2.1 Coding and Unit Testing

Pankaj Jalote
Juan Rodríguez Hortalá

Bibliography

- Jalote, P. A Concise Introduction to Software Engineering, Chapter 7: Coding and Unit Testing. Springer, 2008.

Coding

- Goal is to implement the design in best possible manner
- Coding affects testing and maintenance
- As testing and maintenance costs are high, aim of coding activity should be to write code that reduces them
- Time spent in coding is relatively small (40-20-40 rule)
- Hence, goal should not be to reduce coding cost, but testing and maint cost, i.e. make the job of tester and maintainer easier

Coding…

- Code is read a lot more than it is written
  - Coders themselves read the code many times for debugging, extending etc
  - Maintainers spend a lot of effort reading and understanding code
  - Other developers read code when they add to existing code
- Hence, code should be written so it is easy to understand and read, not easy to write!

Coding…

- Having clear goal for coding will help achieve them
- Weinberg experiment showed that coders achieve the goal they set
  - Different coders were given the same problem
  - But different objectives were given to different programmers – minimize effort, min size, min memory, maximize clarity, max output clarity
  - Final programs for different programmers generally satisfied the criteria given to them

Weinberg experiment..

	Resulting Rank (1=best)				
	o1	o2	o3	o4	o5
Minimize Effort (o1)	1	4	4	5	3
Minimize prog size (o2)	2-3	1	2	3	5
Minimize memory (o3)	5	2	1	4	4
Maximize code clarity (o4)	4	3	3	2	2
Maximize output clarity (o5)	2-3	5	5	1	1

1. Programming Principles

- The main goal of the programmer is write simple and easy to read programs with few bugs in it
- Of course, the programmer has to develop it quickly to keep productivity high
- There are various programming principles that can help write code that is easier to understand (and test…)

Information Hiding

- Software solutions always contain data structures (DS) that hold information
- Programs work on these DS to perform the functions they want
- In general only some operations are performed on the information, i.e. the data is manipulated in a few ways only
- E.g. on a bank’s ledger, only debit, credit, check current balance etc are done

Information Hiding…

- Information hiding – the information should be hidden; only operations on it should be exposed
- I.e. data structures are hidden behind the access functions, which can be used by programs
- Info hiding reduces coupling
- This practice is a key foundation of OO and components, and is also widely used today

Some Programming Practices

- Control constructs: Use only a few control constructs (rather than using a large number of constructs)
- Goto: Use them sparingly, and only when the alternatives are worse
- Info hiding: Use info hiding
- Use-defined types: use these to make the programs easier to read

Some Programming Practices..

- Nesting: Avoid heavy nesting of if-then-else; if disjoint nesting can be avoided
- Module size: Should not be too large – generally means low cohesion
- Module interface: make it simple
- Robustness: Handle exceptional situations
- Side effects: Avoid them, document

Some Programming Practices..

- Empty catch block: always have some default action rather than empty
- Empty if, while: bad practice
- Read return: should be checked for robustness
- Return from finally: should not return from finally
- Assumptions about parameters: should check for compatibility

Coding Standards

- Programmers spend more time reading code than writing code
- They read their own code as well as other programmers code
- Readability is enhanced if some coding conventions are followed by all
- Coding standards provide these guidelines for programmers
- Generally are regarding naming, file organization, statements/declarations, …
- Some Java conventions discussed here

  http://www.oracle.com/technetwork/java/codeconv-138413.html

Coding Standards…

- Naming conventions
  - Package name should be in lower case (mypackage, edu.iitk.maths)
  - Type names should be nouns and start with uppercase (Day, DateOfBirth,…)
  - Var names should be nouns in lowercase; vars with large scope should have long names; loop iterators should be i, j, k…
  - Const names should be all caps
  - Method names should be verbs starting with lower case (eg getValue())
  - Prefix is should be used for boolean methods

Coding Standards…

- Files
  - Source files should have .java extension
  - Each file should contain one outer class and the name should be same as file
  - Line length should be less than 80; if longer continue on another line…

