Windows Programming
Introduction to
Test-Driven Development

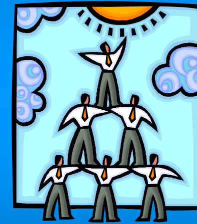
陳偉凱
台北科大資工系

Outline

- Testing and Development
  - Testing early, often and automated (XP)
  - Test-Driven Development (TDD)
  - TDD exercise

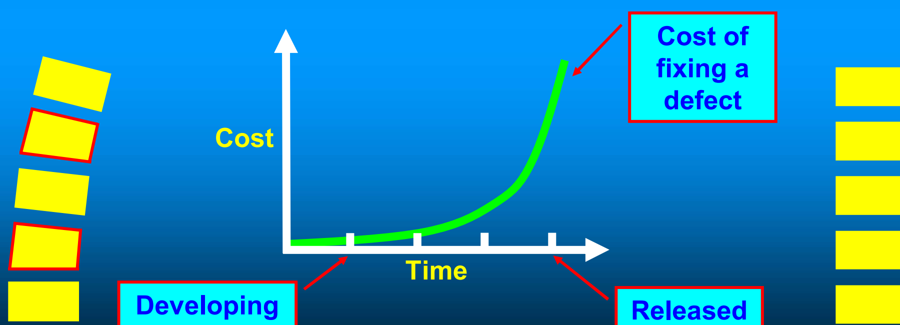
Testing: Early, Often, and Automated 1/6

- Defect destroy the trust
  - Customers need to trust the software.
  - Managers need to trust reports of progress.
  - Programmers need to trust each other.
  - Without trust
    - Defending oneself because someone else may be mistaken
- Dilemma in software development
  - Most defects ends up costing more than it would have cost to prevent them



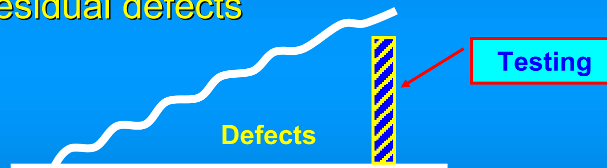
Testing: Early, Often, and Automated 2/6

- DCI (Defect Cost Increase)
  - The sooner a defect is found, the cheaper it is to fix.
  - Catch the defect the minute it is created → cost to fix it is minimal



Testing: Early, Often, and Automated 3/6

- Late testing (Long feedback loop)
  - Expensive
  - Many residual defects



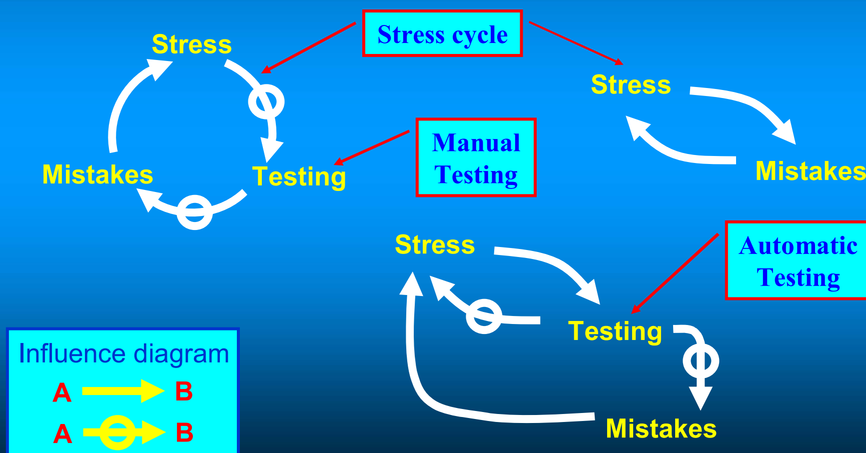
- Frequent testing (Short feedback loop)

Testing: Early, Often, and Automated 4/6

- DCI → Frequent Testing
  - Tests must be automated
    - Automated test breaks the stress cycle
  - One must write one's own tests
    - Can't wait for someone else
    - The same people who make mistakes have to write the tests
  - Bring stress and load testing (if important) into development cycle
    - Run them continuously and automatically
    - What happens if these tests find defects after development is complete?

