Agile/Automated Testing

Dr. Laurie Williams 2004
Associate Professor
North Carolina State University

Filip Bruman

Testing practices

The article presents two testing practices that are important test practices in Extreme Programming (XP) . . .

Test Driven Development (TDD) is a software development technique consisting of short iterations where new test cases covering the desired improvement or new functionality are written first, then the production code necessary to pass the tests is implemented, and finally the software is refactored to accommodate changes. (Wikipedia, 2008)

Acceptance testing is formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system. (IEEE, 1990)

Open source testing frameworks

. . .and two open source testing frameworks.

JUnit. A white box unit testing framework for Java written by Erich Gamma (yes, he is one of the four Design Patterns guys) and Kent Beck (the creator of Extreme Programming).

The Framework for Integrated Test (FIT). The FIT framework is developed for automation of black box acceptance tests and is developed by Ward Cunningham (father of the first wiki).

Main advantages of automating tests

- Running the tests over and over again gives you **confidence** that the new work just added to the system didn't break or destabilize anything that used to work and that the new code does what it is supposed to do.
- Running the tests over and over (particularly acceptance tests) can also help you understand **what portion of the desired functionality has been implemented**.
- Together, the set of automated tests can form a regression test suite. The purpose of these regression tests is to show that the software's behavior is unchanged unless it is specifically changed due to the latest software or data change (Beizer, 1990).

Test Automation Manifesto (Meszaros,Smith et al., 2003)

- **Concise** – Test should be as simple as possible and no simpler.
- **Self Checking** – Test should report its results such that no human interpretation is necessary.
- **Repeatable** – Test can be run repeatedly without human intervention.
- **Robust** – Test produces same result now and forever. Tests are not affected by changes in the external environment.
- **Sufficient** – Tests verify all the requirements of the software being tested.

Test Automation Manifesto (cont.)

- **Necessary** – Everything in each test contributes to the specification of desired behavior.
- **Clear** – Every-statement is easy to understand.
- **Efficient** – Tests run in a reasonable amount of time.
- **Specific** – Each test failure points to a specific piece of broken functionality (e.g. each test case tests one possible point of failure).
- **Independent** – Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.
- **Maintainable** – Tests should be easy to modify and extend.
- **Traceable** – Tests should be traceable to the requirements; requirements should be traceable to the tests.

Test-Driven Development (TDD)

In TDD production code is developed through rapid iterations of the following steps.

1. Spending some time doing high- and/or low-level design (optional);
2. Writing a small number of automated unit test cases;
3. Running these unit test cases to ensure they fail (since there is no code to run yet);
4. Implementing code that should allow the unit test cases to pass;
5. Re-running the unit test cases to ensure they now pass with the new code; and
6. Restructuring the production and the test code, as necessary, to make it run better and/or to have better design.

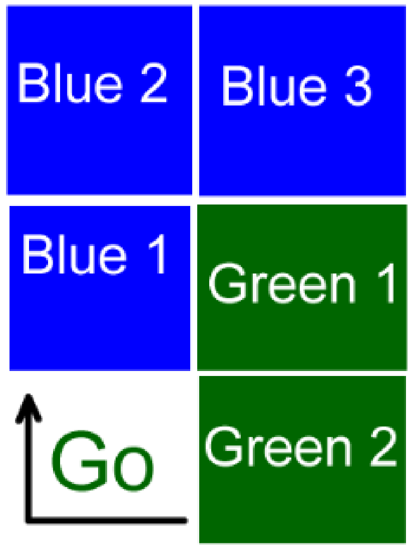
step 3?

1. There's a problem with the test, and it isn't testing what you think it is testing;
2. There's a problem with the code, and it's doing something you didn't expect it to do (it's a good idea to check this area of the code to find out what other unexpected things it's doing); and
3. Maybe the code legitimately already performs the functionality correctly – and no more new code is needed (this is a good thing to know).