Coding Standards…

- Statements
  - Vars should be initialized where declared in the smallest possible scope
  - Declare related vars together; unrelated vars should be declared separately
  - Class vars should never be declared public
  - Loop vars should be initialized just before the loop
  - Avoid using break and continue in loops
  - Avoid executable stmts in conditionals
  - Avoid using the do… while construct

Coding Standards…

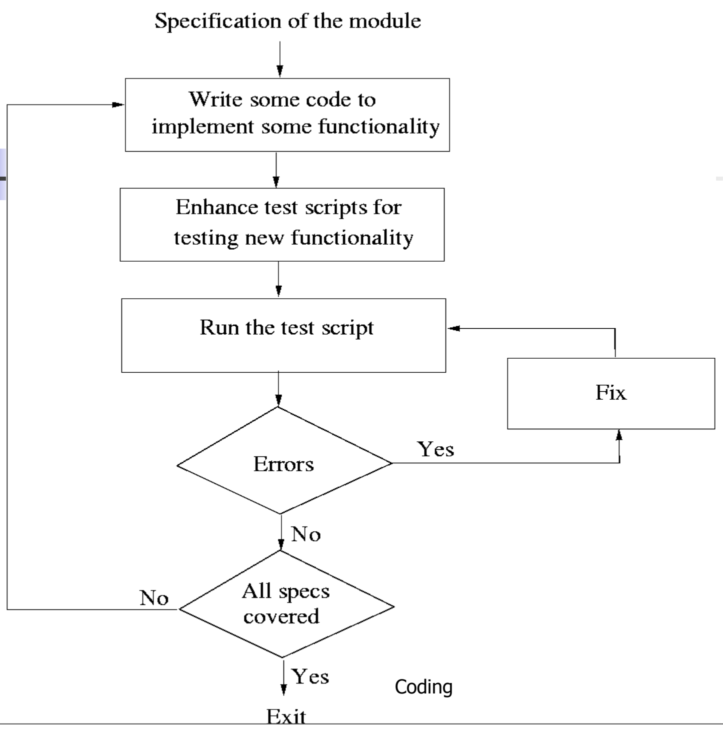
- Commenting and layout
  - Single line comments for a block should be aligned with the code block
  - There should be comments for all major vars explaining what they represent
  - A comment block should start with a line with just /* and end with a line with */
  - Trailing comments after stmts should be short and on the same line

2. Incrementally Developing Code

- Coding starts when specifications for modules from design is available
- Usually modules are assigned to programmers for coding
- In top-down development, top level modules are developed first; in bottom-up lower levels modules
- For coding, developers use different processes; we discuss some here

An Incremental Coding Process

- Basic process: Write code for the module, unit test it, fix the bugs
- It is better to do this incrementally – write code for part of functionality, then test it and fix it, then proceed
- I.e. code is built code for a module incrementally


Specification of the module
Write some code to implement some functionality
Enhance test scripts for testing new functionality
Run the test script
Fix
Errors
Yes
No
All specs covered
No
Yes
Exit

Test Driven Development

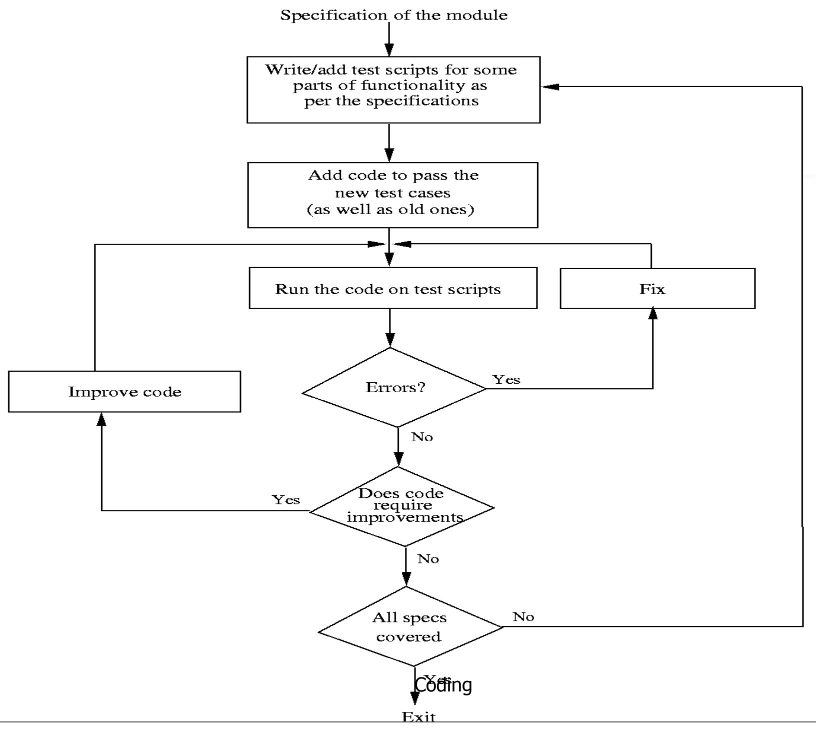
- This coding process changes the order of activities in coding
- In TDD, programmer first writes the test scripts and then writes the code to pass the test cases in the script
- This is done incrementally
- Is a relatively new approach, and is a part of the extreme programming (XP)

