Part III [Objectives]

2. Unit, Integration, and Acceptance Testing

SE-307 Software Testing Techniques

http://my.ss.sysu.edu.cn/wiki/display/SE307/Home

Instructor: Dr. Wang Xinming, School of Software, Sun Yat-Sen University

Overview

- Unit testing
  - JUnit framework for test automation
  - Use mock objects as test oracle
  - Tricks and Best Practice
- Integration testing
  - Test stub
  - Dependency injection
- Acceptance testing
  - FitNesse framework

Unit testing

- **Unit testing**: Looks for errors in sub-systems in isolation.
  - Generally a "subsystem" means a class or object
- Benefits of unit testing:
  - Find bugs earlier.
  - Easier to locate bugs when failure occurs.

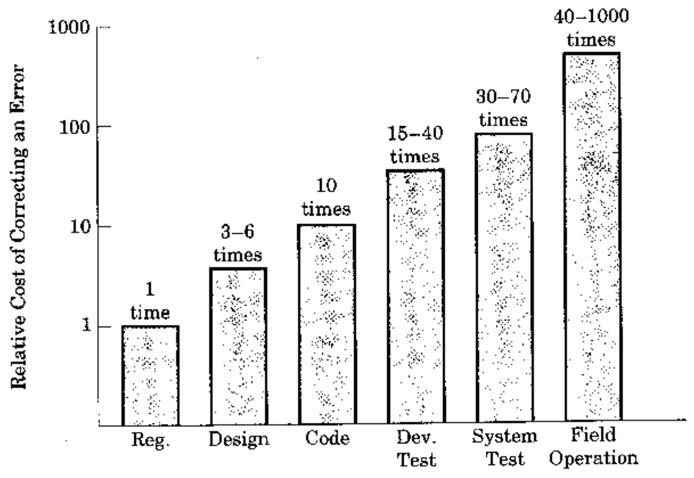


FIGURE 8.1. Relative cost of correcting an error.

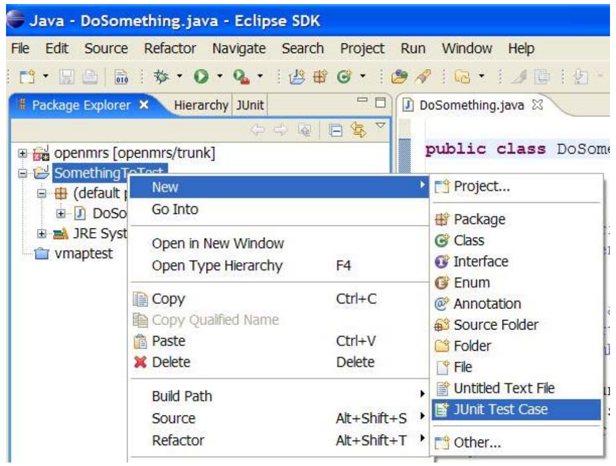
Problems in Unit Testing

- **Test management**
  - Test-driven development
- **Test adequacy**
  - White-box: control flow, data flow, logic
  - Black-box: level-1, level-2
- **Test oracle**
  - Assertion
  - Contracts
  - Mock objects
- **Test generation**
  - Symbolic and concolic test generation
  - Random test generation
  - Genetic test generation
- **Test automation**
  - The JUnit framework

The JUnit Framework

- A unit test automation framework for Java, developed by *Erich Gamma* and *Kent Beck*.
  - Define test cases as Java classes
  - The execution of test cases can be automated
- It is integrated into Eclipse IDE with a GUI.

```
public class DateTestCase {

    public void testDate() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
     }
}
```



JUnit 3.x framework

Steps to create a JUnit test case

- create a **test case class** a sub-class of junit.framework.TestCase
- (optional) create shared objects with **setUp** and **tearDown**
- add to it one or more **test methods**
- add one of more test operations and **assertions** into each test method

```
import junit.framework.TestCase;
public class StackTestCase extends TestCase{

   public StackTestCase(String name) {
        super(name);
   }

   Stack aStack;
   protected void setUp() {
       Stack aStack = new Stack();
       aStack.push(10);
       aStack.push(-4);
   }
   protected void tearDown() {}

   public void testPop () {
       aStack.pop();
       assertSame(aStack.pop(), 10);
   }

   public void testEmpty() {
       assertFalse(aStack.isEmpty);
   }
}
```

JUnit Assertions

`assertTrue`	*(message, test)*
`assertFalse`	*(message, test)*
`assertEquals`	(*message*, **expected**, **actual**)
`assertNotEquals`	(*message*, **expected**, **actual**)
`assertSame`	(*message*, **expected**, **actual**)
`assertNotSame`	(*message*, **expected**, **actual**)
`assertNull`	(*message*, **obj**)
`assertNotNull`	(*message*, **obj**)
`fail`	(*message*)

The use of the “fail” assertion

- Test unit behavior on exceptions
  - We expect a normal behavior and then no exceptions.
  - We expect an anomalous behavior and then an exception.

We expect a normal behavior

```
try {
   // We call the method with correct parameters
   object.method("Parameter");
} catch(PossibleException e){
   fail("method should not fail !!!");
}
```

```
class TheClass {
 public void method(String p)
            throws PossibleException
  { /*... */ }
}
```

We expect an exception

```
try {
  // we call the method with wrong parameters
  object.method(null);
  fail(“method should fail!!");
} catch(PossibleException e){
  // OK
}
```

```
class TheClass {
 public void method(String p)
            throws PossibleException
  { /*... */ }
}
```

Test Automation with JUnit 3.x

- The framework automatically finds all sub-classes of junit.framework.TestCase and then
  - Executes all of their public test methods and ignores everything else.
    - i.e. those whose name starts with “test”
  - If setUp/tearDown are defined, call them before/after the call to every test method

- Test case classes can contain “***helper methods***” that are:
  - Non public methods, or
  - Methods whose name does not begin with “test”

JUnit 4.x

- The same:
  - JUnit 4 can still run JUnit 3 tests
  - All the old **assert*XXX*** methods are the same
- New features:
  - Supports annotations (@)
  - Add protection against infinite loops
  - Supports assertions on exception directly
  - Supports parameterized unit tests

Annotations ‘@’

- Introduced in J2SE 5 to support metadata
  - Used for code documentation, compiler processing (e.g. @Deprecated, @Override), code generation, runtime processing
  - New annotations can be created by developers

- Use annotations to create JUnit 4.x test case class
  - Don’t extend **junit.framework.TestCase**; just use an ordinary class.
  - Use annotations instead of special method names:
    - Instead of a **setUp** method, put @**Before** before some method
    - Instead of a **tearDown** method, put @**After** before some method
    - Instead of beginning test method names with ‘**test**’, put **@Test** before each test method

Infinite Loop and Exception

- To detect infinite loops, an execution time limit can be used.
  - The time limit is specified in milliseconds. The test fails if the method takes too long.

```
@Test (timeout=10)
public void greatBig() {
    assertTrue(program.ackerman(5, 5) > 10e12);
}
```

- Some method calls should throw an exception. We can specify that an exception is expected.
  - The test will pass if the expected exception is thrown, and fail otherwise

```
@Test (expected=IllegalArgumentException.class)
public void factorial() {
    program.factorial(-5);
}
```

Parameterized Tests

- Using **@RunWith(value=Parameterized.class)** and a method **@Parameters**, a test class is executed with several inputs

```
@RunWith(value=Parameterized.class)
public class FactorialTest {

    private long expected;
    private int value;

    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[ ][ ]
            { { 1, 0 }, { 1, 1 }, { 2, 2 }, { 120, 5 } });
    }

    public FactorialTest(long expected, int value) { // constructor
        this.expected = expected;
        this.value = value;
    }

    @Test
    public void factorial() {
        assertEquals(expected, new Calculator().factorial(value));
    }
}
```

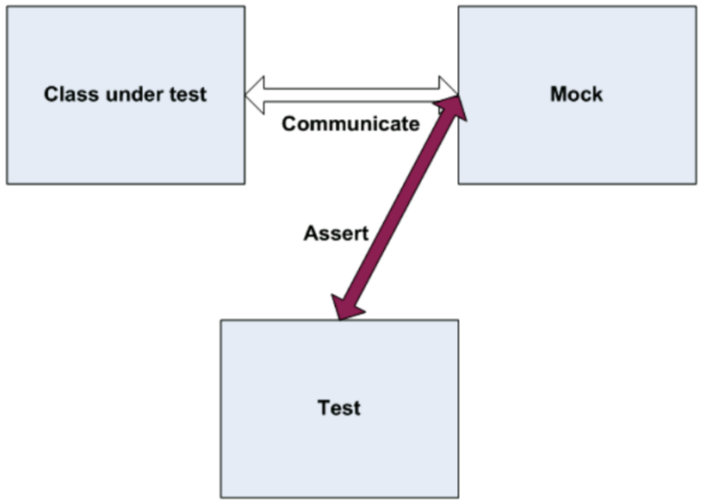
Parameters used to exercise different instances of the class

Test Oracle

- Expecting *state*
  - Assertion
  - Contracts
- Expecting *behavior* (i.e. interaction with external classes)
  - How?