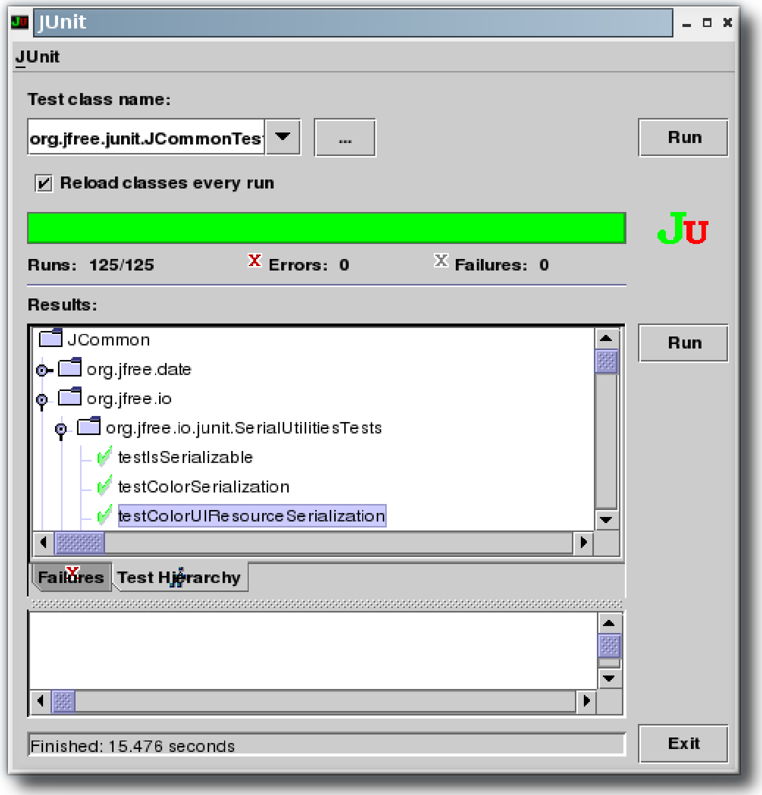
Advantages to running automated unit tests often:

- The test results tell us when we inadvertently break some existing functionality (Martin, 2003).
- You can add functions to a program or change the structure of the program without fear that you are breaking something important in the process (Martin, 2003). A good set of tests will probably tell you if you break something.
- Automated unit tests prevent backtracking and maintain development momentum (Kaner, Bach et al., 2002).

A Simple Monopoly Board Game



JUnit



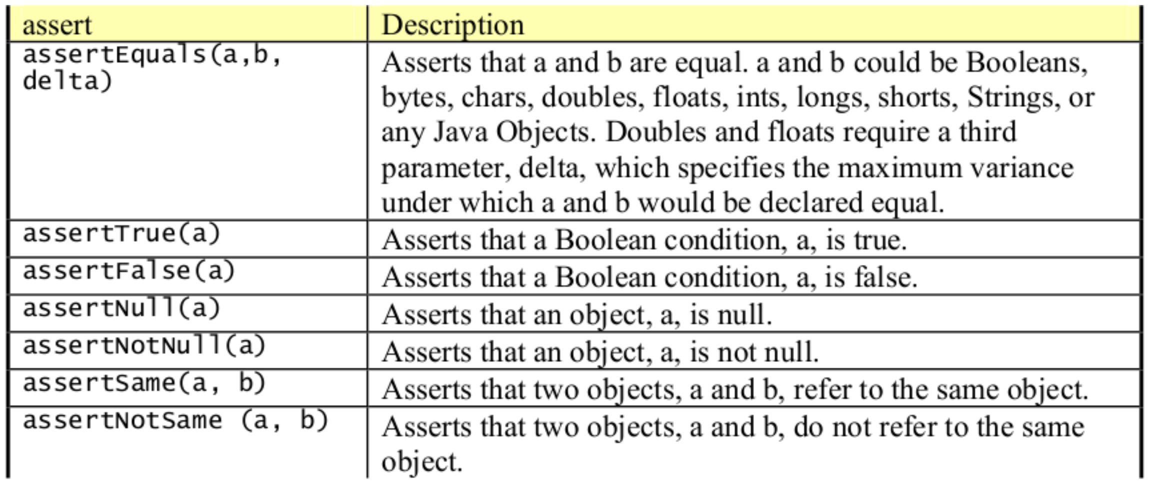
http://junit.org/index.htm

JUnit

The test code is independent from the program being tested.

There is no need to **instrument** code i.e. add lines of code to the program that are only intended to help in the testing.