TDD…

- In TDD, you write just enough code to pass the test
- I.e. code is always in sync with the tests and gets tested by the test cases
  - Not true in code first approach, as test cases may only test part of functionality
- Responsibility to ensure that all functionality is there is on test case design, not coding
- Help ensure that all code is testable

TDD…

- Focus shifts to how code will be used as test cases are written first
  - Helps validate user interfaces specified in the design
  - Focuses on usage of code
- Functionality prioritization happens naturally
- Has possibility that special cases for which test cases are not possible get left out
- Code improvement through refactoring will be needed to avoid getting a messy code


Specification of the module
Write/add test scripts for some parts of functionality as per the specifications
Add code to pass the new test cases (as well as old ones)
Run the code on test scripts
Fix
Errors?
Yes
No
Improve code
Does code require improvements
Yes
No
All specs covered
No
Yes
Exit

Pair Programming

- Also a coding process that has been proposed as key practice in XP
- Code is written by pair of programmers rather than individuals
  - The pair together design algorithms, data structures, strategies, etc.
  - One person types the code, the other actively reviews what is being typed
  - Errors are pointed out and together solutions are formulated
  - Roles are reversed periodically

Pair Programming…

- PP has continuous code review, and reviews are known to be effective
- Better designs of algorithms/data structures/logic/ …
- Special conditions are likely to be dealt with better and not forgotten
- It may, however, result in loss of productivity
- Ownership and accountability issues are also there
- Effectiveness is not yet fully known

3. Managing Evolving Code

- During coding process, code written by a programmer evolves
- Code by different programmers have to be put together to form the system
- Besides normal code changes, requirement changes also cause changes
- Evolving code has to be managed

Source Code Control

- Source code control is an essential step programmers have to do
- Generally tools like SVN, CVS, VSS are used

Refactoring

- As code evolves, the design becomes more complex
- Refactoring is a technique to improve existing code by improving its design (i.e. the internal structure)
- In TDD, refactoring is a key step
- Refactoring is done generally to reuce coupling or increase cohesion

Refactoring...

- Refactorización: cambios realizados sobre la estructura interna del SW para hacer que sea más fácil de entender y más barato de modificar pero que no cambian su comportamiento observable
- Es decir, proceso de modificación del código en el que el código no se extiende con funcionalidades ni tampoco las pierde, pero su diseño sí que se mejora

Refactoring...

- Cohesión: se refiere a que las piezas de código agrupadas en una misma unidad tengan relación entre sí.
- Es mejor hacer agrupar el código en módulos o clases pequeñas y cohesionadas pq:

  - al trabajar en unidades de código más pequeñas nos podemos centrar en lo que hacen esas unidades de código. Si en un módulo tenemos varias partes que hacen cosas distintas es posible que el código que se dedica a una funcionalidad esté disperso por el módulo

  - así es más fácil distribuir el trabajo y que la gente trabaje en paralelo en funcionalidades distintas

  - es más fácil diseñar los tests unitarios para el módulo

Refactoring...