"Mock" Objects (模拟对象)

- A fake object that decides whether a unit test has passed or failed by watching interactions between objects.
  - Used for testing whether the class under test interacts with external classes in an expected way ...
  - without the implementation of the external classes.
  - **Mock != Stub**



Mock Object Frameworks

- Mock object frameworks help with the process.
  - jMock (Java)
  - FlexMock / Mocha (Ruby)
  - SimpleTest / PHPUnit (PHP)
  - ...
- Frameworks provide the following:
  - Auto-generation of mock objects that implement a given interface
  - logging of what calls are performed on the mock objects
  - methods/primitives for declaring and asserting your expectations

Example using jMock

```
import org.jmock.integration.junit4.*;
import org.jmock.*;

@RunWith(JMock.class)
public class ClassATest {
    private Mockery mockery = new JUnit4Mockery();

    @Test  public void testACallsBProperly1() {

        InterfaceB mockB = mockery.mock(InterfaceB.class);

        A a = new A(...);
        a.setResource(mockB);

        mockery.checking(new Expectations() {{
            oneOf(mockB).method1("an expected parameter");
            will(returnValue(0.0));
            oneOf(mockB).method2();
        }});

        a.methodThatUsesB();

        mockery.assertIsSatisfied();
    }
}
```

Assume the class under test is “A”.

We want to test how A interacts with an external object that implements an interface InterfaceB

Create a mock object “mockB” that implements InterfaceB

Create an object of A and attach it to the mock object

Declare expectations for how A shall interact with the externa object through InterfaceB

Execute a method of A, should lead to calls on mockB

Assert the expected interactio

Tool: jMock

www.jmock.org

- Specifying objects and calls:
  - `oneOf(`**`mock`**`).X, exactly(`**`count`**`).of(`**`mock`**`).X,`
  - `atLeast(`**`count`**`).of(`**`mock`**`).X, atMost(`**`count`**`).of(`**`mock`**`).X,`
  - `between(`**`min`**`, `**`max`**`).of(`**`mock`**`).X`
  - `allowing(`**`mock`**`).X, never(`**`mock`**`).X`

The above accept a mock object and return a descriptor that you can call methods on, as a way of saying that you demand that those methods be called by the class under test.

- Example:

```
Mockery mockery = new JUnit4Mockery();
InterfaceB mockB = mockery.mock(InterfaceB.class);
mockery.checking(new Expectations() {{
    atLeast(3).of(mockB).method1();
    atMost(3).of(mockB).method1();
}};
a.methodThatUsesB();
mockery.assertIsSatisfied();
```

“ We expect that `method1` will be called on `mockB` at least 3 times."

Expecting return values

- `will(action)`
  - actions: `returnValue(`**v**`)`, `throwException(`**e**`)`
- values:
  - `equal(`**value**`)`, `same(`**value**`)`, `any(`**type**`)`, `aNull(`**type**`)`, `aNonNull(`**type**`)`, `not(`**value**`)`, `anyOf(`**value1, ..,valueN**`)`
- Example:

```
Mockery mockery = new JUnit4Mockery();
InterfaceB mockB = mockery.mock(InterfaceB.class);
mockery.checking(new Expectations() {{
    atLeast(3).of(mockB).method1();
    will(returnValue(anyOf(1, 4, -3)));

}};
```

"I expect that `method1` will be called on `mockB` once here, and that it will return either 1, 4, or -3."

Advanced features

- Multi-sequences
  - Invocations that are expected in a sequence must occur in the order in which they appear in the test code.
  - A test can expect more than one sequences.
- Conditional invocations
  - A test can create state machines for mock objects.
  - An invocation can be constrained to occur during a state of one more state machines.
- Check out the manual by yourselves!

Tricks and Best Practice

- Keep the unit tests “small”
- Keep the unit tests independent with each other
- Design interface for the sake of unit tests
- Apply unit test patterns
- Use frameworks

Keep the Unit Test “Small”

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Avoid unnecessary redundancy
  - Don’t always copy and paste!
  - Use inheritance, table, sub-functions to reduce redundancy.
- Avoid complicated logic
  - No-No: recursion, nested loops, dynamic data structure…
  - Why?

Keep the Unit Test Independent

- Unit tests should be self-contained and do not rely on each other.

- "**Smells**" (bad things to avoid) in unit tests:
    - *Constrained test order* : Test case A must run before Test case B. (usually a misguided attempt to test order/flow)
    - *Tests call each other* : Test case A calls Test case B's method (calling a shared helper is OK, though)
    - *Mutable shared state* : Test cases A/B both use a shared object. If A breaks it, what happens to B?

Unit Tests Love Interface

- Why? Interfaces enable unit tests to replace external dependency with ***mocks*** and ***stubs***
- Without interface

```
method_under_test(…){
  ......
  time = System.currentTimeMillis();
  ......
}
```

The dependency on system time service cannot be changed in unit test.

- With interface

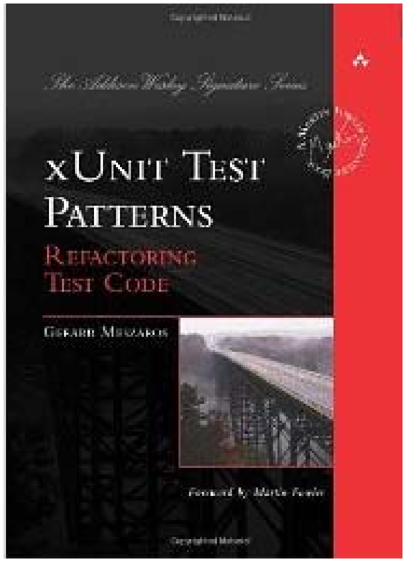
```
public interface IClock {
    long getTime();
}

method_under_test(…){
  ......
  IClock clock = ...;
  time = clock.getTime();
  ......
}
```

The dependency can be replaced with mocks or stubs.

Apply Unit Test Patterns

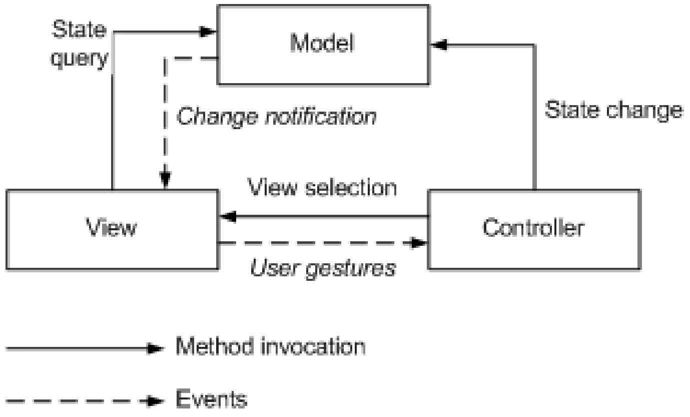
- pass/fail patterns
- collection management patterns
- data driven patterns
- performance patterns
- simulation patterns
- multithreading patterns
- stress test patterns
- presentation layer patterns
- process patterns

