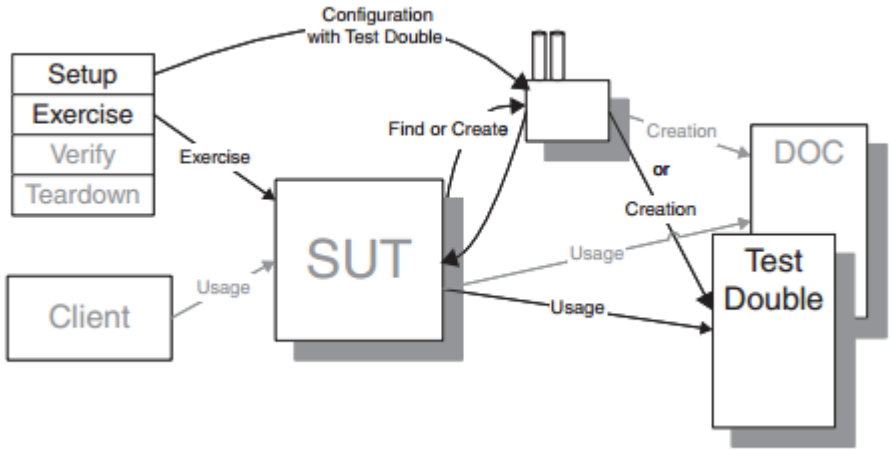


- SUT not "aware" of specific DoC
  - Only of interface
  - Good design practice
  - Parameterizes SUT on DoC
  - Makes SUT (re)usable in other contexts
  - Frequently used in TDD

- Dependency Injection Flavors
- Constructor injection
  - DoC passed to constructor method
  - Stored in private attribute
- Setter injection
  - DoC passed to setter method
  - Double object may replace real DoC
- Parameter injection
  - DoC passed to SUT method

Non scholae sed vitae discimus.

Dependency Lookup

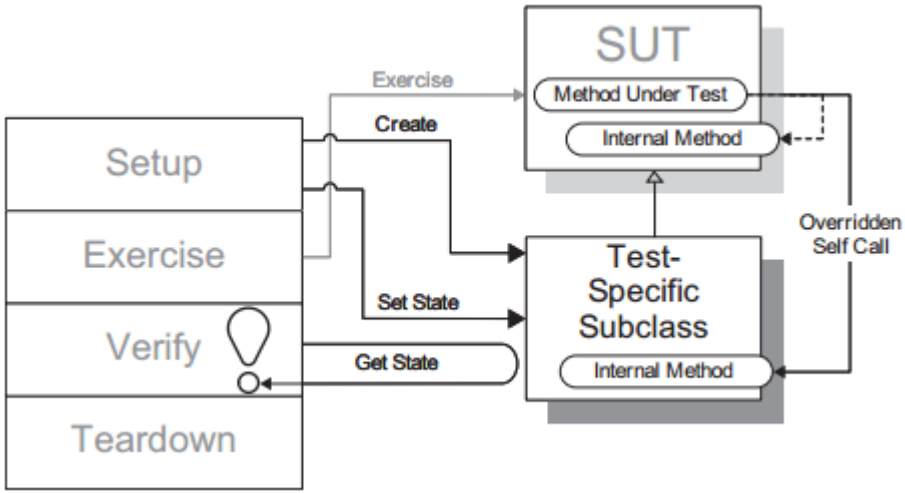


- Dependency Lookup
  - SUT acquires DoC via Object Factory or Service Locator (Registry)
  - Configuration of Factory or Registry
    - During Setup phase
    - controls which DoC the SUT uses

- Object Factory (GoF)
  - SUT acquires DoC by calling Factory method
  - Factory method creates DoC
- Service Locator (Registry)
  - SUT asks Service Locator for DoC
  - Service Locator provides already created DoC

Non scholae sed vitae discimus.

Retrofitting Testability: Test-specific Subclass of SUT



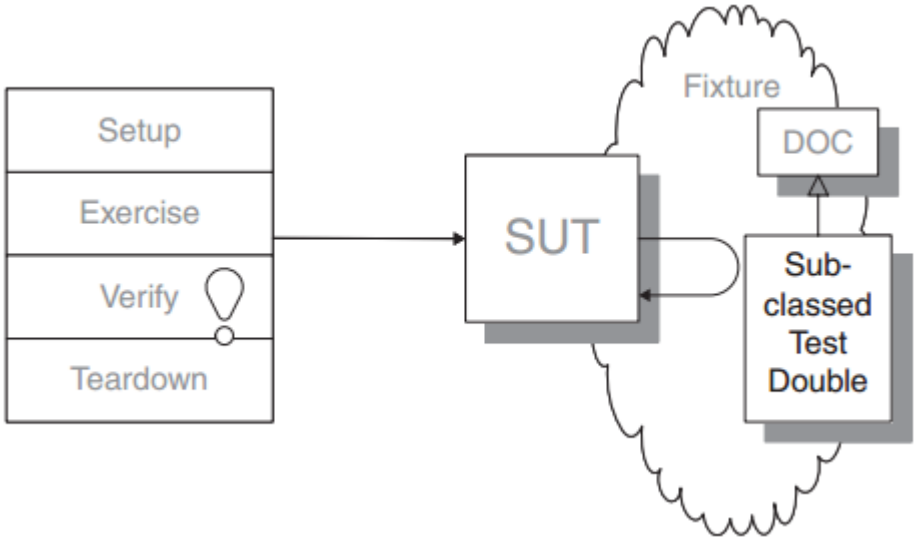
- What if test needs access to private state of SUT?

- SUT
  - Encapsulate DoC access in internal method
    - Clean Code:
      intention revealing abstractions;
      single-purpose functions/methods
- SUT subclass
  - Add methods to access private state
  - Inject dependency on test double
    by overriding DoC access methods

- Drawback
  - Subclass may accidently modify behavior to be tested

Non scholae sed vitae discimus.

Retrofitting Testability: Test-specific subclass of DoC



- Create Test Double as subclass of Doc
  - Override some or all methods used by SUT
  - Provide indirect inputs
  - Monitor indirect outputs

- Safer than subclassing SUT
  - Avoids modifying SUT behavior being tested

Non scholae sed vitae discimus.

Retrofitting Testability: Test hooks

- Test hooks
  - Conditional use of Test Doubles
  - Not mentioned in polite Agile society
  - Intermingles test code with production code
- Legitimate way to make legacy code testable
  - Transition strategy
  - Enables tests without large-scale refactoring
  - After refactoring for testability test hooks can be removed