- Acoplamiento (coupling): grado de dependencia entre varias unidades de SW de un sistema informático.
- Es deseable que el acoplamiento sea lo más bajo posible para que las modificaciones en un trozo del SW afecten lo mínimo a otros trozos =>
  - mantenibilidad mejorada: tocar un trozo de código tiene potencialmente menos consecuencias en el resto del sistema
  - reusabilidad mejorada: las piezas de código son más independientes entre sí
  - evita que los defectos de una unidad de código se propaguen a otras

Refactoring…

- Involves changing code to improve some design property
- No new functionality is added
- To mitigate risks associated with refactoring (loss of functionality) two golden rules
  - Refactor in small steps
  - Have test scripts available to test that the functionality is preserved

Refactoring…

- With refactoring code is continually improving: design, rather than decaying with time, evolves and improves with time
- Refactoring cost is paid by reduced maint effort later
- There are various refactoring patterns that have been proposed
- A catalog of refactorings and how to do them is available online http://refactoring.com/catalog

Refactoring…

- If refactoring is envisaged, the design stage can be simplified: come up with a good and simple design vs. trying to make a complex design, flexible enough to accomodate all types of future changes that can be anticipated
- Refactoring is not a technique for bug fixing: it starts from healthy code and ensures it remains healthy -> make code live longer

Refactoring…

- "Bad smells" that suggest that refactoring may be desired
  - Duplicate code
  - Long method
  - Long class
  - Long parameter list
  - Speculative generality
  - Too much communication between objects
  - Message chaining
- See also http://refactoring.com/catalog

4. Unit Testing and Inspection

- Code has to be verified before it can be used by others
- Here we discuss only verification of code written by a programmer (system verification is discussed in testing)
- There are many different techniques – two most commonly used are unit testing and inspection
- We will discuss these here

Unit Testing

- Is testing, except the focus is the module a programmer has written
- Unit testing = a method by which individual units of source code are tested to determine if they are fit for use.
- A unit is the smallest testeable part of an application
- Most often UT is done by the programmer himself
- UT will require test cases for the module – will discuss in testing
- When used with TDD, test cases specify the intended behaviour of the unit
- Ideally each test case is independent from the others
- A unit test should not change the code of the unit being tested

Unit Testing…

- UT also requires drivers: modules that are responsible of getting the test data, executing the unit with the test data, and then reporting the results
- Besides the driver and test cases, tester needs to know the correct outcome as well
- Unit testing can be implemented manually or be automated
- If incremental coding or TDD is being done, then complete UT needs to be automated, and its execution and check must be very fast. Otherwise, repeatedly doing UT will not be possible: test are going to be executed hundreds of times during development!
- There are tools available to help: the xUnit family (JUnit, CppUnit, NUnit ...)
  - They provide the drivers
  - Test cases are programmed, with outcomes being checked in them
  - I.e. UT is a script that returns pass/fail

Unit testing of Classes

- In OO languages the unit to be tested is usually a class
- To test a class, the programmer needs to create an object of that class, take the object to a particular state (by setting its attributes), invoke a method on it, and then check whether the result is as expected

Unit Testing: JUnit

- Junit is a frameworks that can be used for testing Java classes
- Each test case is a method which ends with some assertions = conditions needed to pass the test
- If assertions hold, the test case pass, otherwise it fails
- Complete execution and evaluation of the test cases is automated
- For enhancing the test script, additional test cases can be added easily

Unit Testing: stubs and mocks

- Method Stub: piece of code used to stand in for some other programming functionality
  - a stub usually doesn't actually do anything other than declare itself and the parameters it accepts, and returns a value expected by the caller: just enough code to allow it to be compiled and linked with the rest of the program
  - may simulate the behavior of existing code (e.g. a procedure on a remote machine) or be a temporary substitute for code not developed yet

Example:

```
Date getCurrentDate() {
     return new Date(10000);
}
```

Unit Testing: stubs and mocks

- Mock objects (simulacros, maquetas): simulated objects that mimic the behaviour of real (non-mock) objects in controlled ways. OO version of a stub
- Have the same interface as the real objects they mimic: client unit is unaware of whether it is talking to a real object or to a mock object
- Useful in UT when a real object is impractical or impossible to incorporate to the unit test, because:
  - the code for the object has not been developed yet or we don't trust the available code
  - is slow: e.g. a remote database connection
  - has states difficult to reproduce (e.g. network error)
  - supplies non-deterministic results (e.g. current time or temperature)
  - we need to include attributes and methods exclusively for testing purposes