Read this book to know more

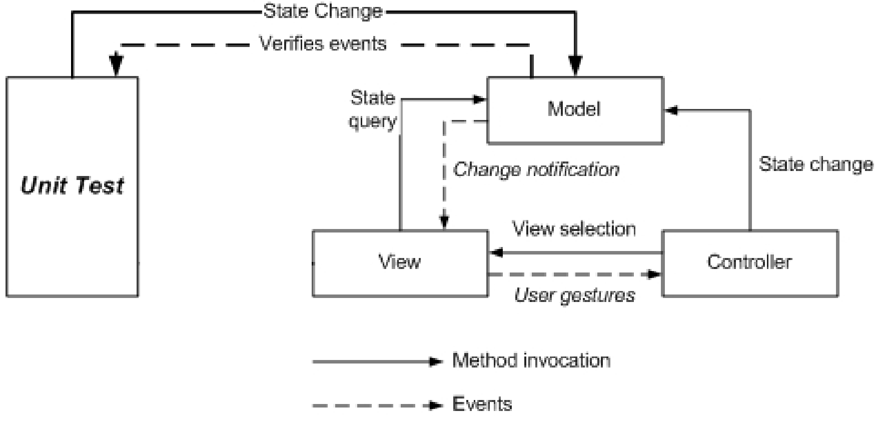


Example: MVC Test Pattern

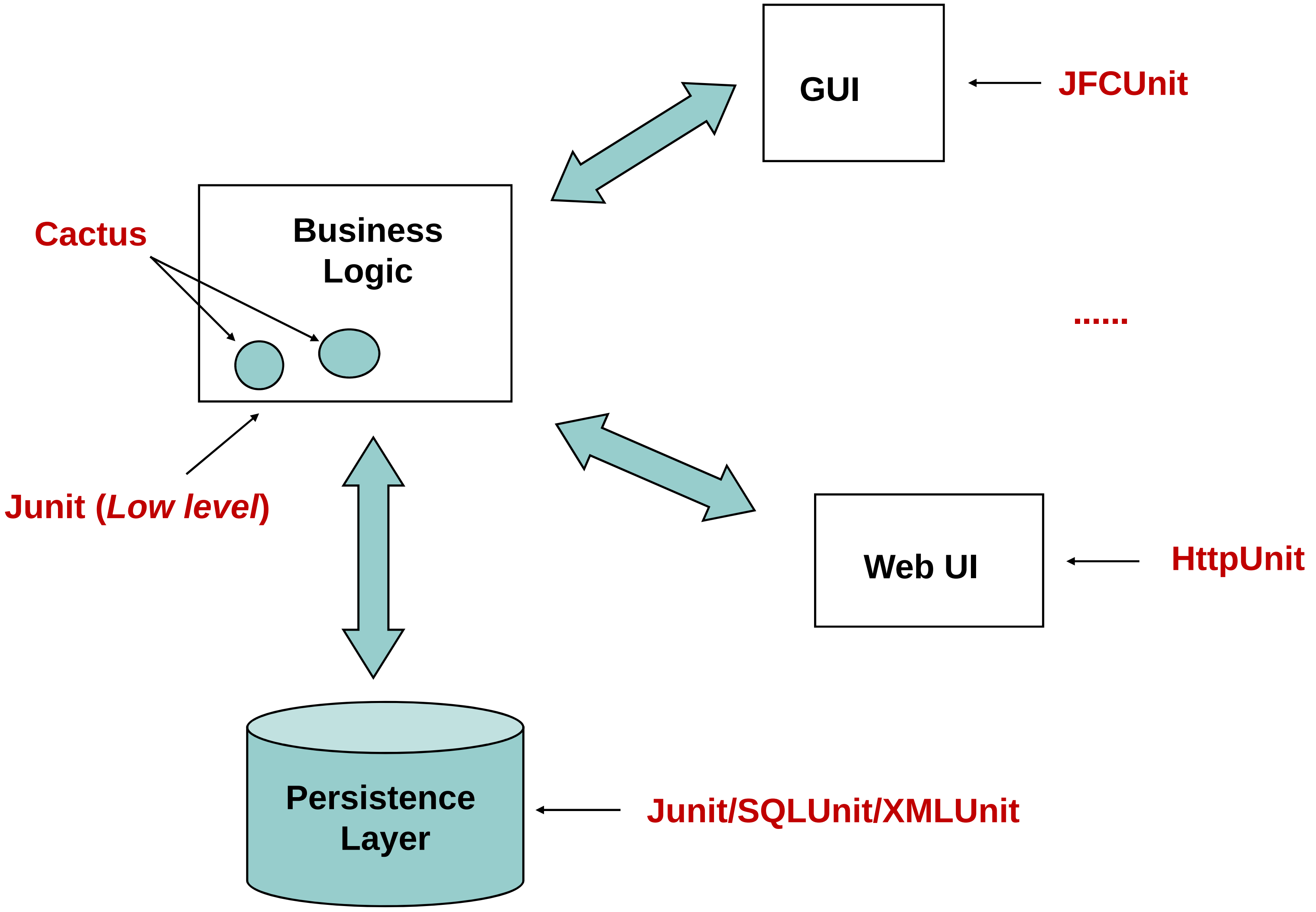
- MVC design pattern as explained by Sun: (http://java.sun.com/blueprints/patterns/MVC-detailed.html)



- MVC Test Pattern for testing the Model component



Frameworks for Unit Test

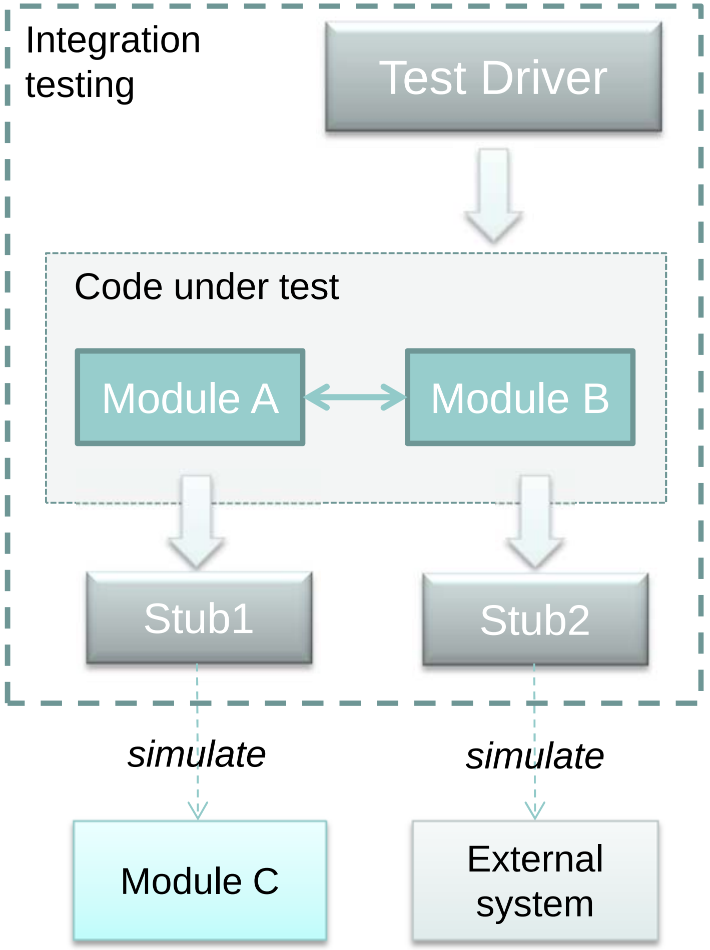


Outline

- Unit testing
  - JUnit framework for test automation
  - Use mock objects as test oracle
  - Tricks and Best Practice
- Integration testing
  - Test stub
  - Dependency injection
- Acceptance testing
  - FitNesse framework

Integration Testing

- Every step integrates a subset of modules
  - Unit testing has been conducted for each module.
  - The purpose is to test the interaction between these modules.
  - Other modules are ignored, even if their implementation is available.
- Main issues in integration testing:
  - Integration order
    - top-down, bottom-up, sandwich
    - continuous testing
  - Test driver
    - JUnit, customized scripts
  - Test stubs

