eXtreme Programming

Angelo Gargantini

Informatica 3 1005/2006

Materiale

- Questi lucidi + appunti
- extremeprogramming.org
  - In inglese

XP Practices

1. ***Planning game***.
2. ***Small releases***.
3. ***Metaphor***.
4. ***Simple design***.
5. ***Tests***.
6. ***Refactoring***.
7. ***Pair programming***.

1. ***Continuous integration***.
2. ***Collective ownership***.
3. ***On-site customer***.
4. ***40-hour weeks***.
5. ***Open workspace***.
6. ***Just rules***.

Planning game

- How to plan?
  - Create and prioritize user stories - customer
  - Estimate difficulties - developers
  - Select stories for next release - customer
  - Split stories into tasks - developers
  - Plan the tasks for the next iteration - customer/developers

Small Releases

- My program is all or nothing! Wrong! Inside every large program there are lots of little programs trying to get out. Make them into small releases.
- Make as many iterations as possible per release
- Keep good track of progress
- Deliver business value to the customer fast
- Gives sense of accomplishment to the team
- Keep the team focused

Customer on-site

- Customer or Developer on-site?
- Get the user stories
- Make him determine priorities – high business value first
- Get him to give you immediate and frequent feedback
- Involve him into specification of functional acceptance tests

Test First

- Test-Driven development
  - Design a test that will fail
  - Compile it and check that it fails
  - Write just enough code to make the test run
- Design evolves from tests
- The benefit must be higher than the cost
- Testing slows you down?
- Apply same quality standards for test and code
- Tests ARE documentation
- Automate

Test first: JUnit/NUnit

- Unit tests framework
- Write tests
- Execute tests
- Assert results
- Show tests failure/success
- Keep the bar green
  - http://junit.sourceforge.net/
  - http://nunit.org/default.htm

Code refactoring

Fowler says that refactoring is the

"*... process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*"

Just cleaning up code.

Why Refactor?

- To improve the quality of the codebase
- Makes software easier to understand
- This in turn helps in finding bugs
- .. and in turn allows you to program faster in the end.

Code refactoring

- Contrary to idealized development strategy:
  - analysis and design
  - code
  - test
- At first, code is pretty good but as requirements change or new features are added, the code structure tends to atrophy. Refactoring is the process of fixing a bad or chaotic design.
- Amounts to moving methods around, creating new methods, adding or deleting classes, ...

tool: RefactorIT

Rename
Renames a method, field, type, package or prefix. Updates all references.

Move Class
Moves a class or interface into another package.

Encapsulate Field
Replaces direct field usage with corresponding accessor methods.

Extract Method
Analyzes the selected piece of code and extracts it into a separate method.

tool: RefactorIT

Extract Super-class/Interface

Extracts selected methods and fields into new superclass or interface.

Minimize Access Rights

Determines the minimal access modifiers for class fields and methods. Automatically changes selected modifiers.

Create Constructor

Creates a simple constructor on group of field declarations that initializes these fields.

Simple design

- How to achieve this?
  - Evolutionary design – design evolves while coding and refactoring
  - If you don’t need it now, you won’t need it ever -develop only functionality that is required
  - Satisfy tests in the simplest possible way
  - Refactoring as soon as the tests run
- And again – keep it as simple as possible!

System Metaphor

- What?
- Everybody is involved, everybody is interested
- Everybody understands and is responsible for:
  - Base architecture
  - Whole system
- How?
  - Don’t separate the team into designers and coders
  - Change pairs often

Pair programming

- Why?
  - 2 > 1
  - Two programmers will understand the code
  - Keeps programmers focused
- When do it?
  - XP says always
  - Is it possible in real life? Is it really necessary all the time?

Continuous Integration

- You can’t put it off forever. Better do it all the time
- Spare yourself the Big Bang disaster
- Keep the tests working 100% during the integration
- Use a dedicated integrated machine
- Keep the build time low
- Automate through scripts, tools, etc…

Collective Code Ownership

- This is not my object! WRONG!
- Use version control system – CVS, VSS, ...
- You brake it, you fix it

Coding Standards

- I can always read my own code. Wait, it’s all my code!
- Let the team setup coding standards prior to coding and agree on them
- Force coding standards application, no exceptions
- Keep it simple

40 hour week

- Tired people make mistakes
- Tired people tend to overlook things like testing, refactoring, etc.
- Work at maximum concentration 8 hours per day
- 6 PM – you are tired, go home
- Don’t work more than 1 consecutive week of overtime

JUnit

Test suites

- build a test suite: a set of tests that can be run at any time
- Disadvantages of a test suite
  - It's a lot of extra programming
    - This is true, but use of a good test framework can help quite a bit
  - You don't have time to do all that extra work
    - False--Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of a test suite
  - Reduces total number of bugs in delivered code
  - Makes code much more maintainable and refactorable
    - This is a huge win for programs that get actual use!