Unit Testing: mocks

- we need to include attributes and methods exclusively for testing purposes, e.g.
  - adding an additional public attribute (not present in the real object) for the UT to access its internal state
  - the mock may write debug information to a public log file
- Mock objects are used to interact with the real object we are testing in a UT
- Advantages of mock objects
  - faster and easier tests: no waiting for network connections or events to occur
  - separation of concerns
- Disadvantages: any mock object must accurately model the behavior of the object they are mocking, otherwise any UT that uses it would be inaccurate. This can be difficult if the specifications of the real object are not clear

Unit Testing: advantages

- UT isolate each part of the program and show that the individual parts are correct. Mock objects help to make each unit test independent from the others
- A unit test provides a written contract that a unit of code must satisfy, reflecting the intended use of the code. In TDD the UT may take place of formal design
- Therefore UT helps in refactoring code, as test can be used to check that the behavior of each unit has not changed
- In UT, tests are a sort of living documentation of the system, which is less susceptible to become outdated, at least if TDD is followed with discipline
- Simplifies integration of code, as it increases confidence in the units

Unit Testing: advantages

- Forces a separation of interface from implementation: usually a class A depends on another class B, so testing class A leads to testing class B. If we are not confident about B, or if B performs networks connection, is nondeterministic ... then it is better to create a mock object for B, to avoid the test for A to depend on the behaviour of B: later if the test fails then we will be sure that the problem is on class A. That forces us to carefully define the interface of class B
- The goal in unit testing is to isolate a unit and validate its correctness. To increase isolation, in automated UT the unit is executed outside its natural environment. That, together with the previous item, results in a decrease of coupling and an increase of cohesion

Unit Testing: limitations

- Like any other form of SW testing, UT can only show the presence of errors, but cannot show the abstence of errors. In other words, we cannot expect to catch every error in the program
- Unit test themselves may have bugs
- To obtain the benefits of unit testing, a rigorous discipline has to be followed
  - ensure that test case faliures are reviewed dialy and addressed inmmediately
  - otherwise the application may evolve out of sync with the unit test suite, increasing false positives and reducing the effectiveness of the suite

Unit Testing: application to TDD

- Automated UT is the basis for TDD and XP (extreme programming)
  - A developer writes a unit test to expose either a SW requirement or a defect. Therefore the new test will fail in either case
  - Then the developer writes the simplest code to pass this test, along with all the other previous unit tests
- Most code in a system is unit tested, but not all paths through the code.
  - XP mandates "test everything that can possibly break" strategy, over the traditional "test every execution path" method.
  - Fewer tests are developed than in classical methods. XP recognizes that testing is rarely exhaustive (it would be too expensive) and provides guidance on how to effectively focus limited resources.
  - Test code is considered a first class project artifact that is maintained at the same quality as the implementation code. Developers release unit testing code to the code repository in conjunction with the code it test

Unit Testing: JUnit4 y Eclipse

- JUnit 4 está integrado en Eclipse 3 (Ganymede)
- Crear una nueva clase de prueba JUnit 4
  - New -> JUnit test case y seleccionamos New JUnit 4 test
  - Se puede elegir cualquier nombre para la clase de prueba
  - Al crear la primera clase de prueba Eclipse nos solicitará añadir el paquete de JUnit 4 al build path

Unit Testing: JUnit4 y Eclipse

- Una clase de prueba simple

```java
package is.testjuint4;
import org.junit.*;
import java.util.*;

public class SimpleTest {
    @Test
    public void testEmptyCollection() {
       Collection<Object> collection = new ArrayList<Object>();
       Assert.assertTrue("this collection should be empty",
collection.isEmpty());
    }
}
```

- Para ejecutar las pruebas de SimpleTest seleccionamos Run As -> JUnit Test

Unit Testing: JUnit4 y Eclipse

- JUnit 4 se basa en
  - un conjunto de anotaciones que podemos añadir a los métodos de las clases de prueba
  - aserciones en forma de métodos estáticos de la clase Assert
- Aserciones
  - Condiciones que deben cumplirse para pasar un método @Test
  - Aserciones típicas: assertTrue, assertFalse, assertEquals, assertNull, assertNotNull, assertSame, assertNotSame
  - El método fail de Assert también puede usarse para hacer fracasar un test inmediatamente
  - Listadas en http://kentbeck.github.com/junit/javadoc/latest/
  - Puede usarse el import estático

```java
import static org.junit.Assert.*;
```

  parar evitarse tener que cualificar cada aserción con "Assert."