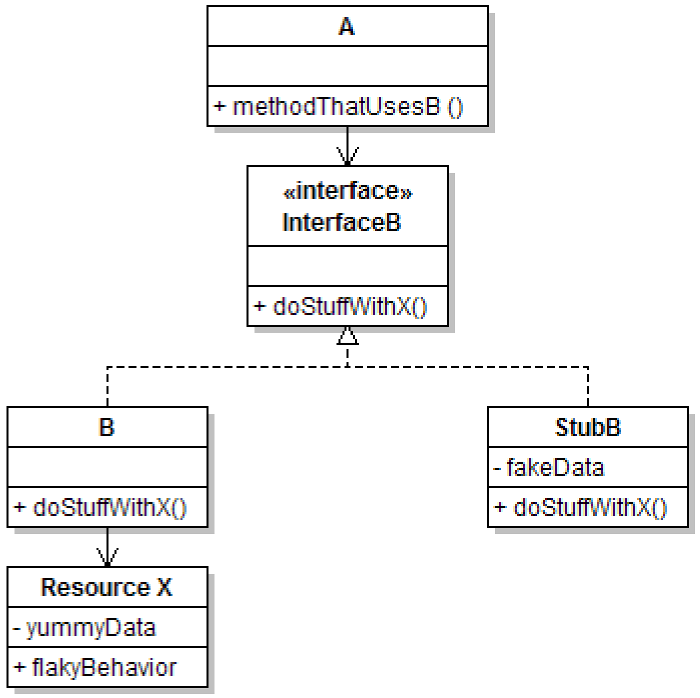


Test Stub

- **Stub**: A controllable replacement for an existing module or external system to which your code under test has a dependency.
- Why replacing existing modules with test stubs:
  - The module is half-baked.
  - The module is full of bugs.
  - The module is not considered in the testing.
- Why replacing external systems with test stubs:
  - Side-effect: hardware devices, file system
  - Tedious to setup: database
  - Slow and unreliable: network / internet
  - Difficult-to-control: time/date- sensitive code, threads

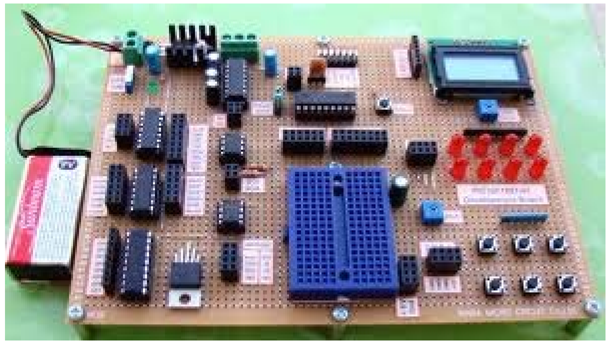
Wiring Stub with Code Under Test

- The dependency of a module on another module or external system is abstracted into an *interface*.
  - Create a "stub" class to implement this interface to simulate what the actual module or external system does.



Question: How to ***wire*** A with StubB instead of B in a way oblivious to A?

Answer: use *dependency injection*

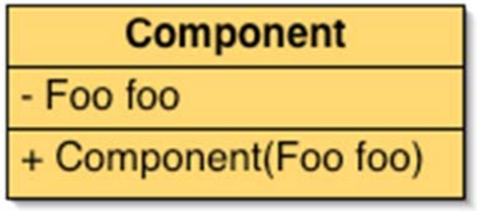


Dependency Injection

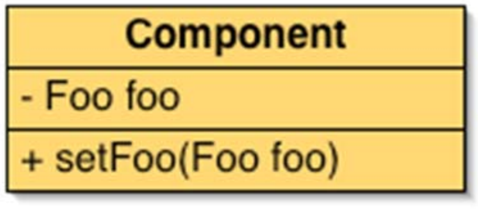
- What is dependency injection?
  - A family of techniques to reduce coupling by moving configuration and dependency wiring outside a module.
    - The module *does not control* which objects it is wired to through the interface (**inversion of control**).
    - Instead, the dependency is “injected” into the module by the framework or test driver.
  - Not merely for testing purpose. Also used for building flexible software.
    - Widely used in *container framework* (e.g. Spring, pico, Unity)

Types of Dependency Injection

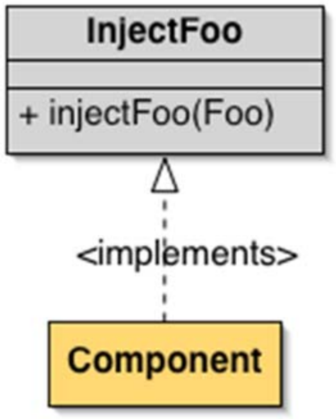
Constructor Injection: Declare dependencies completely in one or more constructors



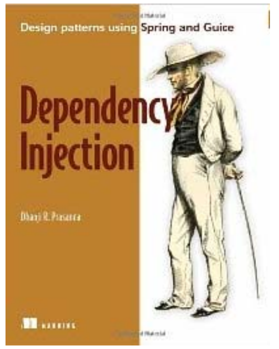
Setter injection: all dependencies declared using setter methods

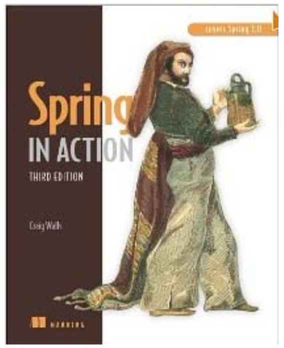


Interface Injection: Define interface used to inject a dependency



Read more on Dependency Injection and Inversion of Control:





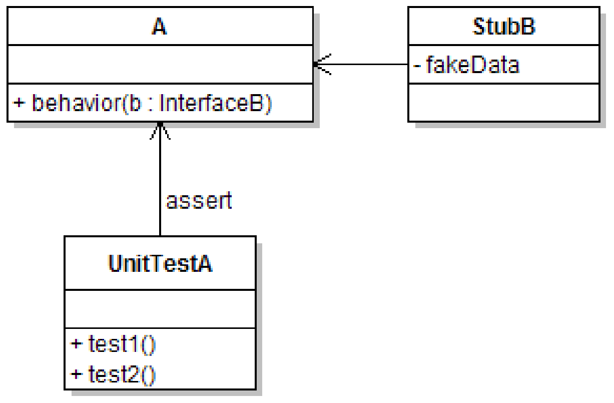
Developing Test Stubs

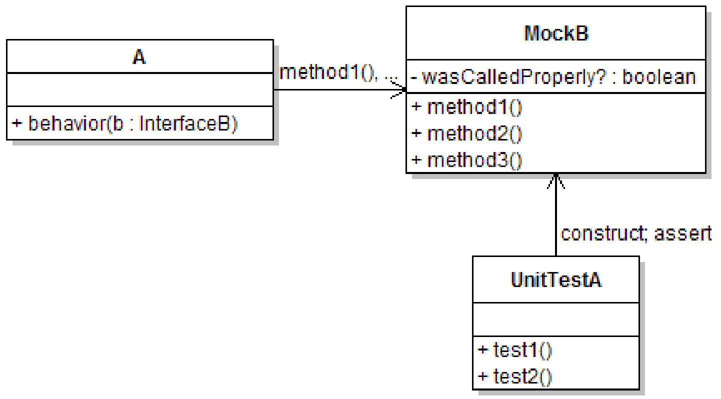
- Any tool or framework support?
  - Unfortunately no.
  - Different projects have different need. There is no general way to derive test stubs.

- *It might be a good idea to ask every module author to provide the stubs of her/his module.*
  - The author knows the most about functionality of the module.
  - The author knows the most about implementation of the module.
  - Test stubs can be used in unit testing of this module as well.

Stubs vs. Mocks

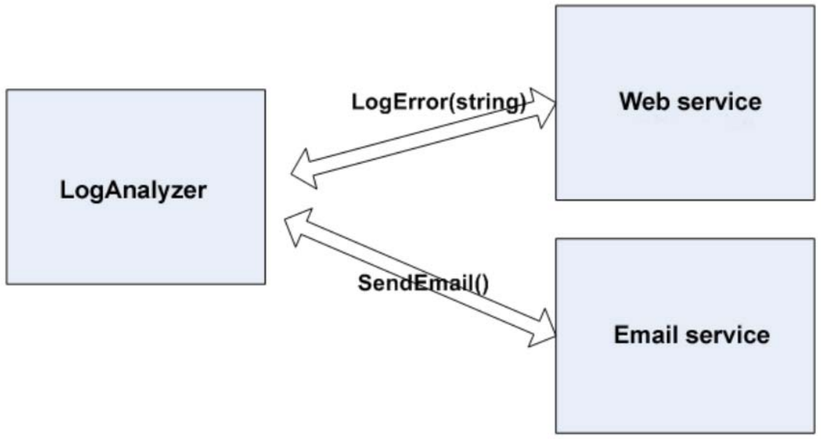
- **Similarities**:
  - Both are used in unit and integration testing.
  - Both simulate the behavior of other code through interfaces
- A **stub** provides an replacement for a module or external system.
  - Stubs contains (fake) application logic implemented by the developers.
  - **Stubs have nothing to do with test oracle**. Developers need to create test oracle to verify the behavior of the class under test.
- A **mock** is trained to assert interaction between class under test and the mocked interface.
  - Mocks do not contain any application logic.
  - **Mocks are test oracle**. Test cases call mocks to verify the behavior of the class under test.

