Exercise Instructions

Testing for Developers Course

Presented by Gerard Meszaros

xUnitTraining@gerardmeszaros.com

Contents:

JUnit 4 & Java/Eclipse-Specific Instructions

Note: Please read before starting first exercise. Note which exercises have Java-specific instructions. Read the Java-specific exercise instructions before starting the corresponding exercise.

JUnit 4 Notes:

Annotations used in JUnit 4 tests:

Identifying Tests:

@Test – makes a method into a test method.

@Test(expected=Exception.class) – to make it an ExpectedException test without using a try/catch inside the test method body.

@Parameters – used to specify the method that provides the dataset for a Parameterized Test. JUnit will create an instance of the TestCase Object for each entry in this dataset and will then run all methods marked @Test on each of those TestCase Objects.

Managing Test Fixture:

@Before – runs the method before **each** @Test method. Used for setting up the test fixture. (Fresh Fixture)

@After – runs the method after **each** @Test method whether it passes or fails. Useful for tearing down the test fixture. (Fresh Fixture)

@BeforeClass – runs the method **once** before **all** the @Test methods on the test class. Used for SuiteFixture SetUp of a Shared Fixture.

@AfterClass – runs the method **once** after **all** the @Test methods on the test class have been run. Used for SuiteFixture TearDown of a Shared Fixture.

Test Suites:

@RunWith(Suite.class) – Specifies the test runner to use with the test class. Can be Suite or something else.

@Suite.SuiteClasses({… }) – Used to specify the classes that belong to the test suite. Use with @RunWith(Suite.class).

See Eclipse-specific instructions on the following pages.

Eclipse Notes:

To load each exercise in Eclipse:

1. Select File -> Import from the toolbar.
2. Select "Existing Project into Workspace" in the first import wizard
3. Click "Browse" and locate the particular exercise directory. Click "OK". Each exercise directory has an Eclipse project file (.project) that informs Eclipse of all libraries and source files to load.

To run all the tests in a JUnit test class:

1. Ensure you are in the "Java" perspective. If not, select Window->Open Perspective->Java
2. Select the particular JUnit Test class you want to run in the Package Explorer.
3. Click the icon with a running man. A drop down menu appears and select Run as -> JUnit test.
   Or:
4. Right-click on the class name in either the Project Explorer or the inside the code window.

To run a single test in a JUnit test class (one @Test method):

1. If you have run the whole class, right-click on one of the individual test results in the JUnit window and select Run.
2. You can also right-click on an @Test method name in the code window and select Run as -> JUnit test.

To view the console output (Exercise 6):

If the console window is not already visible on a tab in one of the panes within the Java perspective,

1. Select Window->Show View->Console

To get a list of methods available on a hidden (e.g. the SUT) class:

1. Create a variable with the type of interest.
2. Use the variable in a statement followed by "." (dot).
3. Press <ctrl><spacebar> to see a list of methods and attributes visible in the current scope. (Be patient; it may take a few seconds to initialize the first time.)

See language-specific instructions for some exercises on the following pages.

Exercise-Specific Language-Specific Instructions

This section contains language-specific instructions for some of the exercises to augment the generic exercise instructions. Please refer to the appropriate exercise instructions and come back here when it says "Please refer to the Language-specific instructions".

Exercise 6 Eclipse/JUnit4-specific Instructions:

Running a Test Suite Several Times in a Row:

To simulate persistence between test runs we use a JUnit4 test suite to run the test classes several times in the same test run because the JUnit plug-in for Eclipse automatically reloads the SUT classes before every run.

- select RunTests_Twice.java in the Package Explorer.
- select Run As -> JUnit Test from the Run menu or from the context menu.

*Note that there is a **bug** in JUnit/Eclipse that causes the results to display strangely in the JUnit4 GUI, so look in the console to see what's really happening each time the tests are run.*

Exercise 8 Eclipse-Specific Instructions:

Installing MockMaker within Eclipse:

1. Copy the zip file from the common directory or download the plug-in from http://www.mockmaker.org/.
2. If Eclipse is open, close it.
3. Unzip the contents into the ECLIPSE_HOME/plugins directory, where ECLIPSE_HOME is the directory containing eclipse executable.
4. Start Eclipse.

Using MockMaker within Eclipse:

1. Ensure MockMaker plugin is installed
2. Open the Java Browsing Perspective
3. Select the package `com.xUnitPatterns.ex8`.
4. Select the interface `AuditLog`. Right-click and select `MockMaker->Select Package`, pointing to the `com. xUnitPatterns.ex8.student.test package`. This will create a class called `MockAuditLog.java`.
5. Repeat step 5, replacing the `AuditLogTestSpy` with your new `MockAuditLog.`

.Net Language-Specific Instructions

Note: Please read first section (up to Exercise-Specific Notes) and scan the last section before starting first exercise. Then use as reference thereafter while doing the exercises.

General Notes:

Some .Net programming idioms are not followed to keep the .Net version of the course in sync with other languages. For example, some method names start with lower case letters, some test methods start with the word "test" even though the [Test] attribute is what identifies the method as a test to NUnit, and some properties are be directly accessible as Xxx = abc even though the notes refer to setXxx( ) or getXxx( ) methods.

Visual Studio Add-ins and Add-ons

Resharper

Resharper is a Visual Studio plug-in that adds many capabilities to the basic set provided by VS. Of most relevance to this course is the ability to run NUnits from the context menu on Projects or Classes, and to do extensive, safe, automated refactoring of your test and product code.

One drawback of Resharper from the perspective of this course is how it handles console output from tests. In a nutshell, terribly. This makes it misleading when running the tests in Exercise 6 where console output is used to help us visualize the execution of various parts of each test.

A 30-day trial version can be installed via Tools->Extensions and Updates.

TestDriven.Net

TestDriven.Net is a Visual Studio plug-in that lets you run tests by right-clicking on a Project or a Class in the Solution Explorer or a class name or method name in the source code itself. Also can be used to launch NUnit GUI (and other installed test frameworks) by right-clicking on the project and selecting Test With … NUnit.
A trial/personal version can be downloaded from http://TestDriven.net and activated via Tools->Add-in Manager…

NUnit GUI:

You can use the NUnit GUI to run tests and see the green bar/red bar from outside VS. It is also required for several of the steps in Exercise 6 because it allows control over whether the assembly is reloaded each time the tests are run.
Most of the exercises depend only on NUnit 2.2.8; a few depend on newer versions for specific capabilities.

NUnit Test Adapter

This Visual Studio extension by Charlie Poole (head maintainer of NUnit) lets you run NUnit tests from within VS. To use it, install it via Tools->Extensions and Updates and then open the Test Explorer window via Test->Windows->Test Explorer. It is most useful if you group the tests by project. This allows you to view or run all the tests in one project at the same time. Unlike the ReSharper plugin, it does provide a useful console output for the whole test run (essential in Exercise 6.)

NUnit Notes:

NUnit 2.x Attributes:

In C#, attributes are enclosed in square brackets (e.g. "[" and "]")

In VB.Net, attributes are enclosed in "pointy" brackets (e.g. "<" and ">")

Identifying Tests:

[Test] – Marks a method as being a test.
[TestFixture] – Marks a class as containing tests (optional).
[ExpectedException(typeof(Exception), "expected Msg")] – Marks a test method as expecting an exception as the correct result

Managing Test Fixtures:

[SetUp] – Marks a method as contain code to be run before each test method.
[TearDown] – Marks a method as contain code to be run after each test method.
[TestFixtureSetUp] – Marks a method as contain code to be run before first test method.
[TestFixtureTearDown] – Marks a method as contain code to be run after last test method.

Test Suites:

NUnit automatically creates a test suite for each namespace and assembly so we don't have to create them manually. When run from ReSharper or the NUnit GUI, this structure becomes very visible.

Visual Studio.Net Notes:

To load each exercise in Visual Studio.Net:

1. Navigate to the exercise folder (ie. Exercise-01).
2. Double-click on the *.sln* file (ie. Exercise-01.sln). This will launch Visual Studio.Net and load the project.

Or, you can launch AllCSharpExercises.sln which loads all the exercises and a project containing the CourseMetaTests. When run, these should all pass.

To run an NUnit test:

Using Resharper

These instructions assume that the Resharper Ultimate Visual Studio extension is installed on your workstation.

1. In the solution explorer right-click on the Project or Test Class you wish to run. You will see an option to **Run Unit Tests**… The output will be shown in the Unit Test Sessions window.
2. You can debug your tests by choosing the **Debug Unit Tests** option.

Using TestDriven.net

These instructions assume that the TestDriven Development.Net visual studio.net add-in is installed on your workstation.