Unit Testing: JUnit4 y Eclipse

- Las anotaciones básicas disponibles son
  - @Test: el método `public void` anotado es un test cuyas aserciones serán comprobadas al ejecutar la clase prueba.
    - Normalmente habrá varios métodos @Test en una clase prueba
    - Para ejecutar cada método @Test, JUnit primera crea una instancia fresca de la clase prueba y entonces ejecuta el método @Test. Como los tests son independientes no se garantiza que se ejecuten en ningún orden en particular.
    - Soporta dos parámetros opcionales
      - expected: el método @Test deberá lanzar una excepción para tener éxito, si lanza una excepción distinta o ninguna entonces el método falla. Ej: `@Test(expected=FileNotFoundException.class)`
      - timeout: el test falla si su ejecución lleva más tiempo que el especificado (en milisegundos). Ej: `@Test(timeout=100)`

Unit Testing: JUnit4 y Eclipse

- @Before y @After: útiles cuando varios métodos @Test trabajan sobre un conjunto común de objetos, que deben ser inicializados y finalizados antes y después de cada test.
  - Cada método `public void` anotado como @Before se ejecuta antes de cada ejecución de cada método @Test.
    - Si hay varios métodos @Before no se garantiza nada acerca de su orden de ejecución relativo
  - Cada método `public void` anotado como @After se ejecuta después de cada ejecución de cada método @Test.
    - Si hay varios métodos @After no se garantiza nada acerca de su orden de ejecución relativo
    - Los métodos @After siempre se ejecutan incluso aunque los correspondientes métodos @Before o el @Test hayan lanzado una excepción
    - Asignar a null objetos inicializados en métodos @Before ayuda al recolector de basura

Unit Testing: JUnit4 y Eclipse

```
import org.junit.*;
import java.util.*;
public class FixtureTest {
    private ArrayList<Object> arrayList;

    @Before
    public void setUp() {
        arrayList = new ArrayList<Object>();
    }
    @After
    public void tearDown() {
        arrayList = null;
    }
    @Test
    public void testEmptyArrayList() {
        Assert.assertTrue(arrayList.isEmpty());
    }
    @Test
    public void testOneItemArrayList() {
        arrayList("itemA");
        Assert.assertEquals(1, arrayList.size());
    }
```

- Posible ejecución: setUp(); testEmptyArrayList();tearDown(); setUp(); testOneItemArrayList();tearDown();

Unit Testing: JUnit4 y Eclipse

- En el código anterior podemos ver cómo la clase testeada (ArrayList en el ejemplo) se incluye como un atributo de la clase test
- Se llama test fixture al conjunto formado por los métodos @Before y @After y los atributos usados (suele incluir a la clase testeada) por varios métodos @Test
- Los métodos @Before y @After sirven por tanto para evitar tener que replicar el código de inicialización y finalización común a varios tests, o tener que meterlo en métodos privados que tienen que ser llamados por cada método @Test
- Como JUnit primera crea una instancia fresca de la clase prueba en la ejecución de cada test, entonces cada bloque "método @Before; método @Test; @método @After" se ejecutará en instancias diferentes de la clase de prueba, sin compartición de estado entre distintos bloques

Unit Testing: JUnit4 y Eclipse

- Las anotaciones @BeforeClass y @AfterClass se pueden aplicar a métodos `public static void` para ser ejecutadas antes y después, respectivamente, de todos los métodos @Test
- Son útiles para inicializaciones costosas de objetos comunes a varios tests, como por ejemplo la conexión a una base de datos o la apertura de ficheros

