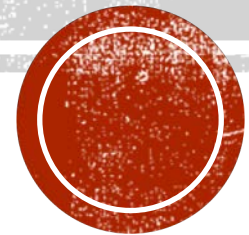


SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





Test Driven Development



Best practices



Advantages and Disadvantages



Fakes, mocks, and integration tests



Example

OUTLINE

INTRODUCTION



A software development process.



Repetition of a very short development cycle



First write an initially failing automated test case that defines a desired improvement or new function



then produce the minimum amount of code to pass that test



and finally re-factor the new code to acceptable standards.



Related to XP



TDD is a design (and testing) approach involving short, rapid iterations of

Unit Test
Code
Re-factor



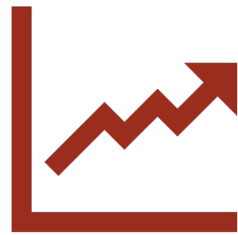
Unit tests are automated



Forces to consider use of a method before implementation of the method

WHAT IS TDD?

WHY TDD?



Software development

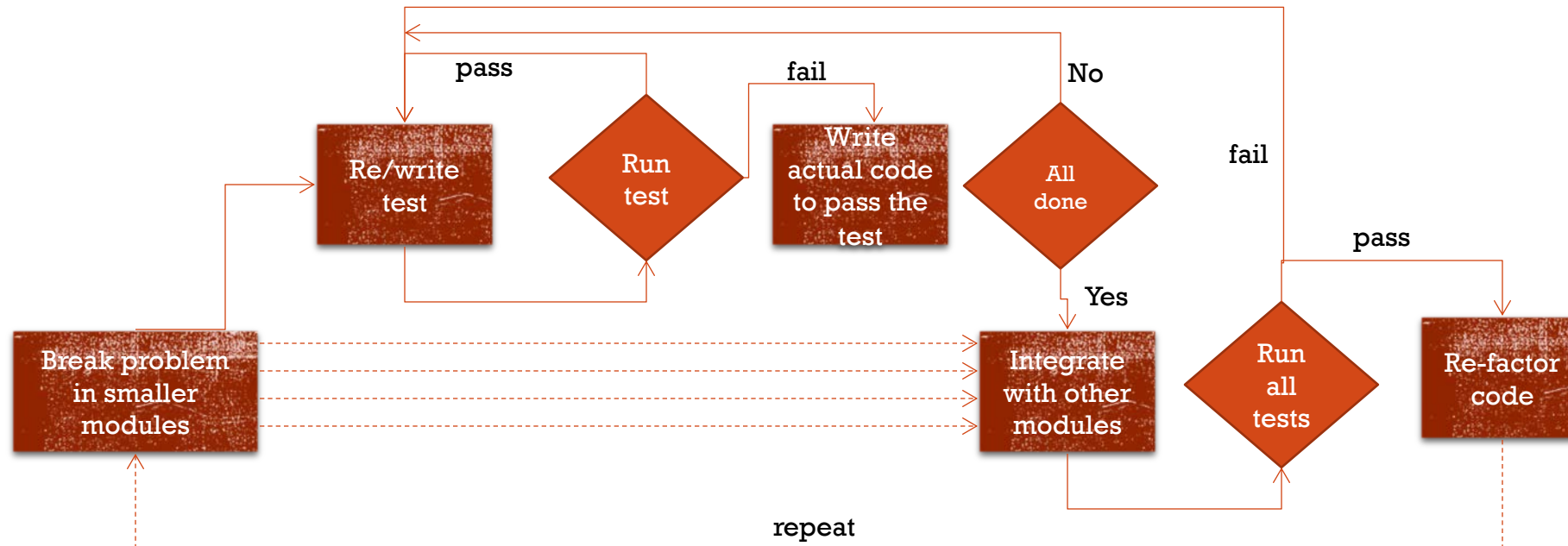
Should be faster
As economical as it could be
Good quality product



With TDD

Good programmers are more effective
Testing can close the gap
Quality improvement

TEST-DRIVEN DEVELOPMENT CYCLE





"keep it simple stupid" (KISS)



"you aren't gonna need it" (YAGNI)



By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer.