1. In the solution explorer right-click on the test you wish to run. You will see an option to **Run-Test(s)**…
2. You can debug your tests by choosing the **Test With… Debugger** option

Using the NUnit GUI Directly

You can also launch the NUnit GUI from the Start Menu and then open the DLL for the project (called ExerciseXx. located in the bin/debug folder. NUnit GUI should automatically reload the DLL / assembly whenever VS rebuilds it (unless you tell it otherwise via options/preferences)

NUnit 2.2

Open the DLL containing the tests (usually called Exercise-XX-Test.dll) using Open…

NUnit 2.4

Open the DLL containing the tests (usually called Exercise-XX-Test.dll) using Open Project

Extracting Methods

Resharper support the Refactor->Extract-Method refactoring.(<ctrl><r>, <ctrl><m>. If you select one or more entire states and invoke Extract Method, it will try to turn all that code into a single method call. It will determine what it needs to pass in as arguments and what it needs to return (only works if one variable assigned in the statements is used later.) However, it will not extract all instances of that code automatically. To do this, you need to use:

Finding and Replacing Code Using Patterns

Select all the code you want to find (and possibly replace) and use Resharper->Find->Search with pattern. It will present you with it's best guess at the pattern you are looking for (you can modify it if you want). And you can also click on the "Replace" button in the top right corner of the dialo to change it to Replace mode. When you click on the Find or Replace button at the bottom of the dialog, it will give you a list of all the places in the code it found the pattern. If you double-click on one, it will position to code window to the matching chunk of code and you can unselect any chunks you don't want changed. Then hit the "Replace" button (yes, this is the 3rd "Replace" button in a row!) and all the changes will be made.
This isn't quite as good as Eclipse's "Replace x additional occurrences of statements with method" check box; hopefully a future version of Resharper will do this automatically for us.

Exercise-Specific Notes

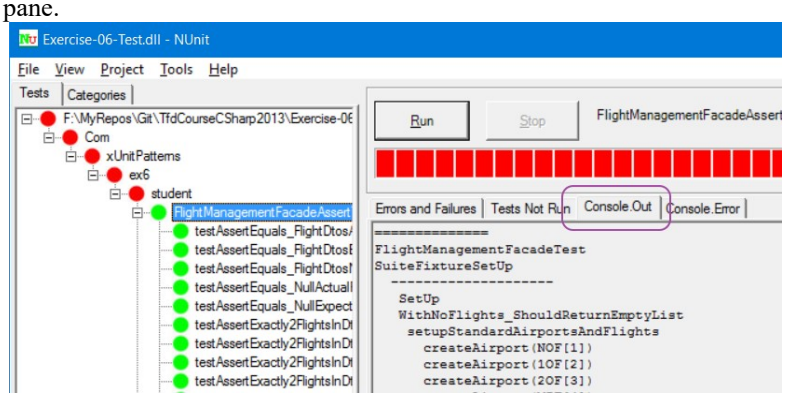
This section contains language specific instructions for some of the exercises to augment the generic exercise instructions. Please refer to the appropriate exercise instructions and come back here when it says "Please refer to the Language-specific instructions".

Exercise 6 Notes:

All Steps: Viewing the console output from the tests:

To provide insight into what goes on under the covers, the tests for Exercise 6 are instrumented with Console.WriteLine statements in each method in the test class plus key methods in the system-under-test (SUT). The best way to see this output is to:

- Run the tests using the NUnit GUI and **click** on the **Console Out** or **Test Output** tab in the right pane.



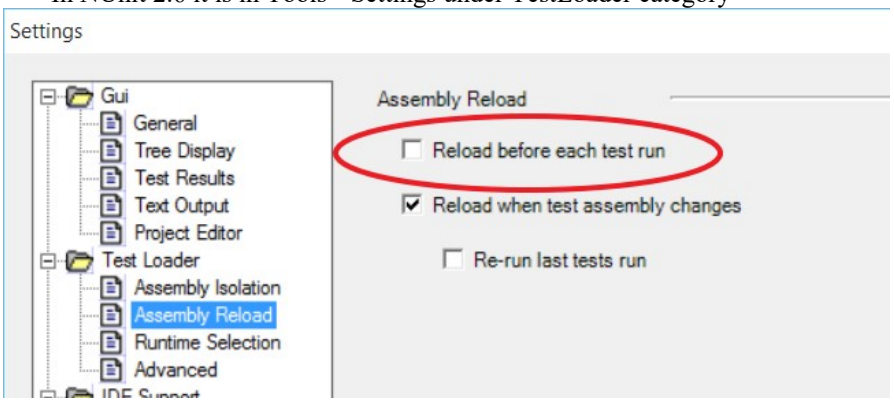
- Another option is to use the NUnit Test Adapter and the Test Explorer window.

Note that Unit Test Sessions window of ReSharper does show the console output for one test at a time but there is no easy way to see the output for the entire test suite in one place.

Step 2: Running the tests more than once in a row:

To do this, we need to use the standalone NUnit test runner:

- ❑ Start the standalone NUnit test runner
  - from StartMenu->Program->NUnit->NUnit GUI then open the DLL using File->Open… and select Exercise-06-Test.dll in the bin/debug directory.
  - Using TestDriven.Net by right-clicking on the project and selecting Test With-> NUnit 2x
  - Note: If you get an error when opening the DLL, make sure you are using the same .Net Framework version in your test exercise project as the NUnit GUI is using (probably V2.0)
- ❑ Run the tests to verify you get the same results as when running from inside Visual Studio. (Optional)
- ❑ In the standalone NUnit TestRunner GUI, uncheck the "Reload before every run" checkbox.
  - In NUnit 2.0 it is in Tools->Options->Tests->Reload before each test run
  - In NUnit 2.4 it is in Tools->Options->Test Loader->Assembly Reload
  - In NUnit 2.6 it is in Tools->Settings under TestLoader category



- ❑ Run the tests in the `FlightManagementFacadeTest` class. What Happens? Why?
  -
  -
  - (Hint: look at the output in the Console.Out tab of the NUnit GUI.)

Note: If you get an error when opening the DLL, make sure you are using the same (or lower) .Net Framework version in your test exercise project as the NUnit GUI is using (e.g. V2.0)

Exercise 8 Notes:

Installing Easymock.Net for use with Visual Studio.Net:

1. If the DLL files for Easymock.net are not in the bin folder at the root of the course project folder then download the current build zip from the following site: http://sourceforge.net/projects/easymocknet/ and unzip this file to a folder on the workstation.
2. To add the Easymock assembly to your project right-click on your reference tab and choose Add Reference…
3. In the Browse tab navigate to the folder where you performed the unzip. Locate the .dll files and press select. It should now appear in the Selected Components table below. Press OK.
4. You should now able to use Easymock.net.

Please note: This step has been done and the Easymock.net dll files are located in the bin folder at the course project root folder.

Other Popular .Net Mocking & Stubbing Frameworks

NMock is another dynamic mocking framework. More information can be reached at http://www.nmock.org. RhinoMock is also popular. See http:// http://ayende.com/projects/rhino-mocks.aspx. MbUnit has some useful features not found in most other members of the xUnit family. See http://www.mbunit.com for details.

Exercise-Specific Result Explanations

The following describe some language-specific nuances of the provided results and what you may have observed during the exercises.

Exercise 6 Results

Two different result solutions are provided. One illustrates the overall result while the other shows the results using the SuiteFixture Setup approach using the [FixtureSetup] and [FixtureTeardown] attributes.

Exercise 1 Instructions

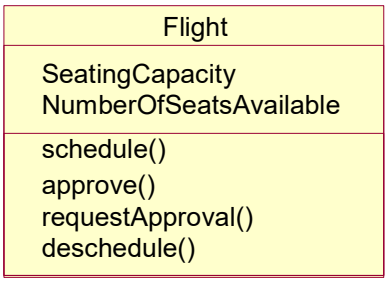
Note: Please read right to the end before starting!

Purpose

To become familiar with creating xUnit tests in the available development environment

System Description

The airline management system we are building contains the following class:



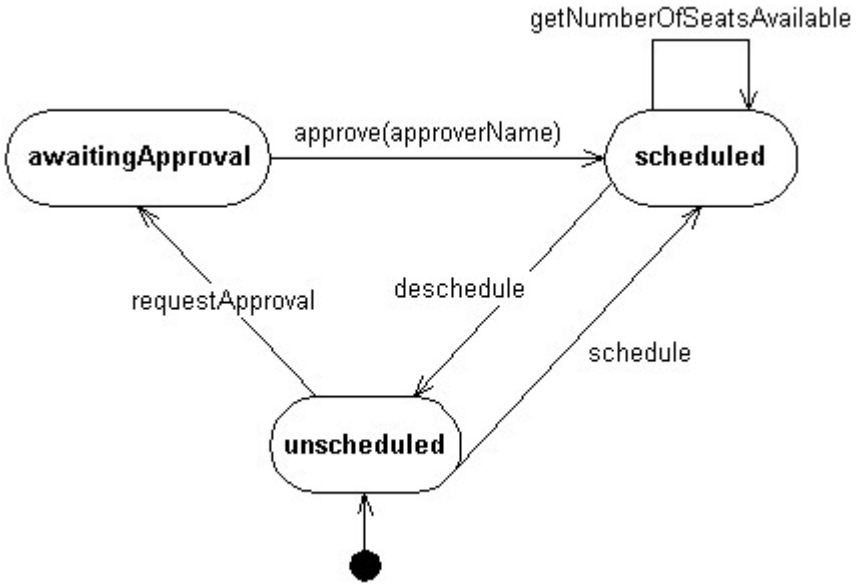
The items in the middle compartment are properties

Java, C++: These are accessed using setter and getter methods

C#, VB.Net: These are accessed as properties (fields and assignment.)

The items in the bottom compartment are operations.

Each flight has the life-cycle described in the following state diagram:



UML Legend:

- Ovals represent states.
- Arrows represent transitions between states. The name of the method or function that causes the transition is shown on the arrow. If the function takes an argument, it is shown in parenthesis as in "approve(approverName)".
- The black spot at the bottom of the diagram is the starting state. On construction, the object is immediately placed into the state at the end of the arrow ("*unscheduled*").