Unit Testing: JUnit4 y Eclipse

```
public class ClassFixture {
private ArrayList<Object> arrayList;
private static int setUpExec = 0;
private static int tearDownExec = 0;
 @BeforeClass // one-time initialization code
 public static void oneTimeSetUp() { System.out.println("oneTimeSetUp running ..."); }
 @AfterClass // one-time cleanup code
 public static void oneTimeTearDown() { System.out.println("oneTimeTearDown running ...");}
 @Before
 public void setUp() {
   arrayList = new ArrayList<Object>();
   setUpExec++; System.out.println("setUp running " + setUpExec + " times ...");
 }
 @After
 public void tearDown() {
    arrayList.clear();
    tearDownExec++; System.out.println("tearDown running " + tearDownExec + " times ...");
 }
 @Test
 public void testEmptyArrayList() { Assert.assertTrue(arrayList.isEmpty()); }
 @Test
 public void testOneItemArrayList() { arrayList.add("itemA"); Assert.assertEquals(1,
arrayList.size()); } }
```

- Salida de consola: `oneTimeSetUp running ...` setUp running 1 times ... tearDown running 1 times ... setUp running 2 times ... tearDown running 2 times ... oneTimeTearDown running ...

Unit Testing: JUnit4 y Eclipse

- Las anotaciones @RunWith(Suite.class) y @Suite.SuiteClasses ({Clase1.class, ..., ClaseN.class}) sobre una clase vacía permiten automatizar la ejecución de varias clases de prueba: útil en TDD para evitar tener que ejecutar varias clases de prueba una a una

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
     FixtureTest.class,
     OutputTest.class
        })
public class TestSuite { }
```

- Ejecutar la clase TestSuite como JUnitTest resultará en la ejecución de las clases FixtureTest y OutputTest

Unit Testing: JUnit4 y Eclipse

- La anotación @Ignore("motivo") sobre un método anotado como @Test hace que el test correspondiente no se ejecute. Sirve para desactivar alguna prueba temporalmente
- Al ejecutar una clase de prueba se ejecutarán todos sus métodos @Test aunque alguno de ellos falle. Sin embargo la ejecución de cada test terminará al encontrar el primer fallo de aserción (aunque pudieran fallar varias aserciones distintas en un mismo test)
- Durante la ejecución de un test, cualquier excepción no capturada, sea causada por un método @Test, @Before, @After, ..., hace que falle el test. Esto se puede utilizar para definir un test que falle si se lanza una excepción: simplemente no capturándola y añadiendo el throws correspondiente en la cabecera del método @Test
- JUnit no ofrece ninguna funcionalidad para construir mock objects: pueden construirse a mano definiendo una clase mock que herede de la clase real a la que intenta simular, y redefiniendo los métodos que se quieran simular (pej. para evitar una costosa conexión remota)
- JUnit se limita a proporcionar un framework para automatizar la ejecución y comprobación de las pruebas, y forzarnos a escribir las pruebas de una forma prefijada

Unit Testing: JUnit4 y Eclipse

- Algunos enlaces
  - FAQ y tutorial de junit.org: http://junit.sourceforge.net/doc/faq/faq.htm
  - JavaDoc de JUnit 4: http://kentbeck.github.com/junit/javadoc/latest/
  - Tutoriales de adictosaltrabajo.com:
    - básico: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=Junit4Eclipse
    - más avanzado: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=junit4

Code Inspections

- Code inspection is another technique that is often used effectively at the unit level
- A kind of “static testing” in which errors are detected not by executing the code but by reading it
- Main goal of inspection process is to detect defects in work products
- First proposed by Fagan in 70s
- Earlier used for code, now used for all types of work products
- Is recognized as an industry best practice

Code review…

- Conducted by group of programmers for programmers (i.e. review done by peers)
- Is a structured process with defined roles for the participants
- The focus is on identifying problems, not resolving them
- Review data is recorded and used for monitoring the effectiveness
- A type of formal technical review applied to the code unit level

Summary

- Goal of coding is to convert a design into easy to read code with few bugs
- Good programming practices can help
- There are many methods to verify the code of a module – unit testing and inspections are most commonly used