XP approach to testing

- Tests are written before the code itself
- If code has no automated test case, it is assumed not to work
- A test framework is used so that automated testing can be done after every small change to the code
  - This may be as often as every 5 or 10 minutes
- If a bug is found after development, a test is created to keep the bug from coming back
- Consequences
  - Fewer bugs
  - More maintainable code
  - Continuous integration--During development, the program always works--it may not do everything required, but what it does, it does right

JUnit

- JUnit is a framework for writing tests
  - written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
  - uses Java's reflection capabilities (Java programs can examine their own code),
  - helps the programmer:
    - define and execute tests and test suites
    - formalize requirements and clarify architecture
    - write and debug code
    - integrate code and always be ready to release a working version
  - BlueJ, JBuilder, Netbeans, and Eclipse provide JUnit tools

Terminology

- A test fixture sets up the data (both objects and primitives) that are needed to run tests
  - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A unit test is a test of a *single* class
- A test case tests the response of a single method to a particular set of inputs
- A test suite is a collection of test cases
- A test runner is software that runs tests and reports results
  - An integration test is a test of how well classes work together
  - JUnit provides some limited support for integration tests

Structure of a JUnit test class

- Suppose you want to test a class named Fraction
- public class FractionTest
  extends junit.framework.TestCase {
  - This is the unit test for the Fraction class; it declares (and possibly defines) values used by one or more tests
- public FractionTest() { }
  - This is the default constructor

Structure of a JUnit test class

- protected void setUp()
  - Creates a test fixture by creating and initializing objects and values
  - non vediamo
- protected void tearDown()
  - Releases any system resources used by the test fixture
  - non vediamo
- public void testAdd(), public void testToString(), etc.
  - These methods contain tests for the Fraction methods add(), toString(), etc. (note how capitalization changes)
  - non vediamo
- public Test suite()

Assert methods I

- Within a test,
  - Call the method being tested and get the actual result
  - assert what the correct result should be with one of the provided assert methods
  - These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an AssertionFailedError if the test fails
  - JUnit catches these Errors and shows you the result

Assert methods I

- static void assertTrue(boolean ***test***)
  static void assertTrue(String ***message***, boolean ***test***)
  - Throws an AssertionFailedError if the test fails
  - The optional *message* is included in the Error

- static void assertFalse(boolean ***test***)
  static void assertFalse(String ***message***, boolean ***test***)
  - Throws an AssertionFailedError if the test fails

Example: Counter class

- For the sake of example, we will create and test a trivial “counter” class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value
- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)

JUnit tests for Counter

```
public class CounterTest extends junit.framework.TestCase {

    Counter counter1;

    public CounterTest() {
        counter1 = new Counter();
    }

    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }

    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

The Counter class itself

```
public class Counter {
        int count = 0;