The following describes additional requirement constraints:

1. The property *NumberOfSeatsAvailable* is valid only when the flight is in the *scheduled* state and will return 0 or more available seats. In all other states, it throws an *InvalidRequestException*. The number of seats defaults automatically or it can be specified by setting the *SeatingCapacity* property.
   - Ja*va/C++:* Access these properties using the *setXxx*() and *getXxx()* method/member function.
   - C#/VB: Access these properties using "dot notation" (e.g. y.Xxx).

2. A flight can only be approved by the flight coordinator whose name is "Fred". If anyone else tries to approve a flight, the *approve* method should throw an *InvalidArgumentException*.
3. All other operations are valid in all states with most exhibiting the same behavior in all states.

Your Mission:

Write a set of unit tests for the *NumberOfSeatsAvailable* property.
(NB: Do not write tests for any other methods at this time! We will do that in Exercise 3.)

At your disposal:

- A test project already set up containing all the relevant resources including:
- An executable library (Jar, Assembly, Lib etc.) containing all relevant classes. You will not have access to the source until a later exercise.
- An xUnit test class called `Exercise1TestCase`. This class contains a single test method test1() that fails (intentionally).
- A helper class called FlightTestHelper with convenient methods for creating flights in various states.
    - *Hint: You can use the "name completion" feature of the development environment (IDE) to discover the available methods. See the language-specific "Cheat Sheet" if you don't know how to do this in your IDE.*

To Do:

It will be helpful during the exercise debrief to take note here while doing each step.

1. Open the project "Exercise-01" and run `Exercise1TestCase`. Verify that it runs a single test with a single failure.
    - *Refer to the language-specific Cheat Sheet for the language and IDE you are using for details of how to run the test and other useful tips & tricks.*

2. Define the test conditions of the NumberOfSeatsAvailable property that you need to verify.

3. Modify the test method to test at least one of these conditions.

4. Add additional test methods to test other conditions you identified.
    - Recall: we are only testing this one method at this time.

Exercise 2 Instructions

Purpose

Practice identifying test conditions and test cases based on a UML description of an entity class.

System Description

See Exercise 1.

Your Mission:

Define the test conditions, test scenarios and test cases for the Flight class. You will use these when writing xUnit tests for the Flight class in the next exercise.

At your disposal:

The description of the system provided for Exercise 1.

To Do:

1. Identify the test conditions you need to test.
   - Make sure you test each state transition of the Flight class.
   - Make sure you test each *public* method of the Flight class.
   - Make sure that you are trying various values for any method arguments. Use *boundary values* between *equivalence classes* to keep the number of values manageable.
   - Make sure that you are trying each method in each state.
   - Make sure you have identified the expected outcome for each test condition.

   Were you able to come up with a systematic way to ensure all the test conditions were identified? How?

2. What are the equivalence classes of the test conditions?

3. Define the test cases you need to verify these test conditions.

4. Did you notice any similarities between the tests? What was similar? How could you organize the test code to take advantage of the similarities?

5. If you have time review the Exercise-02-Result document (available in PDF format.)

Exercise 3 Instructions

Purpose

Gain experience automating the test conditions you have identified as test cases.

System Description

See Exercise 1.

Your Mission:

Based on the test conditions, test scenarios and test cases you identified for the Flight class in the previous exercise, create a complete xUnit test suite to verify the functionality. Make sure you follow the style guidelines.

At your disposal:

- The description of the system provided in Exercise 1.
- A predefined project called "Exercise-03" configured to interact with the executable containing the Flight class.
- Some starting tests for getNumberOfSeatsAvailable (which are the same as Exercise-01-Solution.)
- The test conditions and test cases you identified in Exercise 2 (or you may choose to start with the test conditions in Exercise 2 Results.pdf.)

To Do:

1. Open the project for Exercise 3.

2. Run the tests in the FlightTest test class and verify it has 3 successful tests and no failures.

3. Add additional test methods to automate each of the test conditions you identified in the last exercise.

4. Run your tests frequently to gauge your progress.

5. Refactor the tests into several test classes to group tests with common attributes. Which approach seems to be the best choice? And why?

6. If you have time, load and review the Exercise-03-Result project to see several ways to organize and name these tests. Record your reactions and impressions here:

Exercise 4 Instructions

Purpose

Practice identifying component test conditions and test cases based on a use case of the component and the component dependencies. This is a paper exercise (no code will be written, yet.)

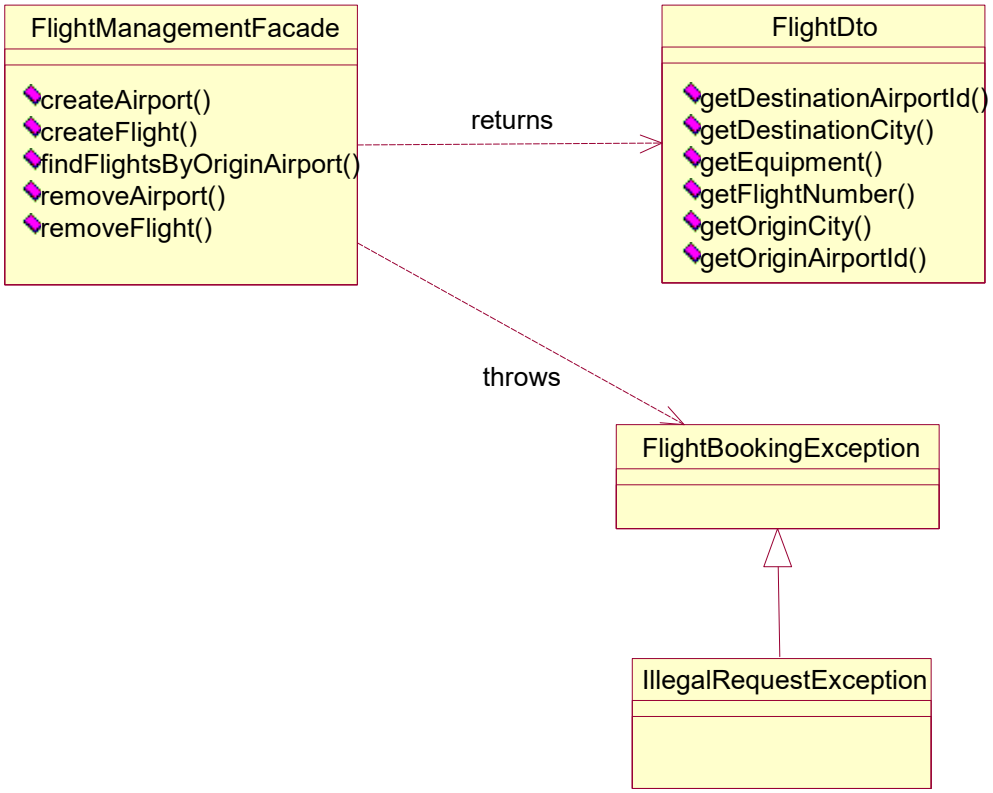
System Description

The functionality of the system is described in the following use case:

- Find Flights From Airport

The system has been partitioned into a presentation component (not discussed here) and a façade that encapsulates all the business logic of the system and is used by the user interface for all functionality. Refer to the interface definitions in the code for details of the API.

The following class diagram summarizes the interface of the API façade class.



The `FlightDto` class is used to return information about flights within the system. The system does not return the actual flight objects that may or may not exist inside the system. The class `FlightManagementFacadeImpl` implements this interface and will be our system under test.

`FlightManagementFacadeImpl` depends on the `FlightDataAccess` component. When requested to find the flights that match a certain criteria, the `FlightDataAccess` component will either return a list of matching objects or it will throw a `DataAccessException` that describes the nature of the problem (e.g. database unavailable, undefined, etc.) Note that we are not testing the functionality of t he `FlightDataAccess` component here so it can be replaced if necessary.

Your Mission:

Define the test conditions, test scenarios and test cases for the FlightManagementFacade. (You will use some of these when writing xUnit tests for the FlightManagementFacade class in Exercise-05A.)

At your disposal:

- The description of the system provided here including the use cases.
- The interface definition for the system façade in Exercise 5.

To Do List:

1. Read the Use Cases.
2. Open the Exercise 5 project and review the interface definition for the `FlightManagementFacade`.
3. Start by identifying the test conditions for the `getFlightsByOriginAirport` method.

   (Hint: what are the various states that the system-under-test can be in? How does this affect the expected results of this method? What are the corresponding direct inputs? Indirect inputs?)

4. What are the equivalence classes (different kinds of outcomes) for the test conditions?

   (Hint: what are the various kinds of expected results of this method? What are the Givens and direct inputs (Whens) that would result in them? What are the Indirect inputs?)

5. Define the test cases you need to verify these test conditions.

6. If you have time, define test conditions for the other methods. Check to see which ones are already covered by the tests for `getFlightsByOriginAirport`. Define any new tests cases you need to cover these test conditions.

7. If you have time, review the Exercise 4 Results document in (available in PDF format.)

Use Case - Find Flights From Airport

Goal:

The user wants to find flights that are departing from a particular airport.

Initiating Actor:

End User or Travel Agent

Other Actors:

None

Scope:

System = Flight Management System

Trigger:

User requests information about flights

Success Scenario:

1. User requests a list of flights departing from a particular airport.
2. System verifies that the requested originating airport is valid.
3. System retrieves all the flights that show that airport as the origination point.
4. System displays information about the found flights to the user. The information includes departure city, destination city, flight number, departure time, arrival time and type of equipment.