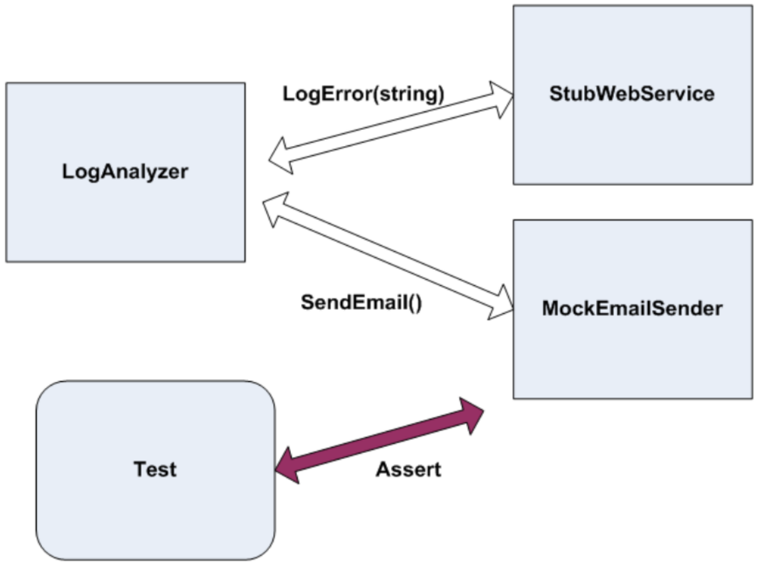




Using Stubs/Mocks Together

- Suppose a log analyzer reads from a web service.
  If the web fails to log an error, the analyzer must send email.
  - How to test to ensure that this behavior is occurring?
  - **Web service** and **Email service**: Which is mock? Which is stub? Why?
- Set up a *stub* for the web service that intentionally fails.
- Set up a *mock* for the email service that checks to see whether the analyzer contacts it to send an email message.





Outline

- Unit testing
  - JUnit framework
  - Mock object
- Integration testing
  - Test stub
  - Dependency injection
- Acceptance testing
  - FitNesse framework

Definition

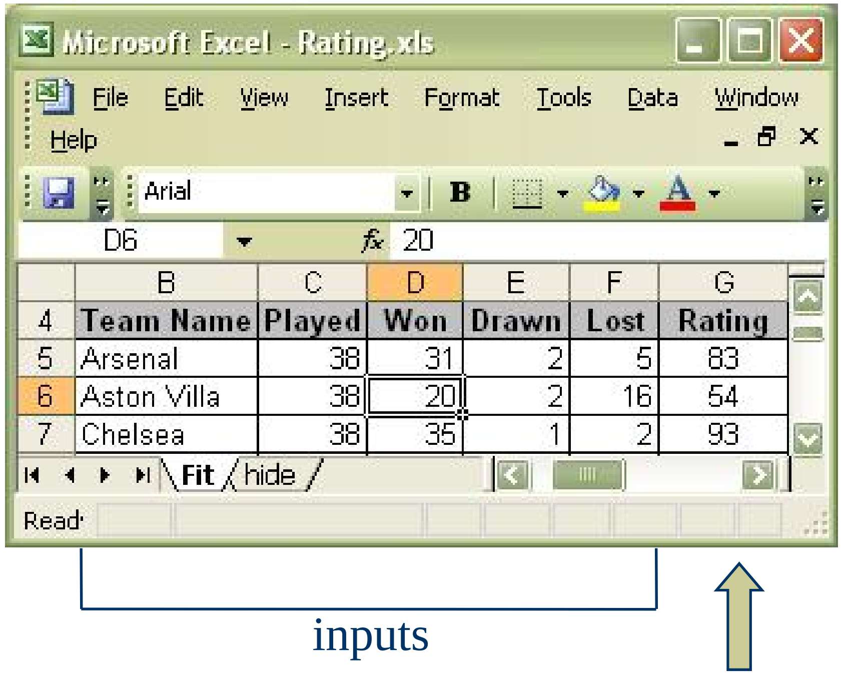
- Tests owned and defined by **the customer** to verify that the application is complete and correct with respect to their actual requirement.

- Manual approach to acceptance testing:
  - User exercises the system manually using his experience.
  - Disadvantages: expensive, error prone, not repeatable

Table-based Approach

- Starting from a user story (use case, textual requirement), the customers define a **table** (spreadsheet, html, Word, …) to describe the expectations of the program's behavior.

- At this point tables can be used as test cases. The customer can manually insert inputs in the application and compare outputs with expected results.
  - Or they can automate the execution of test cases with an acceptance test framework.



	B	C	D	E	F	G
4	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

Example

Fit

- The **Framework for Integrated Test (Fit)** is the most well-known table-based acceptance test automation framework.
- Fit lets customers and analysts write “executable” tests using simple HTML tables.
- Developers write “fixtures” to link the test cases with the actual system itself.
- Fit compares these test cases, written using HTML tables, with actual values, returned by the system, and highlights the results with colours and annotations.

Using Fit

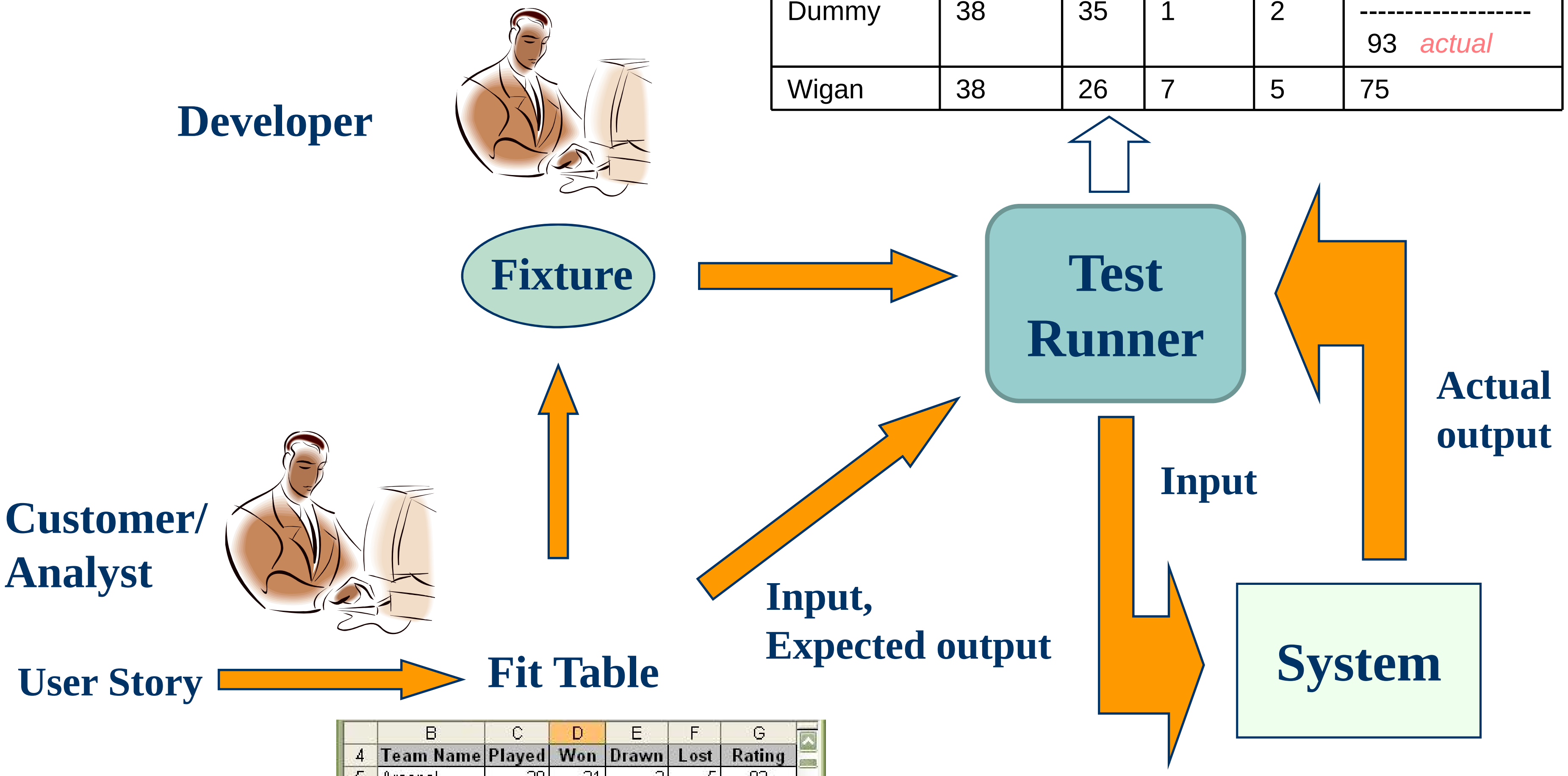
- Two major steps to automate test execution using Fit:
  - Express a test case in the form of a **Fit table.**
  - Write the glue code, called a **Fixture**, that bridges the test case and system under test.
- Fit table
  - A Fit table is a way of expressing the business logic using a simple HTML table.
  - Fit tables help developers better understand the requirements and are used as acceptance test cases.
- Fixture
  - A Fixture is an interface between the Fit framework, Fit tables, and the system under test.
  - Fixtures are **procedures/functions/classes** usually written by developers.
  - In general, there is a one-to-one mapping between a Fit table and fixture.