Testing: Early, Often, and Automated5/6

- Automated test breaks the stress cycle



Testing: Early, Often, and Automated6/6

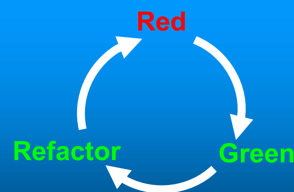
- DCI → Put testing near coding
  - Tests are written after implementation, or
  - Tests are written before implementation
    - Called Test-Driven Development
      (Used to be called test-first programming)
    - XP encourages TDD
  - Software testing is double checking
    - Testing and implementation are two different expressions of the same problem.
    - If they match → likely to be correct.
- Testing Early, Often and Automated
  - The Eclipse open source project has more than 21,000 unit tests

TDD: Goal[1/19]

- Clean code that works
  - When finished, no long bug trail.
  - A chance to think of a better thing.
  - Let's your teammates count on you, and you on them.
  - Improves the lives of the users of your software.
  - Feels good to write it.
    - Test-driven development is a lot more fun than writing tests after the code seems to be working.

TDD: The Rules[2/19]

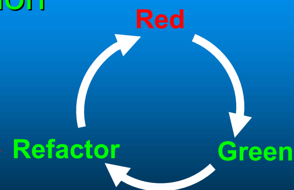
- Two simple rules
  - Write new code only if an automated test has failed
  - Eliminate duplication
- The TDD mantra
  - Red: Write a failed test (possibly doesn't compile)
  - Green: Make the test pass quickly (and dirtily)
  - Refactor: Eliminate duplication (dirty codes and tests)



TDD: The Rhythm of TDD 3/19

- TDD Rhythm
  - Quickly add a test
  - Run all tests and see the new one fail
  - Make a little change ← Do it as quickly as possible
  - Run all tests and see them all pass
  - Refactor to remove duplication

Takes time, but keeps green



TDD: The Fibonacci Number Example 4/19

- Fibonacci number f_n is defined as:

$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

- The sequence is:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

TDD: The Fibonacci Number Example 5/19

RED
1. Write the first test

```
public void testFibonacci() {
    assertEquals(0, fib(0));
}
```

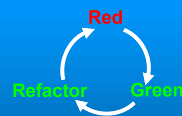
Defines interface & functionality

Run all tests → fail (no fib())

Implement fib()

```
int fib(int n) {
    return 0;
}
```

Quick fix



GREEN 2. Run all tests → pass

REFACTOR 3. Refactor → pass

$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

TDD: The Fibonacci Number Example 6/19

4. Write the second test

```
public void testFibonacci() {
    assertEquals(0, fib(0));
    assertEquals(1, fib(1));
}
```

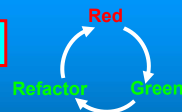
Define new functionality

Run all tests → Fail

Add new functionality

```
int fib(int n) {
    if (n == 0) return 0;
    return 1;
}
```

Quick fix



5. Run all tests → pass

$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

TDD: The Fibonacci Number Example 7/19

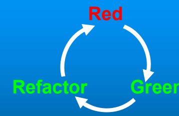
6. Refactor (test case)

```
public void testFibonacci() {
    int cases[ ][ ]={{0,0},{1,1}};
    for (int i=0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Run all tests → pass

Refactor test code

Keeps green



$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

TDD: The Fibonacci Number Example 8/19

7. Write the third test

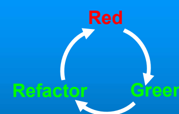
```
public void testFibonacci() {
    int cases[ ][ ]={{0,0},{1,1},{2,1}};
    for (int i=0; I < cases.length; i++)
        assertEquals(cases[i][1],
          fib(cases[i][0]));
}
```

Run all tests → pass

8. Run all tests → pass
9. Refactor → pass

Define new functionality

$f_2 = 1$



$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

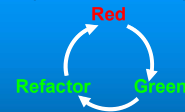
TDD: The Fibonacci Number Example 9/19

Define new functionality

10.Write the fourth test

```
public void testFibonacci() {
    int cases[ ][ ]={{0,0},{1,1},{2,1},{3,2}};
    for (int i=0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Run all tests → fail



$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

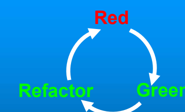
NTUT CSIE 17

TDD: The Fibonacci Number Example 10/19

Add new functionality

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 2;
}
```

11.Run all tests → pass



$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

NTUT CSIE 18

TDD: The Fibonacci Number Example 11/19

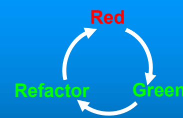
12.Refactor

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return fib(n-1)+fib(n-2);
}
```