Alternate Scenarios:

The following scenarios branch off from the main scenario at the indicated step

2a. Airport code is invalid

1. The system informs the user of the error (airport code was invalid)
2. Use case ends in failure; the user's goal is not achieved.

4a No flights are found

1. The system displays "You can't go anywhere from there! Try the bus."

Related Use Cases:

Flights are defined using the Create Scheduled Flight use case.

Exercise 5a Instructions

Purpose

Practice automating functional test conditions and test cases using intent-revealing assertions.

System Description

The functionality of the system is described in Exercise 4.

Your Mission:

Improve the assertion logic in the tests for the `getFlightsByOriginAirport` method.
(Leave the fixture setup logic as it is.)

At your disposal:

The description of the system provided in Exercise 4.
An initial set of tests for the system façade. These tests are convoluted and have complex assertion logic inline.

To Do List:

1. Review the Exercise 4 results. (Optional)
2. Open the project for Exercise-05a.
3. Run the xUnit test class, `FlightManagementFacadeTest`, to verify it has been installed correctly. What happens?

4. Review the provided `FlightManagementFacadeTest`. What problems do you see or smell?

5. Identify any additional test cases needed to cover the test conditions for the system.

6. Try replacing the sequence of assertions used to verify the fields of the `FlightDto` with a single call to `assertEquals`() and an "expected object". What happens?

7. Identify chunks of duplicated test code in the verification part of the test that might be refactored into custom assertions. Try using the "comment elimination technique." (Ignore the fixture setup logic as that will be the subject of Exercise 6.)

8. Refactor the existing tests to use the custom assertions you defined.

9. Use these custom assertions to automate the missing tests.

10. If you have time, write tests for the custom assertion.

11. If you have time, load and review the Exercise-05A-Results project (in your IDE.) Record your thoughts here:

Exercise 5b - Setup Instructions

Purpose

Practice making test condition "*given*"s visible by using intent-revealing fixture setup.

System Description

The functionality of the system is described in Exercise 4.

Your Mission:

Improve the fixture setup logic in the tests for the `getFlightsByOriginAirport` method. The goal is to make the *givens* of each test easy to understand by reading the first few lines of each test method. This should also reduce the overall amount of code and make it easier to write new tests by reducing the amount of code needed for each new tests.

At your disposal:

The description of the system provided in Exercise 4.
The tests and test utilities you created in Exercise 5a and the tests and test utilities in Exercise-05a-Result. For your convenience, the latter have already been copied into the project for Exercise-05b.

To Do List:

1. Open the project for Exercise-05b.
2. Review the provided `GetFlightsByOriginAirportTest` class. What problems do you see or smell?

   Run the tests to see what happens.

3. Start with a simple cleanup refactoring: In method `withOneOutboundFlight_...` extract a method called `givenAFlightDto` from the existing fixture setup code to create the DTO. (The other tests already call a similar method but this will give you practice in creating such method via refactoring.)

   What got in the way on your first attempt?

   (Hint: Fields and Constants are not passed into methods when we extract them; you may need to extract a local variable first to force it to be passed; you can inline the variable after the refactoring.)

4. Review the test conditions from Exercise 4. Think about how you could make the *given* part of test conditions obvious using Creation Methods. What Creation Methods would make this easy to do? (Aim for just one method per flight; that is, a test that requires 2 flights should only have 2 lines of fixture setup.)

   What information do you need to pass to the creation methods from a previously created flight?

What could you call the variables that hold the result to make it easier to understand what role the newly created test object plays in the test?

5. Try "roughing in" a few new tests for the missing test conditions (such as "sightseeing flight" and "2 flights on same route") using your proposed utility methods. (You don't need to implement the bodies of the utility methods at this point; just the signatures.)

6. Now try refactoring the existing tests to create these utility methods. What refactoring steps will make this easier? (You may need to delete the existing signatures you created in the last step if your IDE's refactoring tools object to them.)

7. Pretend you have another test class somewhere else in the system from which you would like to reuse some of these custom assertions and creation methods. Refactor your classes so that this other test class could use these methods *without* having to subclass the class `AbstractFlightManagementFacadeTestCase.`

8. Review all your tests paying particular attention to how well the test class and method names combine to articulate which test condition each test method verifies. If the class.method names don't provide a succinct summary of the test condition (all 3 parts: Given-When-Then) then try renaming either the class, the method or both to do so.

   Which style of test class organization did you choose? Testcase Class per Class, Testcase Class per Feature, or Testcase Class per Fixture?

9. If you have time, write tests for the custom assertion(s) and creation method(s) you created.

10. If you have time, read the Exercise 5b Results PDF document and open the project Exercise-05b-Results and review the class `GetFlightsByOriginAirportTest_3_FullyRefactored` and compare it with what you came up with.

11. If any of the refactorings didn't go smoothly, go back and try them again. You may choose to start with partially refactored classes in Exercise-05-Setup-Result project: `GetFlightsByOriginAirportTest_1_InlineUniqueIdsForAirports` and `GetFlightsByOriginAirportTest_2_AnonymousAirportCreationMethods` and extract Anonymous Creation Method(s) for creating the Airports. (Working backwards from the result may make it more obvious to you what "refactoring chess moves" to make.)

Exercise 6 Instructions

Purpose

Explore various ways of doing test fixture setup and teardown. Understand how the various setup and teardown mechanisms provided by xUnit work. Experience the impact of the two main choices of fixture strategy: Shared versus Fresh Fixtures.

System Description

Four tests exist for the getFlightsByOriginAirport façade method (in the FlightManagementFacadeTest). Each of these tests expects a certain number of airports and flights as part of their test fixture (the name of the test and the specific helper create methods used indicate what is needed by each test).

Your Mission:

Identify and remove the bad smells that exist in the fixture setup and teardown of the tests for the getFlightsByOriginAirport method.

Please follow the detailed instructions on page 2 to get the maximum value from this exercise.
So that the instructor(s) don't need to bother you, please tick off each item as you complete it.

At your disposal:

- The description of the system provided here including the use cases.
- The library (lib or JAR) containing the system façade (hint: use your IDE to see what methods it supports.)
- An initial set of tests (FlightManagementFacadeTest) for the system façade.
- A FlightManagementTestHelper class with some useful methods on it:
  - A method to create all standard airports and flights
  - A method to remove all airports and flights
  - Methods to return the expected flights for different kinds of airports
  - Methods to create airports and flights with airport codes guaranteed to be unique
- A pre-built mechanism for setting up a Shared Test Fixture.
  - *See the Language-Specific Instructions for details*
  - `FlightManagementTestSetup` decorator and a test suite (`AllTests`) that installs the decorator for you.
- Messages written to the Console to show you what is happening behind the scenes including calls to:
  - setUp
  - testMethods
  - tearDown
  - etc.

See page 2 for the list of things to do

Note: Some find it useful to do each step in a different class. This allows you to retry a previous step without having to undo any changes. There are two strategies for doing this:

1) Copy and rename the class for each subsequent step.
2) Or starting in step 2, move the setup/teardown methods to a subclass of FlightManagementFacadeTest and copy/paste just that class for each step

JUnit4, NUnit: Just do Run ... Test on a different subclass of FlightManagementFacadeTest. (You would have to duplicate or modify RunTests_Twice, though.) Using Eclipse or Resharper, you can do Refactor->Extract SuperClass to create a NoFixture superclass from which you can then create subclassed test classes. Simply select the 4 test methods and give the new super-class a suitable name.

CppUnit: You'll need to do this manually, create a new project for each step and create the subclass in that project. You'll also need to create a new module with a main() function,

To Do List:

Please follow these instructions EXACTLY, checking off each step as you finish it. Takes notes about what you observe in the spaces provided as we will be discussing what you observed (and why) at each step during the post-exercise discussion.

1. Install the Exercise 6 starting point
   - ❑ Open the project for Exercise 6.
   - ❑ Rerun the tests (in `FlightManagementFacadeTest`) to verify they have been installed correctly. Some of the tests should fail because the system now remembers airports and flights. How many fail?
     - o
   - ❑ Review the provided test cases to see why they are failing
     - o *Hint: look at the output in the Console; refer to the language-specific Cheat Sheet for how to see the console.*
     - o
     - o
   - ❑ Try running each of the test methods individually. What happens?
     - o *Java, C#, VB: There are several ways to do this described in the language-specific Cheat Sheet.*
     - o *C++/CppUnit: There is no way to do this without using the CppUnit GUI (which we aren't using.) You'll need to comment out all but one of the test registration macro calls...*
     - o
     - o
   - ❑ What behavior smell are you encountering?
     - o
     - o

2. Try moving the *Standard Fixture* setup from within each test into the `setUp` method on the `FlightManagementFacadeTest` class.
   - ❑ Move the call to `setupStandardAirportsAndFlights` into the previously empty `setUp()` method on your `FlightManagementFacadeTest` class.
     - o (Do not put anything into `tearDown() yet.`)
   - ❑ Run the tests. Does this solve the problem? Why or why not?
     - o
     - o
   - ❑ What behavior smell are you encountering?
     - o
     - o
   - ❑ Run the tests several times in a row.
     - o *NUnit:*
       - ▪ *We need to use the standalone NUnit GUI test runner to run the test assembly several times without reloading it. There is an option "Reload before each test run" which we need to deselect. Please refer to the language-specific instructions.*
     - o *JUnit4:*
       - ▪ *To simulate persistence between test runs we use a JUnit4 test suite to run the test classes several times in the same test run because the JUnit plug-in for Eclipse automatically reloads the SUT classes before every run.*
         - select `RunTests_Twice`.java in the Package Explorer.
         - select Run As -> JUnit Test from the Run menu or from the context menu.
     - o *CppUnit:*
       - ▪ *To simulate persistence between test runs we include the FlightManagementFacadeTest test class twice in a test suite that runs the test classes several times in the same test run. To avoid having to edit the main() function between each test run, we have included the FlightManagementFacadeTest in another project with the appropriate main() function called Exercise-06-RunTwice:*
         - select Exercise-06_RunTwice project in the Solution Explorer and make it the startup project.
         - Use <ctrl><F5> to run the tests.
   - ❑ What happens? How many tests are being run? Does each test have the same results the second time it is run? (Compared to the first time it is run from the RunTwice project.) Why not?
     - o
     - o
     - o *Hint: look at the output in the Console*
     - o *JUnit4: Watch out for the JUnit reporting bug causing confusing output in the JUnit Window.*

- *NUnit: Reload the tests in the GUI manually to see how they behave the first time they are run.*

❑ What behavior smell are you encountering?