The extra (instrumented) lines of code could cause errors, affect performance, and/or may need to be commented out when testing is complete.



assert	Description
`assertEquals(a,b, delta)`	Asserts that a and b are equal. a and b could be Booleans, bytes, chars, doubles, floats, ints, longs, shorts, Strings, or any Java Objects. Doubles and floats require a third parameter, delta, which specifies the maximum variance under which a and b would be declared equal.
`assertTrue(a)`	Asserts that a Boolean condition, a, is true.
`assertFalse(a)`	Asserts that a Boolean condition, a, is false.
`assertNull(a)`	Asserts that an object, a, is null.
`assertNotNull(a)`	Asserts that an object, a, is not null.
`assertSame(a, b)`	Asserts that two objects, a and b, refer to the same object.
`assertNotSame (a, b)`	Asserts that two objects, a and b, do not refer to the same object.

An example of some JUnit code

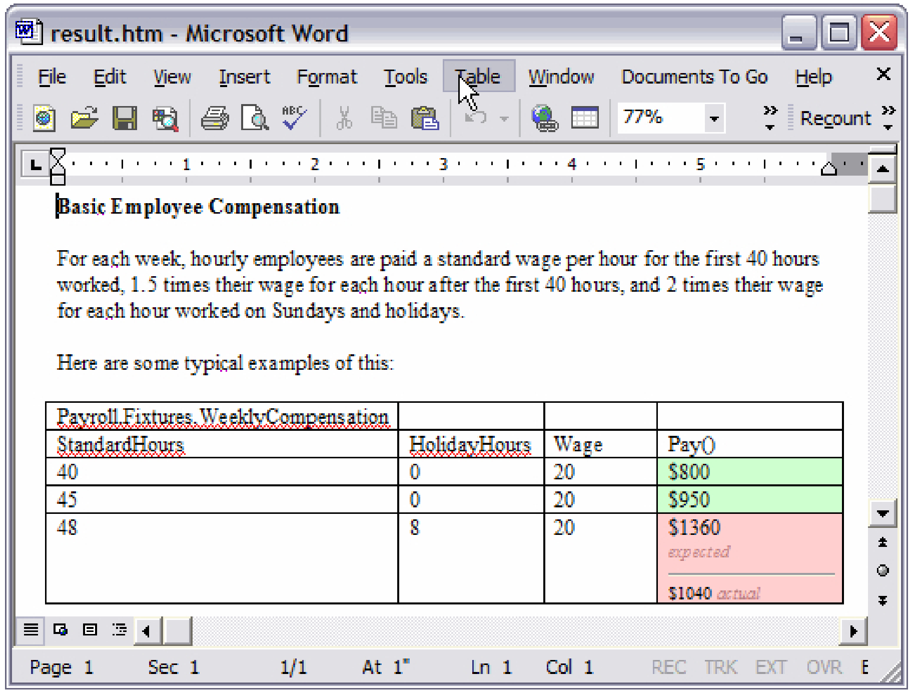
```
public class GameboardTest extends TestCase {

      :

      public void testAddCell() {
            GameBoard gameboard = new GameBoard();
            assertEquals(1, gameboard.getCellNumber());
            Cell cell = new Cell();
            gameboard.addCell(cell);
            assertEquals(2, gameboard.getCellNumber());
      }

      public void testFirstCell() {
            GameBoard gameboard = new GameBoard();
            Cell firstCell = gameboard.getCell(0);
            assertEquals(‘‘Go’’, firstCell.getName);
      }
}
```

The Framework for Integrated Test (FIT)


result.htm - Microsoft Word
File Edit View Insert Format Tools Table Window Documents To Go Help
77%
Recount
Page 1 Sec 1 1/1 At 1" Ln 1 Col 1 REC TRK EXT OVR