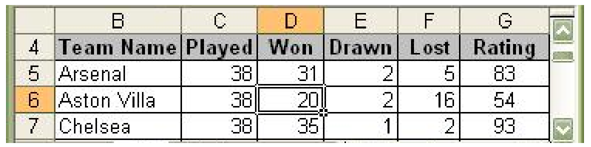
The Fit process

Output Table

sample.VerifyRating

team name	played	won	drawn	lost	rating()
Arsenal	38	31	2	5	83
Aston Villa	38	20	2	16	54
Chelsea	38	35	1	2	93
Dummy	38	35	1	2	100 expected ------------------- 93 *actual*
Wigan	38	26	7	5	75

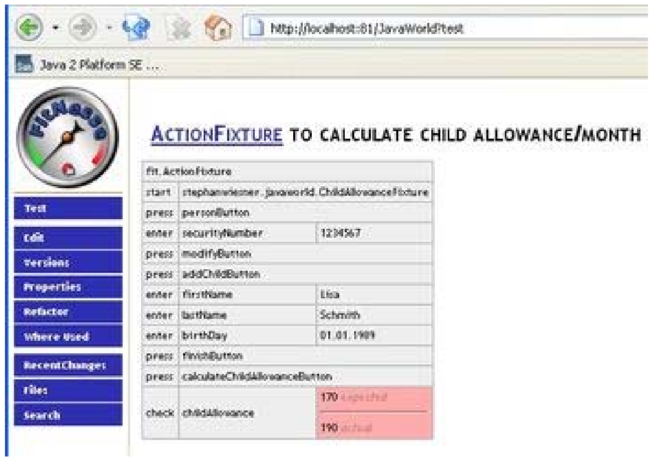




	B	C	D	E	F	G
4	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

Fit Tools family

- **Fit -** http://fit.c2.com/wiki.cgi?DownloadNow
  - Java, C++, …
- **FitRunner** - http://fitrunner.sourceforge.net/
  - Eclipse plug-in
- **FitNesse** - http://www.fitnesse.org
  - Java, C#, C++
- **Other Fit tools**
  - .Net, Python, Perl, SmallTalk, C++, Ruby, Lisp, Delphi, etc.



ACTIONFIXTURE TO CALCULATE CHILD ALLOWANCE/MONTH

fit.ActionFixture		
start	stephanwiesner.javaworld.ChildAllowanceFixture	
press	personButton	
enter	securityNumber	1234567
press	modifyButton	
press	addChildButton	
enter	firstName	Lisa
enter	lastName	Schmith
enter	birthDay	01.01.1989
press	finishButton	
press	calculateChildAllowanceButton	
check	childAllowance	170 expected 190 actual

Case Study: Football Website

- A sports magazine decides to **add a new feature** to its Website that will allow users *to view top football teams based on their ratings.*

- An analyst and a developer get together to discuss the requirements.

- The outcome of the discussion is:
  - a user story card that summarizes the requirements
  - a set of acceptance tests
  - an excel file with sample data

A user can search for top N football teams based on rating.

Note:
rating = ((10000*(won*3+drawn))/(3*played))/100
round the resulut

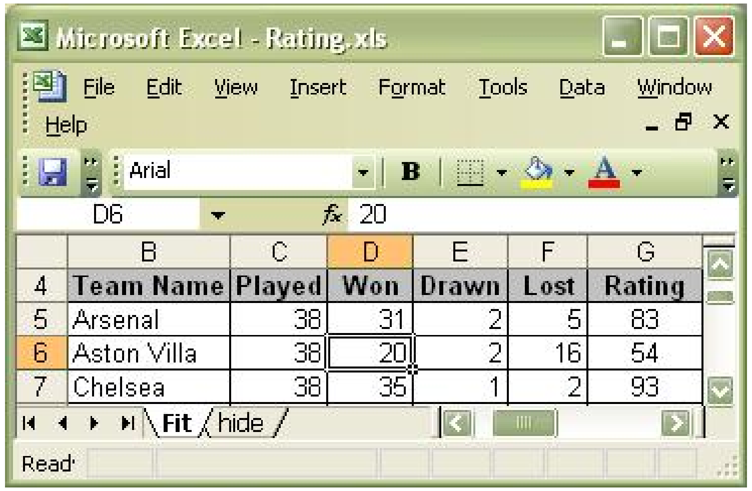
user story (requirement)

set of tests

Test 1: Verify the rating is calculated properly.

Test 2: Search for top 2 teams using the screen and validate the search results.

Note: The time taken to search should be less than 2 sec.