3. Try building a *Shared Fixture* using the SuiteFixture Setup approach (which runs the setup code only once for the all the test methods on a testcase class.)
   - *CppUnit: Skip this section as CppUnit does not support SuiteFixture Setup. Resume at step 3C.*

❑ Add the call to `setupStandardAirportsAndFlights` into the one-time `SuiteFixtureSetUp` method and run the tests. (Do not put anything into the one-time `SuiteFixtureTearDown` method yet.)
   - *NUnit: We can run setup code once per class by moving the call to setupStandardAir…s method into the oneTimeSetup method which has the [TestFixtureSetup] attribute. Note: You may have to access the setupStandard… method statically (via the classname FlightManagementTestHelper.)*
   - *JUnit4: We can run setup code once per class by moving the call to setupStandardAir…s method into the oneTimeSetup method which is annotated with @BeforeClass. Note: You may have to access the setupStandard… method statically (via the classname FlightManagementTestHelper.)*

❑ What happens?

❑ Remove the `setupStandardAirportsAndFlights` from the `setUp` method of `FlightManagementFacadeTest` (if you haven't already done so.) Any better? How many tests are being run?

❑ Run the test suite several times in a row from the same test runner.
   - *NUnit:* Run the tests several times in a row using the standalone NUnit GUI.
   - *JUnit4: Run the tests via the class* `RunTests_Twice`

❑ What happens? Do you get the same results the 2nd (and subsequent runs) as the first run? Why? How many tests are actually being run?
   - *Hint: look at the output in the Console.*
   - *JUnit4: Watch out for the JUnit reporting bug causing confusing output in the JUnit Window.*

❑ Try running a single test from within *FlightManagementFacadeTest*
   - *See the Cheat Sheet for how to do this..*

❑ What happens?

3c. Try running the setup code just once using a Test Setup Decorator
   - *NUnit: Skip this section; NUnit doesn't support Decorators. Instead, use the SetupFixture approach described in 3d.*

❑ Add the call to `setupStandardAirportsAndFlights` into the setUp() method on `FlightManagementTestSetUp` and run the tests. (Do not put anything into the tearDown() method yet.)
   - *JUnit4: While JUnit4 doesn't support Decorators, you can use the JUnit3-based TestSetup decorator provide in the class FlightManagementTestSetup. You can run it by using the provided RunTestsDecorated class:*
     i. *select the class RunTestsDecorated.java in the Package Explorer.*
     ii. *Select Run As-> JUnit Test from the Run Menu or the context menu.*
   - *CppUnit: To run setup code only once per class, we need to use a TestSetupDecorator. You can add the call to setupStandardAir…s to the setUp function in the class FlightManagementTestSetup.cpp in the project Exercise-06-Decorated and make it the startup project.*

❑ What happens?

- ❑ *JUnit4 only:* Remove the `setupStandardAirportsAndFlights` from the `SuiteFixtureSetUp` method of `FlightManagementFacadeTest` (if you haven't already done so.) Any better? How many tests are being run?
  - o
  - o
  - o
- ❑ Run the test suite several times in a row from the same test runner.
  - o *JUnit4: Run the tests via the class* `RunTests_Twice`
  - o *CppUnit: Run the tests via the class* `RunTests_Decorated_Twice in the` *project Exercise-06-Decorated-Twice*
- ❑ What happens? Do you get the same results the 2nd (and subsequent runs) as the first run? Why? How many tests are actually being run?
  - o
  - o
  - o
  - o *Hint: look at the output in the Console.*
  - o *JUnit4: Watch out for the JUnit reporting bug causing confusing output in the JUnit Window.*
- ❑ Try running a single test from within *FlightManagementFacadeTest*
  - o *JUnit4: See the Cheat Sheet for how to do this.*
  - o *CppUnit: Sorry, no way to do this easily! So try running the whole* FlightManagementFacadeTest *without using the Decorator.*
- ❑ What happens?
  - o
  - o
  - o
  - o

3d. Try running the setup code just once using a Setup Fixture [NUnit]

NUnit Only: JUnit/CppUnit Skip this section;

- ❑ Open the project Exercise-06-Step3d SetupFixture.
- ❑ Review the class `FlightManagementTestSetUp`.
- ❑ Run the tests in the project using the NUnit GUI. (See NUnit cheat sheet for how.)
- ❑ Add the call to `setupStandardAirportsAndFlights` into the [SetUp] method on `FlightManagementTestSetUp` and run the tests. (Do not put anything into the tearDown() method yet.) What happens?
  - o
  - o
  - o
- ❑ Remove the call to `setupStandardAirportsAndFlights` from the `SuiteFixtureSetUp` method of `FlightManagementFacadeTest` (if you haven't already done so.) Any better? How many tests are being run?
  - o
  - o
  - o
- ❑ Run the test suite several times in a row from the same test runner.
  - o *NUnit4: Run the tests via the GUI with "reload" unchecked.*
- ❑ What happens? Do you get the same results the 2nd (and subsequent runs) as the first run? Why? How many tests are actually being run?
  - o
  - o
  - o
  - o *Hint: look at the output in the Console in the NUnit GUI's "test output" tab.*
  - o *Resharper: Watch out for the reporting bug causing confusing output in the Unit Test Sessions window.*
- ❑ Try running a single test from within *FlightManagementFacadeTest*
  - o *NUnit GUI: Select individual tests or suites of tests (See the Cheat Sheet for how to do this.)*
- ❑ What happens?
  - o
  - o
- ❑ Try cloning the `FlightManagementTestSetUp` class and change the namespace on it. (And change the text in the WriteLine's while you are at it.) Run the tests. What happens?
  - o

- ○
- ○

4. Try *cleaning up the test fixture*:
   - ❑ Add the call to the `removeStandardAirportsAndFlights()` method of the Test helper in the tearDown() method.
     - ○ JUnit4: @After
     - ○ NUnit: [TearDown]
     - ○ CppUnit: teardown() on `FlightManagementFacadeTest`
   - ❑ What happens? Why?
     - ○
     - ○
     - ○
   - ❑ Move the call to `removeStandardAirportsAndFlights()` to the *one-time* tear-down method. Now what happens?
     - ○ *JUnit4:Annotated with @AfterClass*
     - ○ *NUnit: [TearDown]attribute*
     - ○ *CppUnit: Skip this step*
     - ○
     - ○
     - ○
   - ❑ *JUnit4 & CppUnit only:* Next, move the call to the tear-down method on the Test Setup Decorator (`FlightManagementTestSetup`); Now what happens?
     - ○
     - ○
     - ○
   - ❑ Run the tests several times in a row. Now what happens? Why?
     - ○ *JUnit4: Run the tests via the RunDecorated_Twice class.*
     - ○ *CppUnit: Run the tests via the RunDecorated_Twice project.*
     - ○ *NUnit: Run the tests several times from the standalone GUI.*
     - ○
     - ○
     - ○
     - ○ Hint: look at the output in the Console
5. Try modifying the *Shared Test Fixture*:
   - ❑ Add a call to `TestHelper.addExtraFlights()` into the `setUp` method used to set up the Shared Fixture.
     - ○ *JUnit4, CppUnit: on FlightManagementSetup decorator*
     - ○ *NUnit: In the SuiteFixtureSetup method*
   - ❑ What happened? Why?
     - ○
     - ○
     - ○
   - ❑ What behavior smell are you encountering?
     - ○
     - ○
6. Try out the *Fresh Fixture with Unique-Ids* approach:
   - ❑ Change the tests to call the corresponding `FlightManagementTestHelper.createXnn()` instead `of findXxx()`.
   - ❑ Run the tests the same way you ran them in Step 5. Do some tests still fail? Why?
     - ○
     - ○
     - ○
   - ❑ Run them several times in a row from the same TestRunner window. What happens?
     - ○
     - ○
7. If you have time, load and review the Exercise-6-Results projects (in your IDE.) Record your thoughts here:

Exercise 6 Instructions

Purpose

Practice various ways of doing test fixture setup and teardown. Learn when to use a TestSetup decorator or SuiteFixture setup to manage a Shared Test Fixture. Understand the impact of choosing to use a Shared versus Fresh Fixture approach to testing.