Run all tests → pass

13.Write the fifth test

14.Run all tests → pass



Red

Refactor

Green

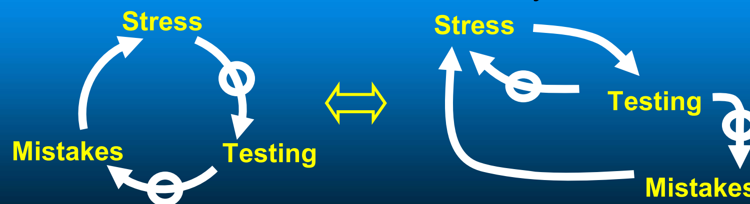
$$f_n = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ f_{n-1} + f_{n-2} & if \quad n \geq 2 \end{cases}$$

TDD: TDD Patterns 12/19

- Test
- Isolated Test
- Test List
- Test First
- Assert First
- Test Data
- Evident Data

TDD: TDD Patterns 13/19

- Test (noun)
  - Test (verb): to evaluate (e.g. poking buttons and look at answers on the screen)
  - Test (noun): a procedure leading to acceptance or rejection
  - No software engineers release even the tiniest change without testing
  - Testing changes is not the same as having tests
  - Automated test breaks the stress cycle

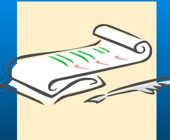


TDD: TDD Patterns 14/19

- Isolated Test
  - Long running, overnight, GUI-based tests (Kent Beck)
    - Good days, bad days
    - A huge stack of paper didn't mean a huge list of problem
  - Make the tests so fast to run them often
    - Catch errors before anyone else sees them
  - Seeking test at smaller scale than the whole AUT
  - Tests should be able to ignore one another completely (isolated test)
    - Tests are order independent → a subset of tests can be run
    - Work hard to break problems into little orthogonal dimensions → to make test easy and quick
    - Encourage solutions out of many highly cohesive, loosely coupled objects.

TDD: TDD Patterns 15/19

- Test List
  - A "now" list and a "later" list
  - Write a list of all tests you know you have to write
    - Don't implement them all (long way from green bar)
    - Test a little, code a little
    - Never more than one change away from green bar
  - What we put on the list is what we want to implement
  - The implementation will imply new tests
    - Write the new tests down on the list
    - Write refactoring needs on the list
    - For larger refactorings move them to "later" list
  - Never move a test case to "later" list



TDD: TDD Patterns 16/19

- Test First
  - Write the test before the code to be tested
- Assert First
  - Write asserts in a test case before other test codes

```
TestCompleteTransaction() {
    …
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents);
}
```

```
TestCompleteTransaction() {
    Server writer = Server( "abc");
    Socket reader = Socket("localhost");
    Buffer reply = reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents);
}
```

TDD: TDD Patterns 17/19

- Test Data
  - Use data that makes tests easy to read and follow
  - If the system has multiple inputs then tests should reflect multiple inputs
  - Never use the same constant to mean more than one things
    - One constant one meaning
    - Test 2+2 is bad → test 2 + 3 is better (order of argument)
  - Realistic data is useful when
    - Testing real-time system using trace of external events gathered from the actual execution
    - Matching the output of the current system with the output of a previous one
    - Refactoring a simulation and expect precisely the same answers

TDD: TDD Patterns 18/19

- Evident Data
  - How to represent the intent of the data?
    - Include expected and actual results in the test, and try to make their relationship apparent

```
Bank bank = new Bank();
bank.addRate("USD", "GBP", 2);
bank.Comission(0.015);
Money result = bank.convert(new Note(100, "USD", "GBP"));
assertEqual(new Note(100 / 2 * (1 – 0.015), "GBP"), result);
```

Actual

Excepted

  - An exception to "no magic numbers"
    - For small scopes
    - Use symbolic constants if it's already there

TDD: Bottom Line19/19

- What do I do when a defect is reported?
  - Write a failing test that exposes the defect
    - When the test passes, you know the defect is fixed!
  - This is a learning opportunity
    - Perhaps the defect could have been prevented by being more aggressive about testing everything that could reasonably break.
- When you need to add new functionality to the system, write the tests first.
- If you find yourself debugging using println() or debugger write a test case instead.

References

1. Kent Beck, Test-Driven Development, By Example, Addison-Wesley.
2. Kent Beck, Extreme Programming Explained, Embrace Change, 2nd Edition, Addison-Wesley.

TDD: exercise