Microsoft Excel - Rating.xls

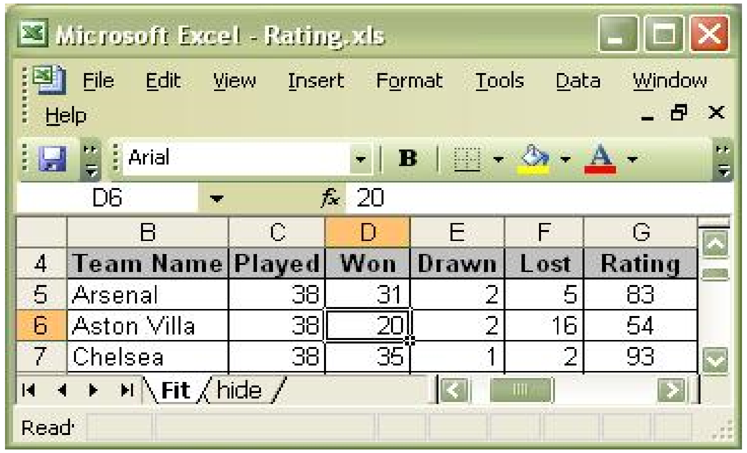
D6 fx 20

	B	C	D	E	F	G
4	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

excel file with sample data

Test 1: Writing FitTable

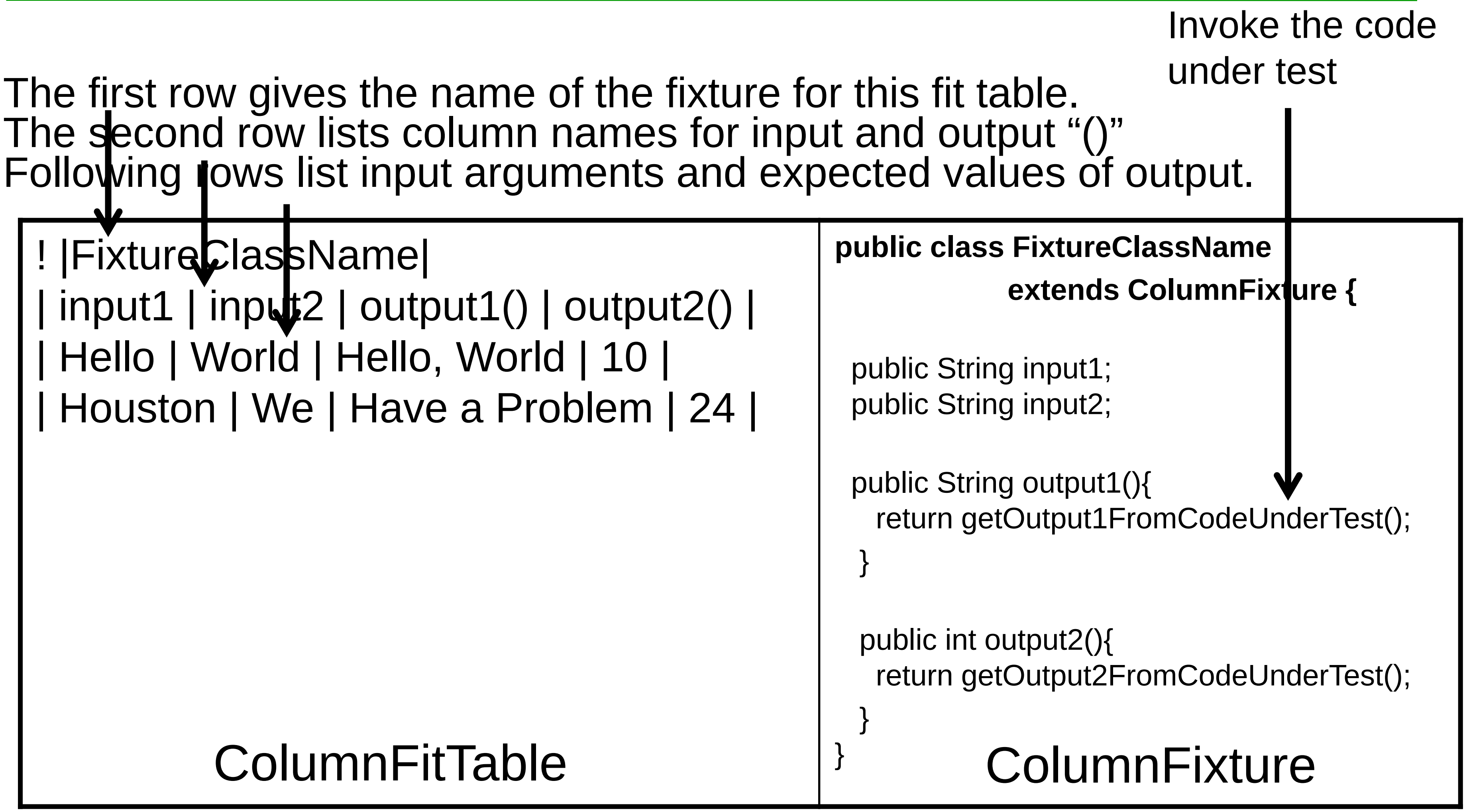
- Verify the rating is calculated properly
  - The first step is to convert the excel table to a Fit table.
    - In this case, we will use a **ColumnFitTable**
  - The next step is to develop a Fixture for the table.
    - In this case, we will develop a **ColumnFixture**.


Microsoft Excel - Rating.xls
File Edit View Insert Format Tools Data Window Help
Arial
D6
20
Fit
hide
Read

	B	C	D	E	F	G
4	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

```
! | VerifyRating |
| Team Name| Played | Won | Drawn | Lost | rating() |
| Arsenal | 38 | 31 | 2| 5 | 83|
....
```

ColumnFixture



Invoke the code under test

The first row gives the name of the fixture for this fit table.
The second row lists column names for input and output "()"
Following rows list input arguments and expected values of output.

```
! |FixtureClassName|
| input1 | input2 | output1() | output2() |
| Hello | World | Hello, World | 10 |
| Houston | We | Have a Problem | 24 |
```

ColumnFitTable

```
public class FixtureClassName
              extends ColumnFixture {

  public String input1;
  public String input2;

  public String output1(){
    return getOutput1FromCodeUnderTest();
  }

   public int output2(){
    return getOutput2FromCodeUnderTest();
  }
}
```

ColumnFixture

The fixture class extend **fit.ColumnFixture** and declare public fields and methods to match the Fit table

Test 1: Writing Fixture

- For each input attribute represented by Columns 1 through 5 in the second row of the Fit table, there is a public member with the same name.
- A public method `rating()` corresponds to the calculation in the sixth column.
- The rating() method in VerifyRating is where the bridging between the test case and the system under test happens.

```
! | VerifyRating |
| Team Name| Played | Won | Drawn | Lost | rating() |
| Arsenal | 38 | 31 | 2| 5 | 83|
....
```

```java
public class VerifyRating
   extends ColumnFixture {

  public String teamName;
  public int played;
  public int won;
  public int drawn;
  public int lost;

  public long rating() {
    Team team = new Team(teamName,
          played,won,drawn,lost);
     return team.rating;
  }
}
```

Test 1: Execute the Test Cases

Here is what happens when you run the test:

1. Fit parses the table and for each row in the table Fit set the values specified in Columns 1 through 5 to the corresponding fields in the fixture.
2. The rating() method is executed to get the **actual value** to be compared against the **expected value** specified in the sixth column.
3. If the expected value matches the actual value, then the test passes; otherwise it fails.

sport.fixtures.VerifyRating					
team name	**played**	**won**	**drawn**	**lost**	**rating()**
Arsenal	38	31	2	5	83
Aston Villa	38	20	2	16	54
Chelsea	38	35	1	2	93
Dummy	38	35	1	2	100 expected ------------------- 93 *actual*
Wigan	38	26	7	5	75

🟩	**passed**
🟥	**failed**
🟨	**exception**

RowFixture

Compare the expected list (FitNesse table) with the actual result (from fixture code) and report any additional or missing items.

- The second row describes the properties or methods that you want to verify
- All rows after that describe expected objects in the array

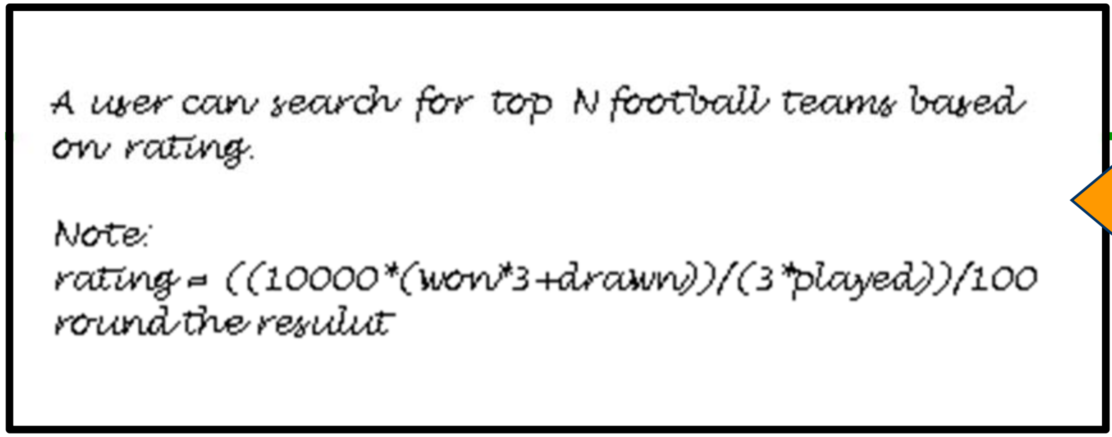
```
!|RowFixtureTest|
|name|post_code|
|John Smith|SW4 66Z|
|Michael Jordan|NE1 8AT|
```

```java
import info.fitnesse.fixturegallery.domain.Player;
import fit.RowFixture;

public class RowFixtureTest extends RowFixture{

   public Class getTargetClass() {
           return Player.class;
   }
   public Object[] query() throws Exception {
           return Player.players.toArray();
   }
}
```

Override:

- getTargetClass: returns the Type or Class object representing the type of objects in the array.

- query: returns the actual array of objects to be verified.