System Description

Four tests exist for the getFlightsByOriginAirport façade method (in the FlightManagementFacadeTest). Each of these tests expects a certain number of airports and flights as part of their test fixture (the name of the test and the specific helper create methods used indicates clearly what is needed by each test).

Your Mission:

Identify and remove the bad smells that exist in the fixture setup and teardown of the tests for the getFlightsByOriginAirport method.

Please follow the detailed instructions on page 2 to get the maximum value from this exercise. So that the instructor(s) don't need to bother you, please tick off each item as you complete it.

At your disposal:

- The description of the system provided here including the use cases.
- The library (lib or JAR) containing the system façade (hint: use your IDE to see what methods it supports.)
- An initial set of tests (FlightManagementFacadeTest) for the system façade.
- A FlightManagementTestHelper class with some useful methods on it:
  - A method to create all standard airports and flights
  - A method to remove all airports and flights
  - Methods to return the expected flights for different kinds of airports
  - Methods to create airports and flights with airport codes guaranteed to be unique
- A pre-built mechanism for setting up a Shared Test Fixture.
  - *See the Language-Specific Instructions for details*
  - `FlightManagementTestSetup` decorator and a test suite (`AllTests`) that installs the decorator for you.
- Messages written to the Console to show you what is happening behind the scenes including calls to:
  - setUp
  - testMethods
  - tearDown
  - etc.

See page 2 for the list of things to do

Hot Tip: To make it easier to keep things straight, consider doing each step in a different class, or starting in step 2, move the setup/teardown methods to a subclass of FlightManagementFacadeTest and copy/paste that class for each step. This allows you to retry a previous step without having to undo any changes. Just do Run As JUnit test on a different subclass of FlightManagementFacadeTest. (You would have to duplicate or modify RunTests_Twice, though.) In Eclipse, you can do Refactor->Extract SuperClass to create a NoFixture superclass from which you can then create subclassed test classes. Simply select the 4 test methods and give the class a suitable name.

To Do List:

Please follow these instructions EXACTLY, checking off each step as you finish it. Takes notes about what you observe in the spaces provided as we will be discussing what you observed (and why) at each step during the post-exercise discussion.

1. Install the Exercise 6 starting point
   - ❑ Open the project for Exercise 6.
   - ❑ Rerun the tests (in `FlightManagementFacadeTest`) to verify they have been installed correctly. Some of the tests should fail because the system now remembers airports and flights. How many fail?
     - ○
   - ❑ Review the provided test cases to see why they are failing
     - ○ *Hint: look at the output in the Console; refer to the language-specific instructions for how to see the console.*
     - ○
     - ○
   - ❑ Try running each of the test methods individually. What happens?
     - ○ *Hint: refer to the language-specific instructions for how to run tests individually.*
     - ○
     - ○
   - ❑ What behavior smell are you encountering?
     - ○
     - ○
2. Try moving the *Standard Fixture* setup from within each test into the `setUp` method on the `FlightManagementFacadeTest` class.
   - ❑ Move the call to `setupStandardAirportsAndFlights` into the previously empty `setUp()` method on your `FlightManagementFacadeTest` class.
     - ○ (Do not put anything into `tearDown() yet.`)
   - ❑ Run the tests. Does this solve the problem? Why or why not?
     - ○
     - ○
   - ❑ What behavior smell are you encountering?
     - ○
     - ○
   - ❑ Run the tests several times in a row.
     - ○ *JUnit4: To simulate persistence between test runs we use a JUnit4 test suite to run the test classes several times in the same test run because the JUnit plug-in for Eclipse automatically reloads the SUT classes before every run.*
       - select `RunTests_Twice`.java in the Package Explorer.
       - select Run As -> JUnit Test from the Run menu or from the context menu.
         - ○ Note that there is a **bug** in JUnit/Eclipse that causes the results to display strangely in the JUnit4 GUI.
         - ○ Look in the console to see what's really happening each time the tests are run.
   - ❑ What happens? On subsequent runs do you get the same results as the first run? Why?
     - ○
     - ○
     - ○ *Hint: look at the output in the Console*
   - ❑ What behavior smell are you encountering?
     - ○
     - ○
3. Try using a *Shared Fixture* by running the setup code only once for the entire suite of tests.

- ❑ Add the call to `setupStandardAirportsAndFlights` into the `SuiteFixtureSetUp` method and run the tests. (Do not put anything into the one-time `SuiteFixtureTearDown` method yet.)
  - o *JUnit4: We can run setup code once per class by moving the setupStandard… method into the oneTimeSetup method which is annotated with the @BeforeClass. Note: You may have to access the setupStandard… method statically (via the classname FlightManagementTestHelper.)*
  - *a) JUnit4: You can also run the setup code only once for an entire test suite by using the JUnit3-based TestSetup decorator provided: the FlightManagementFacadeTestDecorator. You can run it by using the provided RunTestsDecorated class:*
    - o *select the class RunTestsDecorated.java in the Package Explorer.*
    - o *Select Run As-> JUnit Test from the Run Menu or the context menu.*

- ❑ What happens?
  - o
  - o
  - o
- ❑ Remove the `setupStandardAirportsAndFlights` from the `setUp` method of `FlightManagementFacadeTest` (if you haven't already done so.) Any better?
  - o
  - o
- ❑ Run the test suite several times in a row from the same test runner.
  - o *JUnit4: Run the tests via the class `RunTests_Twice`*
- ❑ What happens? Do you get the same results as the first run on subsequent runs? Why?
  - o
  - o
  - o *Hint: look at the output in the Console.*

4. Try *cleaning up the test fixture*:
   - ❑ Add the call `removeStandardAirportsAndFlights` method on the Test helper in the one-time teardown method. What happens? Why?
     - o
   - ❑ Run the tests via the decorator `FlightManagementTestSetup`.
     - o *JUnit4: Run the tests via the class `RunTestsDecorated`*
   - ❑ What happens? Why?
     - o
     - o
     - o
   - ❑ Make sure you have removed any setup from the decorated test class; now what happens?
     - o
     - o
   - ❑ Run the tests several times in a row.
     - o *JUnit4: Run the tests via the RunDecorated_Twice class.*
   - ❑ What happens? Why?
     - o
     - o
     - o Hint: look at the output in the Console
   - ❑ Try running the tests without using the decorator
     - o *JUnit4: Run* `FlightManagementFacadeTest` *class using Run As … JUnit Test. (It will also be listed in your Test Run History menu.)*
     - o *C#/NUnit users can skip this step since SuiteFixture Setup is not enabled/disabled dynamically.*
     - o Run the tests the same way you ran them back in step 2.
   - ❑ What happens? Why?
     - o .

- ○

5. Try modifying the *Shared Test Fixture*:
   - ❑ Add a call to `TestHelper.addExtraFlights()` into the `setUp` method used to set up the Shared Fixture.
   - ❑ What happened? Why?
     - ○
   - ❑ What behavior smell are you encountering?
     - ○
     - ○
6. Try out the *Fresh Fixture with Unique-Ids* approach:
   - ❑ Change the tests to call the corresponding `FlightManagementTestHelper.createXnn()` instead `of findXxx()`.
   - ❑ Run the tests without using the decorator.
     - See the Language-Specific Instructions (A.K.A. Cheat-Sheet) if you cannot recall how to do this.
   - ❑ Do they still pass? Why?
     - ○
     - ○
   - ❑ Run them several times in a row from the same TestRunner window. What happens?
     - ○

Exercise 7 Instructions

Purpose

- To practice testing a component in isolation of any dependent components. In this exercise, the dependent component is a source of *indirect inputs*.
- To identify the correct boundary conditions

System Description

The `TimeDisplay` component displays the current time in a string formatted in HTML. The following business rules apply to this clock component:

1) At noon, it should return:

   Noon

2) At midnight, it should return:

   Midnight

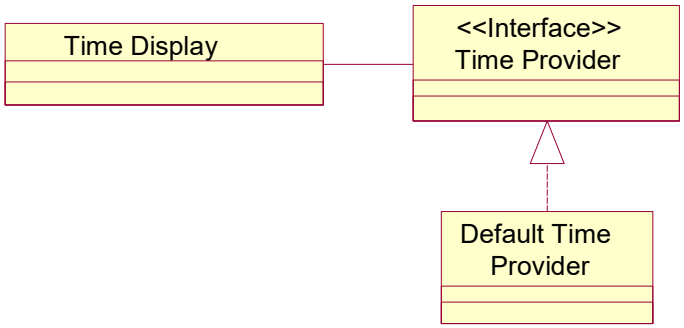
3) During all other times, the component should return the time in the format:
   hours:minutes AM/PM surrounded by the HTML tags. Some examples are shown below:

   2:30 PM

   12:02 AM

4) TimeDisplay will catch any exceptions that it receives and return the message text contained in the exception.

The system under test is called `TimeDisplay`, and the class diagram is shown below. `TimeDisplay` uses another component to produce the actual time called `TimeProvider`.



Your Mission:

Test the method `TimeDisplay.getCurrentTimeAsHtmlFragment()`. Define the test conditions for this method.