Basic Employee Compensation

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $1040 *actual*

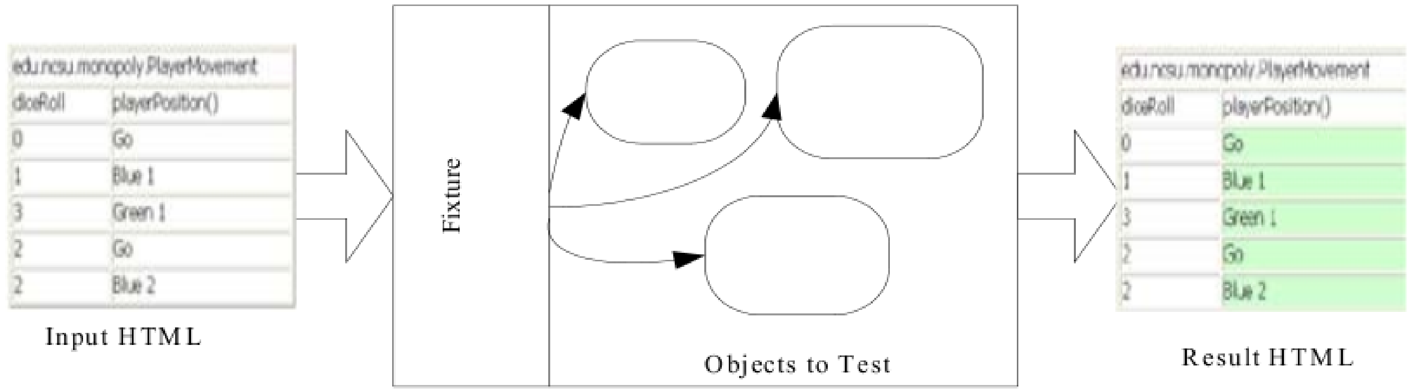
http://fit.c2.com

The Framework for Integrated Test (FIT)

Acceptance test cases are created in (HTML) tables of tests and their associated expected results.

FIT uses “runners” to parse the HTML tables, feed data to a “fixture”, run test and generate test results.

The fixtures exercise the objects that are part of the test.

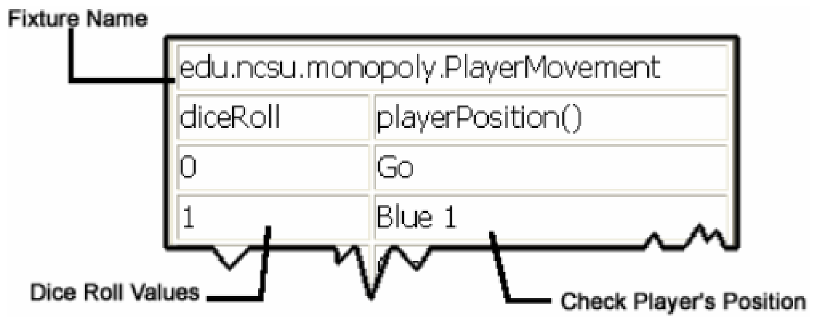


edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1
2	Go
2	Blue 2

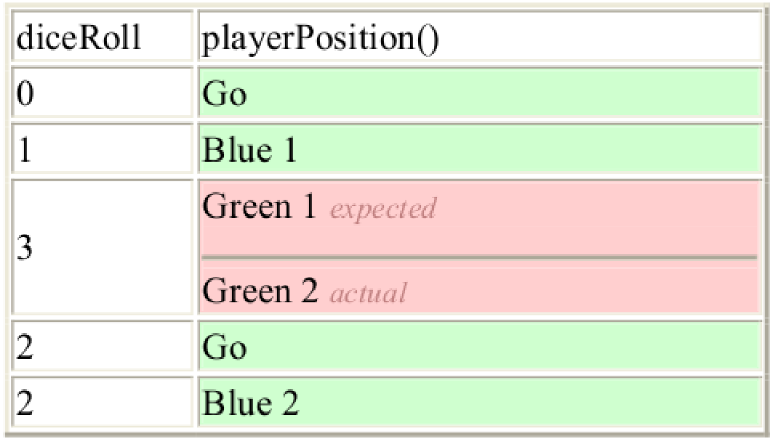
edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1
2	Go
2	Blue 2

There are three types of fixtures that can be extended: `RowFixture`, `ColumnFixture` and `ActionFixture`.