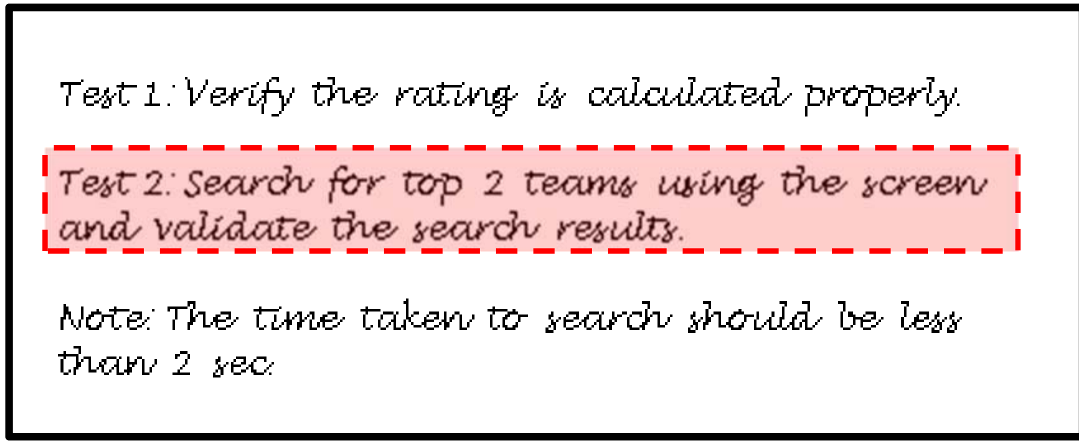


A user can search for top N football teams based on rating.

Note:
rating = ((10000*(won*3+drawn))/(3*played))/100
round the resulut

user story (requirement)

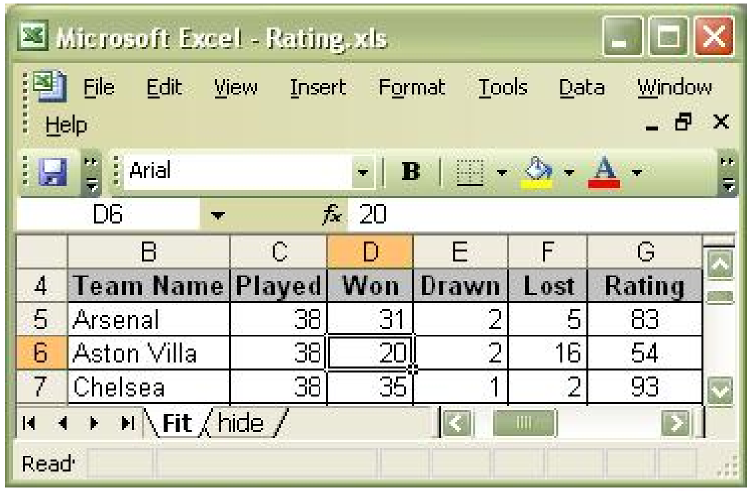
set of tests



Test 1: Verify the rating is calculated properly.

Test 2: Search for top 2 teams using the screen and validate the search results.

Note: The time taken to search should be less than 2 sec.

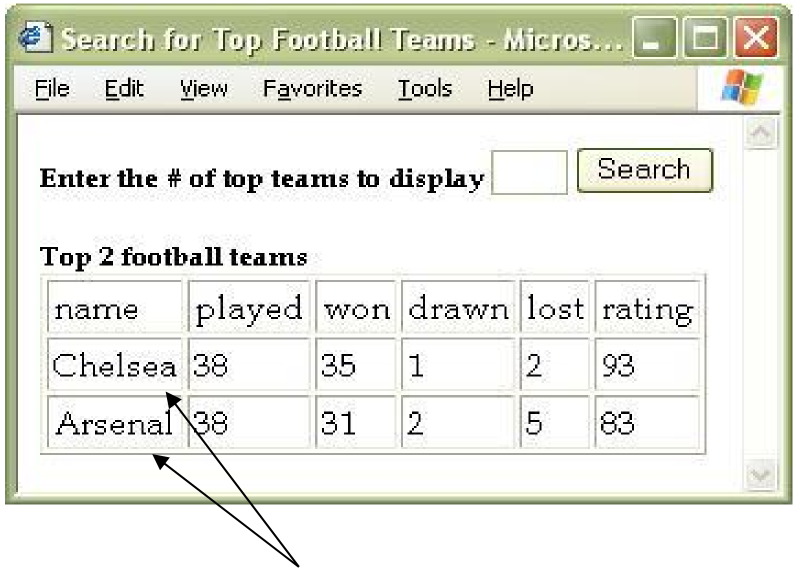

Microsoft Excel - Rating.xls
File Edit View Insert Format Tools Data Window Help
Arial
B
D6
20
Fit
hide
Read

4	Team Name	Played	Won	Drawn	Lost	Rating
5	Arsenal	38	31	2	5	83
6	Aston Villa	38	20	2	16	54
7	Chelsea	38	35	1	2	93

excel file with sample data

Test 2: Search for Top Two Teams

- This test involves a screen (Web page) through which the user:
  - provides input (*types 2 in the number of top teams text box*)
  - clicks on a button (*clicks on the search button*)
  - seeks verification that the results returned match a collection of objects as expected. (*expects to see the top two teams displayed*)



Enter the # of top teams to display

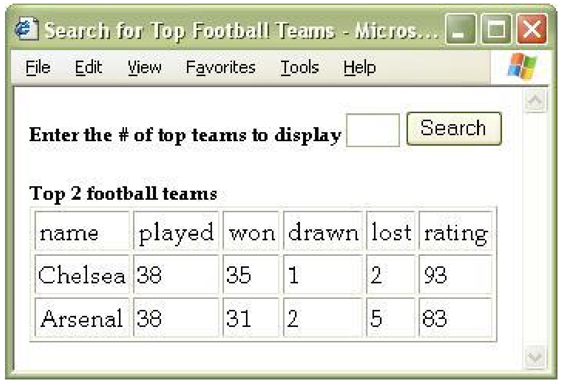
Top 2 football teams

name	played	won	drawn	lost	rating
Chelsea	38	35	1	2	93
Arsenal	38	31	2	5	83

"Top 2 teams based on rating"

Test 2: Writing ActionFitTable

- Write a **ActionFitTable** to describe this requirement.
  - Use **actions** to specify these steps.
  - Similar to keyword-driven test automation, but actions are more abstract than keywords.


Search for Top Football Teams - Micros...
File Edit View Favorites Tools Help
Search

Enter the # of top teams to display

Top 2 football teams

name	played	won	drawn	lost	rating
Chelsea	38	35	1	2	93
Arsenal	38	31	2	5	83

fit.ActionFixture		
start	VerifyRating2	
enter	number of top teams	2
press	search	
check	number of results	2

- **ActionFitTable** supports four types of actions (“commands”) :
  - **Start:** establish the precondition.
  - **Enter:** input data into the interface.
  - **Press:** trigger the concerned functionality.
  - **Check**: examine the results.

ActionFixture

All rows after the first begin with a command cell, followed by command arguments in the remaining cells.

Possible commands are:

start — it starts the test by taking the class name for the actual fixture to automate
enter — it executes a method and passes an argument to it.
press — it executes a void method without testing anything.
check — it executes a method and verifies its value.

```
!|ActionFixture|
| start | ActionFixtureTest|
| enter | firstPart | Hello |
| enter | secondPart | World |
| press | join |
| check | together |Hello, World|
```

ActionFitTable

```
public class ActionFixtureTest extends fit.Fixture{
  private String first, second, both;

  public void firstPart(String s){
      first=s;
  }
  public void secondPart(String s){
      second=s;
  }
  public void join(){
      both=first+ ", "+second;
  }
  public String together(){
      return both;
  }
}
```

ActionFixture

Test 2: Writing ActionFixture

fit.ActionFixture		
start	VerifyRating2	
enter	number of top teams	2
press	search	
check	number of results	2

fit.ActionFixture		
start	VerifyRating2	
enter	number of top teams	-1
press	search	
check	number of results	0

```
public class VerifyRating2 extends fit.Fixture{

   public void numberOfTopTeams(int n){
        Input the number of top teams to the application
   }

   private Collection<Team> results;
   public void search() {
        results = the output from the application
   }

   public int numberOfResults(){ return results.size(); }
}
```

Thank you!