        public int increment() {
            return ++count;
        }

        public int decrement() {
            return --count;
        }

        public int getCount() {
            return count;
        }
}
```

- Is JUnit testing overkill for this little class?
- The Extreme Programming view is: If it isn't tested, assume it doesn't work
- You are not likely to have many classes this trivial in a real program, so writing JUnit tests for those few trivial classes is no big deal
- Often even XP programmers don't bother writing tests for *simple* getter methods such as getCount()
- We only used assertTrue in this example, but there are additional assert methods

Assert methods II

- assertEquals(***expected***, ***actual***)
  assertEquals(String ***message***, ***expected***, ***actual***)
  - This method is heavily overloaded: *arg1* and *arg2* must be both objects *or* both of the same primitive type
  - For objects, uses your equals method, ***if*** you have defined it properly, as public boolean equals(Object o)--otherwise it uses ==

Assert methods II

- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***,
  Object ***actual***)
  - Asserts that two objects refer to the same object (using ==)

- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***,
  Object ***actual***)
  - Asserts that two objects do not refer to the same object

Assert methods III

- assertNull(Object ***object***)
  assertNull(String ***message***, Object ***object***)
  - Asserts that the object is null

- assertNotNull(Object ***object***)
  assertNotNull(String ***message***, Object ***object***)
  - Asserts that the object is null

- fail()
  fail(String ***message***)
  - Causes the test to fail and throw an AssertionFailedError
  - Useful as a result of a complex test, when the other assert methods aren't quite what you want

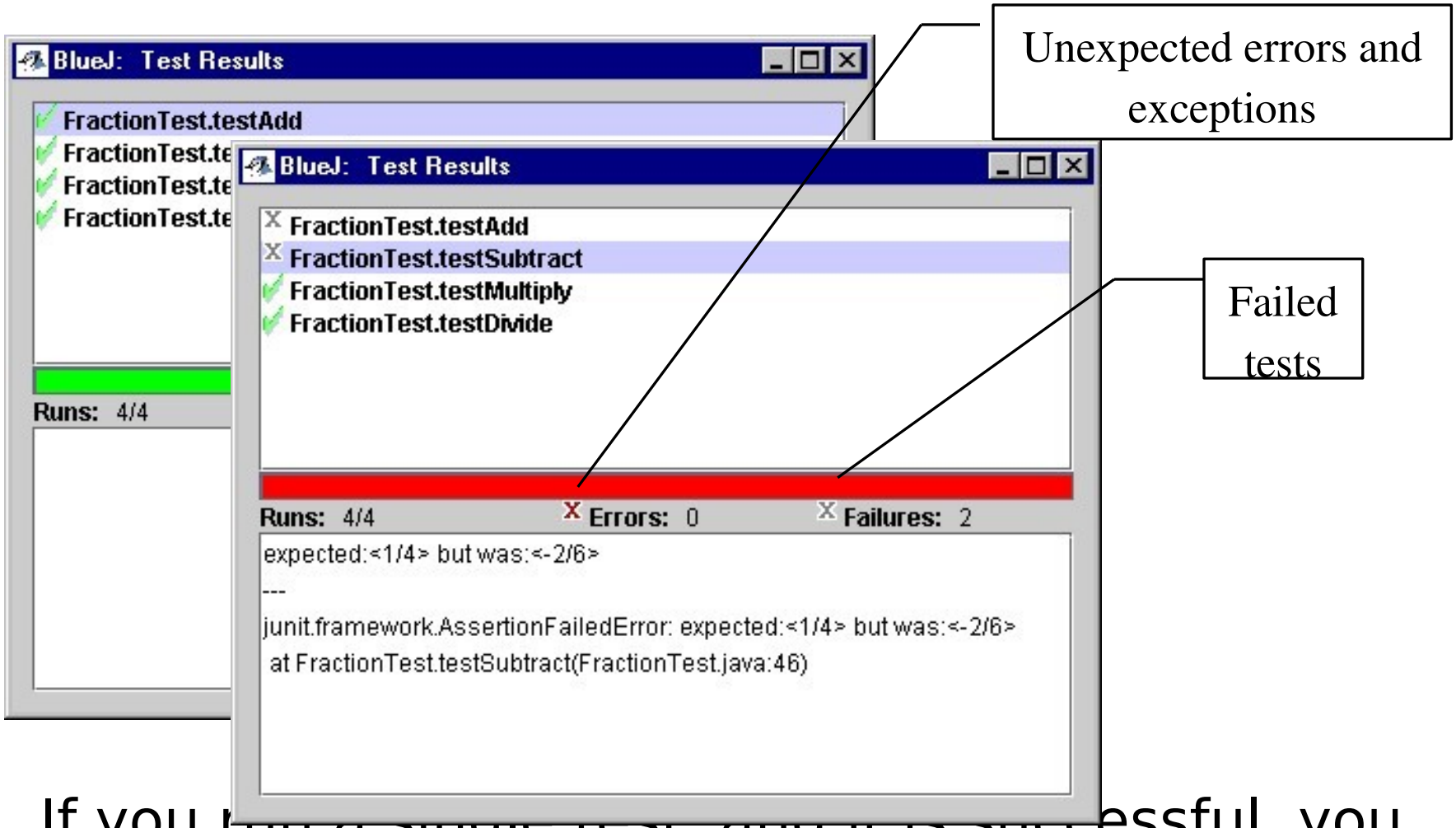
The assert statement

- Earlier versions of JUnit had an assert method instead of an assertTrue method
  - The name had to be changed when Java 1.4 introduced the assert statement

The assert statement

- There are two forms of the assert statement:
  - assert ***boolean_condition***;
  - assert ***boolean_condition***: ***error_message***;
  - Both forms throw an AssertionFailedError if the ***boolean_condition*** is false
  - The second form, with an explicit error message, is seldom necessary
- When to use an assert statement:
  - Use it to document a condition that you “know” to be true
  - Use assert false; in code that you “know” cannot be reached (such as a default case in a switch statement)
  - Do **not** use assert to check whether parameters have legal values, or other places where throwing an Exception is more appropriate

Viewing test results



If you run a single test, and it is successful, you just get a message in the status line

Problems with unit testing

- JUnit is designed to call methods and compare the results they return against expected results
  - This works great for methods that *just* return results, but many methods have side effects
    - To test methods that do output, you have to capture the output
      - It's possible to capture output, but it's an unpleasant coding chore
    - To test methods that change the state of the object, you have to have code that checks the state
      - It's a good idea in any case to write self-tests for object validity
  - It isn't easy to see how to unit test GUI code
- Private methods cannot be tested

“Functional” Style

- I think heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse

First steps toward solutions

- Rather than always printing on System.out, you can do your printing on an arbitrary PrintStream
  - The PrintStream can be passed into methods as a parameter
  - Alternatively, you can redefine System.out to use a different PrintStream with System.setOut(***PrintStream***)
- You can "automate" GUI use by "faking" events
  - Here's a starter method for creating your own events:
    - ```
      public void fakeAction(Component c) {
          getToolkit().getSystemEventQueue().postEvent(
              new ActionEvent(c, ActionEvent.ACTION_PERFORMED, ""));
      }
      ```
  - You can explore the Java API to discover how to create other kinds of events
      ```

Eclipse e Junit

Come usare junit in Eclipse

- Scrivi la tua classe al solito
- Seleziona la classe per cui vuoi creare i casi di test, tasto destro -> new -> JUnit Test Case
- Si apre un dialogo (seleziona tearDown, setUp e main se vuoi avere questi metodi - non è necessario in genere per piccoli esercizi)
- fai next -> seleziona il metodo per cui vuoi creare i casi di test
- Riempi il metodo (con eclipse devi fare tu).

•Esempio

- Ad esempio se hai un metodo foo della classe Es1 che prende un array di Stringhe, avrai un metodo testFoo in cui devi testare foo.
- Potresti scrivere istruzioni di questo genere:

```
public void testFoo() {
  String[] b = new String[0];
  expectedReturn = null;
  actualReturn = Es1.foo(b);
  assertEquals("return value", expectedReturn, actualReturn);

  String[] c = {"cane", "grattacielo", "blu"}; ;
  expectedReturn = "blu";
  actualReturn = Es1.foo(c);
  assertEquals("return value", expectedReturn, actualReturn);
}
```

•

- usa le istruzioni assertEquals e così via di Junit (vedi i lucidi)

•Esecuzione

- Per eseguire i test fai
- tasto destro sulla classe -> run As -> Junit Test

erfaccia per eseguire i casi di test)

classe Es1

```
public class Es1 {

    /** dato un array di stringhe, mi restituisce
     * la più corta null se l'array è vuoto
     */
    public static String piuCorta(String[] strs) {
        if (strs == null) {
            return null;
        }
        if (strs.length == 0) {
            return null;
        }
        // vettore non vuoto e non nullo !!!
        String cortissima = strs[0];
        for (int i = 0; i< strs.length; i++){
            if (strs[i].length()< cortissima.length())
                cortissima = strs[i];
        }
        return cortissima;
    }

}
```

Es1Test

```
public class Es1Test extends TestCase {
    protected void setUp() throws Exception {
        super.setUp();
    }
    protected void tearDown() throws Exception {
         super.tearDown();
    }
    public void testPiuCorta() {
        //
        // testo con array vuoto
        String[] strs = new String[0];    String actualReturn = Es1.piuCorta(strs);
        assertEquals("array vuoto", null, actualReturn);
        //
        // test con array null
        assertEquals("array null", null, null);
        // test con qualche array
        String[] caso1 = {"oggi", "ciao", "blu"};
        assertEquals("caso1", "blu", Es1.piuCorta(caso1));
    }
}
```

Esercizi

Nota: per confrontare due stringhe s1 e s2

s1 > s2 ? =>

```
s1.compareToIgnoreCase(s2) > 0
```

- Scrivere un metodo che prende un array di stringhe e restituisci quella maggiore (in ordine alfabetico) o null se l'array è vuoto o nullUn metodo che restituisce un array di stringhe uguale a quello passato ma ordinato

Per confrontare due array a1 e a2 usa Arrays.equals:

```
assertTrue(java.util.Arrays.equals(a1,a2)); ...
```