Example of `ColumnFixture`

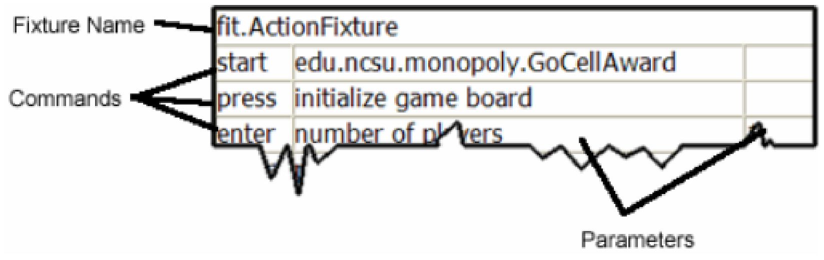


edu.ncsu.monopoly.PlayerMovement	
diceRoll	playerPosition()
0	Go
1	Blue 1

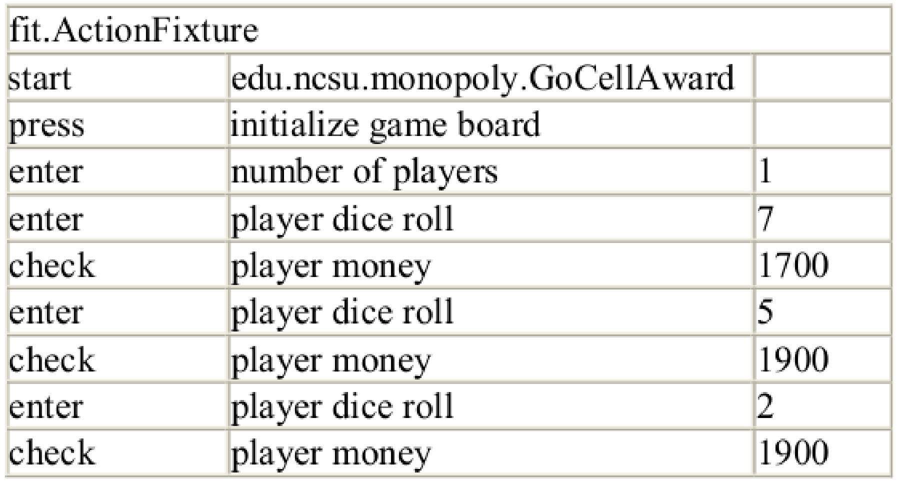


diceRoll	playerPosition()
0	Go
1	Blue 1
3	Green 1 *expected* Green 2 *actual*
2	Go
2	Blue 2

Example of `ActionFixture`



fit.ActionFixture		
start	edu.ncsu.monopoly.GoCellAward	
press	initialize game board	
enter	number of pl‍ayers	



fit.ActionFixture		
start	edu.ncsu.monopoly.GoCellAward	
press	initialize game board	
enter	number of players	1
enter	player dice roll	7
check	player money	1700
enter	player dice roll	5
check	player money	1900
enter	player dice roll	2
check	player money	1900

An example of some ActionFixture code

```
public class GoCellAward extends ActionFixture {
    private GameMaster gameMaster;
    public void initializeGameBoard() {
        gameMaster = GameMaster.instance();
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
        gameMaster.setGameBoard(new SimpleGameBoard());
    }

    public void numberOfPlayers(int number) {
        gameMaster.setNumberOfPlayers(number);
    }

    public void playerDiceRoll(int diceRoll) {
        gameMaster.movePlayer(0, diceRoll);
    }

    public int playerMoney() {
        return gameMaster.getCurrentPlayer().getMoney();
    }
}
```

Summary – ideas for Automated Test

- Running automatic tests often will help you see if your new code broke any existing functionality. Collect all the tests from the entire time for the entire code base. Run these tests often – at least once per day.
- In automating tests, consider the advice in the Test Automation Manifesto.
- When a defect is found in your code, add automated tests to reveal the defect. Then, fix the defect and re-run the automated tests to make sure they all pass now.
- Work with your customer to create acceptance tests – then automate them. You can use the number (or percent) of acceptance test cases that pass as the means of determining the progress of your project.

Questions?

References

Beizer, B. (1990). Software Testing Techniques. London, International Thompson Computer Press.

IEEE (1990). IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

Kaner, C., J. Bach, et al. (2002). Lessons Learned in Software Testing. New York, John Wiley and Sons, Inc.

Martin, R. C. (2003). Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ, Prentice Hall.

Meszaros, G., S. M. Smith, et al. (2003). The Test Automation Manifesto. Extreme Programming and Agile Methods – XP/Agile Universe 2003, Lncs 2753. F. Maurer and D. Wells. Berlin, Springer.

Wikipedia, (2008-03-05) `www.wikipedia.org`