At your disposal:

- The interface description for the system under test (the getCurrentTimeAsHtmlFragment method of theTimeDisplay class) and its dependency (the getTime method of the DefaultTimeProvider class.)
  - *Note: In .Net, the interface used by the TimeDisplay class to access the current time is called ITimeProvider to comply with .Net idioms.*
- An initial set of Tests in the file `TimeDisplayTest.java`.

- A `TimeProviderTestStub` that allows you to set/get the hours and minutes using either the complete constructor or the setTime method. The arguments to either method are the hours as a 24 hour clock (e.g. 0 = midnight, 1= 1 am, 10 = 10 am, 12 = noon, 22 = 10 pm) and the minutes after the hour (0-59).

To Do List:

1. Open the project for Exercise 7.
2. Run the TimeDisplayTest and note the results.
    - How many tests pass?
    - How many fail?
3. Review the tests. Why do they fail?
4. Use the provided test stub, `TimeProviderTestStub`, to restructure the tests to have control over the indirect inputs. Note your results (*1/2 tests still fail).* Why does one test still fail?
5. What are the additional boundary conditions that have not been tested yet? Add additional tests for these boundary conditions.
6. If time permits, create a test to verify the behavior of `TimeDisplay` when it receives an exception. Create a new test stub that throws a hard-coded Exception when the getTime method is called. This will give you some experience writing your own hard-coded test stubs. Note the results of running this test (*the new test should pass*).
7. If time permits, copy the test created in step 6, but use the existing `TimeProviderTestStub`. Modify the existing `TimeProviderTestStub` to be configured to throw an exception
    - (hint: add a new method on `TimeProviderTestStub` to set the exception, and modify the `getTime` method to throw the exception that was set). This will give you some experience writing configurable test stubs. Note the results of running this test
    - (*the new test should pass*).
8. If time permits, create a test to verify the behavior of `TimeDisplay` when it receives a null date. Create a new test stub to return a null date when the `getTime` method is called. What result do you expect? What does the specification say about the handling of null dates?
9. If you have time, review the Exercise-07-Result project. Record your impressions here:

Exercise 8 Instructions

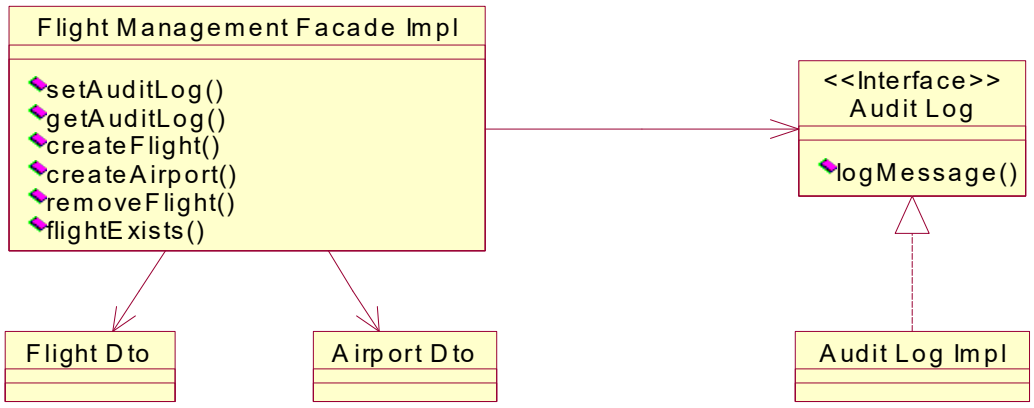
Purpose

Practice using Test Spies and Mock Objects to test indirect outputs.

System Description

Certain FlightManagementFacade methods are required to be logged. It must log the **date, user, action code, and details**. Nothing is logged if the method encounters an error and does not complete successfully.

Façade Method	Log Message Action Code (Constant in helper)	Log Message Detail
CreateAirport	'CreateAirport' (CREATE_AIRPORT_ACTION_CODE)	Airport code
CreateFlight	'CreateFlight' (CREATE_FLIGHT_ACTION_CODE)	Flight number
RemoveFlight	'RemoveFlight' (REMOVE_FLIGHT_ACTION_CODE)	Flight number



Your Mission:

Write unit tests that verify that AuditLog.logMessage() method is being called as specified.

At your disposal:

All of the classes and methods indicated in the diagram above, including:

- FlightManagmentFacadeImpl that calls AuditLogImpl via the AuditLog interface.
- AuditLogTestSpy (implements the AuditLog interface.)
- MockAuditLog (implements the AuditLog interface.)
- Method on FlightManagementFacadeImpl (setAuditLog) to install either of these test stubs.
- Methods and constants on the Helper class:
  - CREATE_FLIGHT_ACTION_CODE
  - CREATE_AIRPORT_ACTION_CODE
  - REMOVE_FLIGHT_ACTION_CODE
  - TEST_USER_NAME
  - getTodaysDateWithoutTime()
- Methods on the FlightManagementFacadeTest class
  - assertAuditLogCalledCorrectly

To Do List:

1. Open the project for Exercise 8.
2. Run the FlightManagementFacadeTest and note the results. **(4/4 tests should pass).**

3. Inspect the tests; do the tests verify the audit logging is performed correctly?

4. Test Spy: Enhance the existing *removeFlight_LeavesNoFlight* to verify the *indirect outputs*.
   - ❑ Rename the test to more accurately reflect the purpose (e.g. `RemoveFlight_ShouldLog_TestWithSpy`)
   - ❑ Create an instance of AuditLogTestSpy, and install it on the façade (use the *setAuditLog* method or use the *AuditLog* property depending on your programming language.)
   - ❑ After invoking the removeFlight method on the façade use an EqualityAssertion to compare some of logged fields captured by the Test Spy with expected values.
   - ❑ Try using the custom assert assertAuditLogCalledCorrectly to compare all the relevant fields' actual values with the expected values.
   - ❑ Rebuild and run xUnit. Note your results **(4/4 tests should pass)**

5. Mocking: Create a new test by copying the one created in step 4.
   - ❑ Name the new test something like *RemoveFlight_ShouldLog_TestWithMockLog*
   - ❑ Create an instance of MockAuditLog and configure it using the *setExpectedNumberOfCalls* method with one parameter (the integer value 1).
   - ❑ Configure it using the *setExpectedLogMessage* method with 4 parameters: date, user name, action code, and detail (flight #).
   - ❑ Install the MockAuditLog on the façade.
   - ❑ In the verify outcomes part of the test, call the verify() method on the test stub
   - ❑ Rebuild and run your tests. Note your results **(5/5 tests should pass).**

6. Write a new test for the CreateAirport method using a Test Spy. (I.e. Repeat step 4.)
   - ❑ Note your results **(1/6 tests fail)**. What bug did you find?[i]

7. Write a new test for the CreateAirport method using a Mock Object. (I.e. Repeat step 5.)
   - ❑ Note your results **(2/7 tests fail)**. What bug did you find?[ii]

8. Write a new test for the CreateFlight method using a Test Spy. (I.e. Repeat step 4.)

- ❑ Note your results **(the new test should pass).** [iii]

9. Write a new test for the CreateFlight method using a Mock Object. (I.e. Repeat step 5.)
   - ❑ Note your results. Does it have the same results as the previous step? If not, have a close look at the error message and the test.

   - ❑ Is it possible to properly configure the Mock Object in this circumstance? Hint:when are flight numbers created vs. when is the Mock Object configured vs. when is the verification of the actual parameters?

10. If you have time, review the *Constructor Test*.
    - ❑ Retrieve the AuditLog from the façade and use an assertion to verify that the correct audit log is installed by default.
    - ❑ Note your results. What bug did you find?[iv]

    - ❑ What did you just learn about testing with test stubs (is it enough?).

11. This exercise used static stubs or mock objects. If you have time, use your favourite mocking tool or framework to repeat each of the above.
    - ❑ Compare the resulting tests with their hand-rolled equivalents. Are the resulting tests easier or harder to understand? To write?

12. If you have time, review the code in the Exercise 8 Result project.

If you are new to mocking and stubbing, and would like to explore dynamic mock objects, see : http://www.easymock.org or http://www.jmock.org or http://mockito.org/ for Java, http://sourceforge.net/projects/easymocknet/ for .Net or http://www.assembla.com/wiki/show/hippomocks for C++.

[i] If the bug wasn't related to the action code, then there is a problem in your test.
[ii] If it isn't the same bug as the previous step, then there is a problem with your test.
[iii] If your test fails there is a problem with your test: remember that the flight number is not known until after the flight is created.
[iv] If the bug wasn't related to the existence of a real audit log, then there is something wrong with your test.

Exercise 9 Instructions

Purpose

To devise strategies for improving the testability of an existing class.