To achieve some advanced design concept, such as a design pattern, tests are written that generate that design.



The code may remain simpler than the target pattern, but still pass all required tests.

DEVELOPMENT STYLE



For TDD, a unit is most commonly defined as a class or group of related functions often called a module.



Keeping units relatively small is claimed to provide critical benefits.



Reduced Debugging Effort

When test failures are detected, having smaller units aids in tracking down errors.



Self-Documenting Tests

Small test cases have improved readability and facilitate rapid understandability.

DEVELOPMENT STYLE



Advanced practices of TDD can lead to Acceptance Test-driven development (ATDD) .



The criteria specified by the customer are automated into acceptance tests.



Give customer an automated mechanism to decide whether the software meets their requirements.



Focused development.

DEVELOPMENT STYLE



A commonly applied structure for test cases has:

setup, execution, validation, and cleanup



Treat your test code with the same respect as your production code.



It must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.



Review your tests and test practices with team to share effective techniques and catch bad habits.

BEST PRACTICES



Do not have test cases depend on system state manipulated from previously executed test cases.



Execution order has to be specifiable and/or constant.



Do not test precise execution behavior timing or performance.



Do not try to build “all-knowing oracles.”

PRACTICES TO AVOID, OR "ANTI- PATTERNS"

ADVANTAGES



A study shows that programmer using TDD are:

more productive
Rarely debug
Effective and efficient system design



Few failed systems



No unnecessary code



Focus on task



Reduced bugs in the code



More time needed initially



Requirements change



May not be flexible if only pass and fail is required



Code maintenance



Complex systems may require extended development

DISADVANTAGES



Unit tests focus on a unit



A complex module may have a thousand unit tests



Tests used for TDD should never cross process boundaries in a program.



Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite.



Introducing dependencies on external modules or data also turns unit tests into integration tests.

FAKES, MOCKS AND INTEGRATION TESTS



When code under development relies on a database, a web service, or any other external process

an interface should be defined that describes the access available



The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock.



Fake and mock objects:

methods return data to help the test process by always returning the same, realistic data that tests can rely upon



Integration tests:

any database should always be designed carefully with consideration of the initial and final state of the database, even if any test fails.

FAKES, MOCKS AND INTEGRATION TESTS

EXAMPLE — TEST-DRIVEN FIBONACCI

- The Fibonacci numbers or series or sequence are the numbers in the following integer sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- The sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0, F_1 = 1$$

EXAMPLE — TEST-DRIVEN FIBONACCI

- The second test shows that $\text{fib}(1) = 1$.

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

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

EXAMPLE — TEST-DRIVEN FIBONACCI

- The first test shows that $\text{fib}(0) = 0$. The implementation returns a constant.

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

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

EXAMPLE — TEST-DRIVEN FIBONACCI

- We can factor out the common structure of the assertions by driving the test from a table of input and expected values.

```
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]));  
}
```

EXAMPLE — TEST-DRIVEN FIBONACCI

- Now adding the next case requires 6 keystrokes and no additional lines:

```
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]));  
}
```

- The test works. It just so happens that our constant “1” is right for this case as well.

EXAMPLE — TEST-DRIVEN FIBONACCI

- On to the next 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]));  
}
```

- Hooray, it fails. Applying the same strategy as before (treating smaller inputs as special cases), we write:

EXAMPLE — TEST-DRIVEN FIBONACCI

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

- Now we are ready to generalize. We wrote “2”, but we don’t really mean “2”, we mean “1 + 1”.

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

EXAMPLE — TEST-DRIVEN FIBONACCI

- That first “1” is an example of fib(n-1):

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

- The second “1” is an example of fib(n-2):

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

EXAMPLE — TEST-DRIVEN FIBONACCI

- Cleaning up now, the same structure should work for fib(2), so we can tighten up the second condition:

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

- And there we have Fibonacci, derived totally from the tests.

HOMEWORK



Review class notes.



Additional reading:
Examples of UML diagrams



Start a discussion on Google
Groups to clarify your doubts.