System Description

Given a flight number for the inbound plane and a flight number for the outbound plane at a particular airport, the `FlightConnectionAnalyserImpl` class calculates the connection time between the 2 flights. It determines the connection time as 'legal' in the following conditions:

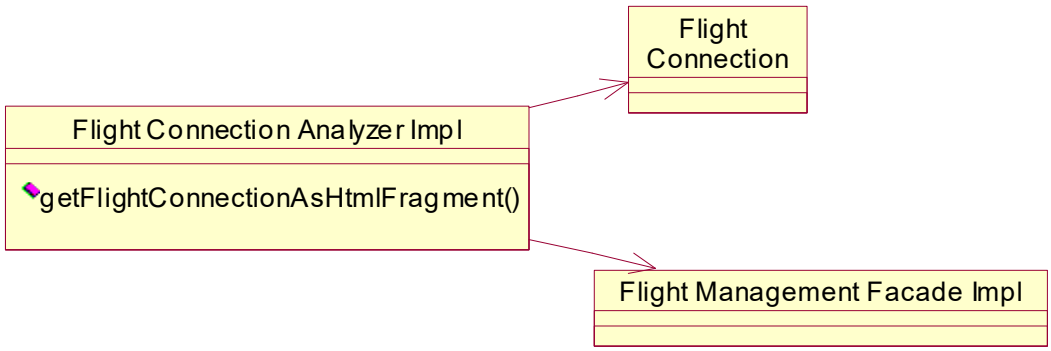
- incoming flight and outbound flight are from the same airline, and the connection time is greater than 29 minutes
- incoming flight and outbound flight are from different airlines, and the connection time is at least 1 hour

The `FlightConnection` encapsulates the connection time in minutes and indicates if the connection is legal.

In addition, the business user requires the presentation web page to highlight any 'illegal connections' in bold, red font.

Refer to the interface documentation for details of the API. *(See the language-specific instructions for details of how to do this.)*

The following class diagram summarizes relationships of the relevant classes.



Your Mission:

Develop a strategy for refactoring `FlightConnectionAnalyzerImpl` to make it easier to test.

At your disposal:

The description of the system provided here.
The code to be refactored (located in the Exercise-10 project)
Source code for `FlightConnectionAnalyzerImpl`, which is the target of your refactoring (located in the Exercise-10 project.)

To Do List:

1. Import the project for Exercise-10 and run the unit tests.
2. Examine the test results. Can you determine what the bug is by looking only at the names of the tests that are failing?

3. Review the System description above and the source code for the `FlightConnectionAnalyserImpl` and dependant classes in the Exercise-10 source code. What are the different domains/concerns that are contained in this class? Which of the domains are we actually trying to test? Should any of the domains already be tested elsewhere?

   Domain 1:

   Domain 2:

   Domain …:

4. List at **least** two test conditions for **each** domain/concern identified above.
   - For domain 1:

     1.

     2.

   - For domain 2:

     1.

     2.

   - For domain …:

     1.

     2.

5. What changes could you make to the source code to make it easier to test without introducing any new classes? (Suppose you were planning to go into production next week.) What testing techniques should you use in tests for each of the subdomains listed above? Do you need to introduce a Test Double? If so, what do you need to replace? What kind of Test Double could you use?
   - Subdomain 1:
   - Subdomain 2:

6. What changes would you make to separate the (sub)domains into separate classes? What new class(es) would you need? What method(s) would you move to this new class(es)? What new methods would you create? What method remains in `FlightConnectionAnalyzerImpl`.

   Class 1:

   Class 2:

   Class 3:

7. What testing techniques should you use for each of the classes listed above? Do you need to introduce a Test Double? If so, what object do you need to replace? What kind of Test Double will you use?

   Class 1:

   Class 2:

   Class 3:

Exercise 10 Instructions

Purpose

Make FlightConnectionAnalyzerImpl easier to test by refactoring it.

System Description

See Exercise 9.

Your Mission:

Implement the refactoring in small steps that minimize the risk involved.

At your disposal:

- Your solution to exercise 9 and the solutions discussed in class.
- FlightConnectionAnalyzerStub, a hand-coded test stub to enable you to stub out the FlightConnectionAnalyzerImpl with FlightConnection objects constructed in the tests.
- The assertFlightConnectionsEqual custom assert method on FlightConnectionAnalyzerTest.
- A Helper class with all the methods you should require to create anonymous flights.

To Do List:

1. Create a copy of the project Exercise 10 and call it Exercise-10-Result-1.

2. Implement the low-risk solution which allows you to test the HTML and Business Logic sub-domains separately. Evolve the existing tests so that you have separate tests for the each sub-domain.
    - You may find it useful to code at least one test for each sub-domain before you do the refactoring to see what the refactoring target looks like.

    - Extra Points: Since the analyse() method is stubbed out when testing `getFlightConnectionAsHtmlFragment`, do you need to create registered flights to build your connections? Consider creating some new utility methods that create legal and illegal connections without creating flights or airports in the Flight Management System.

3. Now create a copy of your Result-1 project and call it Exercise-10-Result-2.

4. Implement a more elegant solution that separates the sub-domains into at least 2 separate classes. Make sure you have enough tests to protect you before you start the refactoring.

5. If your solution doesn't already do this, make another copy (Result-3) and try removing the need for the complex utility methods by further refactoring the Analyser to make it independent of the FlightManagmentFacade.

TDD Exercise 1 Instructions

Purpose

To incrementally develop some simple functionality through Test-Driven Development and refactoring.

System Description

You are building a Roman Numeral translator for Arabic numbers.

Your Mission:

Implement the `RomanNumeralGenerator` in a test-Driven approach. At each step, use the simplest algorithm that could possibly work. When you see duplication in the logic, refactor it to simplify it. Try to do refactoring only when you have a green bar. (Consider disabling the yet-to-be-passed test while refactoring.)

At your disposal:

- A start on the tests for the generator in TestRomanNumeral.java.
- A very-simple hard-coded implementation to evolve in RomanNumeralGenerator.java.
- A description of the Roman Numeral system (from Wikipedia):

> Roman numerals are based on seven symbols: a stroke (identified with the letter I) for a unit, a chevron (identified with the letter V) for a five, a cross-stroke (identified with the letter X) for a ten, a C (identified as an abbreviation of *Centum*) for a hundred, *etc.*:
>
> | Symbol | Value |
> |---|---|
> | I | 1 (one) (*unus*) |
> | V | 5 (five) (*quinque*) |
> | X | 10 (ten) (*decem*) |
> | L | 50 (fifty) (*quinquaginta*) |
> | C | 100 (one hundred) (*centum*) |
> | D | 500 (five hundred) (*quingenti*) |
> | M | 1,000 (one thousand) (*mille*) |
>
> Symbols are iterated to produce multiples of the decimal (1, 10, 100, 1,000) values, with V, L, D substituted for a multiple of five, and the iteration continuing: I "1", II "2", III "3", V "5", VI "6", VII "7", etc., and the same for other bases: X "10", XX "20", XXX "30", L "50", LXXX "80"; CC "200", DCC "700", etc. At the fourth iteration, a subtractive principle may be employed, with the base placed before the higher base: IIII or IV "4", VIIII or IX "9", XXXX or XL "40", LXXXX or XC "90", CCCC or CD "400", DCCCC or CM "900".

To Do List:

1. Open the project for Exercise TDD 1.
2. Review the test (TestRomanNumerals) and the SUT (RomanNumeralGenerator).
3. Run the test, note the result.
4. Make the tests pass for 0, 1
5. Add the test for 2 and make it pass.
   - Is there any way to simplify the code? The description of the system provided here including the use cases.

6. Add the test for 3 and make it pass.
   - Is there any way to simplify the code?

7. Pick the next case to add? What should it be? 10, 20, 30 is similar to 1,2,3 but with a different letter. Or you could stick with the next number in sequence (4) or the next unique Roman (V=5) Write your decision here:

8. Add the first test for this equivalence class and make it pass.
9. Add the next test for this equivalence class and make it pass.
   - Can you remove any duplication of code? Can you change the code you have so that duplication is easier to detect?

10. Pick the next equivalence class to implement. Pick one specific test to add and make it pass.
11. Add another test case for the same equivalence class and make it pass.
12. Now generalize the solution to remove code duplication.

13. Lather, Rinse, Repeat. Record your chosen sequence here:

14. If you didn't implement the "Subtractive" approach (e.g. "IV" instead of "IIII"), adjust your tests to reflect it and get them to pass. Then refactor the additional logic.
    - How many tests did you change or add at one time?

    - Was this too big a step? If so, try it again with smaller steps.

Evaluation Form

Testing for Developers with XUnit

Day

One Two Three Workshop

Overall Impression

Great Very Good Average Below Average Poor

Would you recommend it to an interested collegue if it were offered again?

Definitely yes Neutral Advise Against

Length of Course

Too Long Good Too Short

Level of Detail

Too Detailed Good Insufficient Detail

Relevance of Information

Very relevant Somewhat relevant Not very relevant

Clarity of Presentation

Very clear Somewhat clear Not very clear

Usefulness of Exercises

Very relevant Somewhat relevant Not very relevant

Complexity of Exercises

Too complex Just Right Too Simple

Language of Study

Java C++ C#

Level of Previous XUnit Experience

None Heard of it Done the Tutorial Written lots of Tests

Is there anything you would have liked to have known before you took this course?

Additional Comments?

Evaluation Form

Testing for Developers with XUnit

Day

One Two Three Workshop

Overall Impression

Great Very Good Average Below Average Poor

Would you recommend it to an interested collegue if it were offered again?

Definitely yes Neutral Advise Against

Length of Course

Too Long Good Too Short

Level of Detail

Too Detailed Good Insufficient Detail

Relevance of Information

Very relevant Somewhat relevant Not very relevant

Clarity of Presentation

Very clear Somewhat clear Not very clear

Usefulness of Exercises

Very relevant Somewhat relevant Not very relevant

Complexity of Exercises

Too complex Just Right Too Simple

Language of Study

Java C++ C#

Level of Previous XUnit Experience

None Heard of it Done the Tutorial Written lots of Tests

Is there anything you would have liked to have known before you took this course?

Additional Comments?