

eXtreme Programming and TDD

Stefano Fornari, Edoardo Schepis

Example: Fibonacci Function

Our goal: fib(x)

{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 }

A number in the Fibonacci sequence is generated by taking the sum of the previous two numbers.

The first test shows that $\text{fib}(0) = 0$.

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

Example: Fibonacci Function

Our goal: fib(x)

{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 }

A number in the Fibonacci sequence is generated by taking the sum of the previous two numbers.

The first test shows that $\text{fib}(0) = 0$.

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

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

Example: Fibonacci Function

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

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

Example: Fibonacci Function

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

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

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

Example: Fibonacci Function

The duplication in the test case is starting to bug me, and it will only get worse as we add new cases.

Let's drive 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]));  
}
```

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]));  
}
```

Example: Fibonacci Function

Disconcertingly, the test works.

It just so happens that our constant “1” is right for this case as well. 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!!!

Example: Fibonacci Function

Disconcertingly, the test works.

It just so happens that our constant “1” is right for this case as well. 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!!!

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


Example: Fibonacci Function

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;  
}
```

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: Fibonacci Function

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);  
}
```

DONE!!!

Test Driven Development

How to learn TDD

- Many books are articles now printed and published on the web
- It's OK to read *one*. It almost doesn't matter which one.
- *Test Driven Development*, Kent Beck
- Generally, people are still “learning by doing”, mostly by “doing it with somebody that already knows how”

Definition: Test Driven Development

- *A software development process*
 - Not a testing technique, per se, but depends heavily on testing as a tool
- Write tests first – the tests determine what code is to be written
- Testing is done in a fine-grained fashion

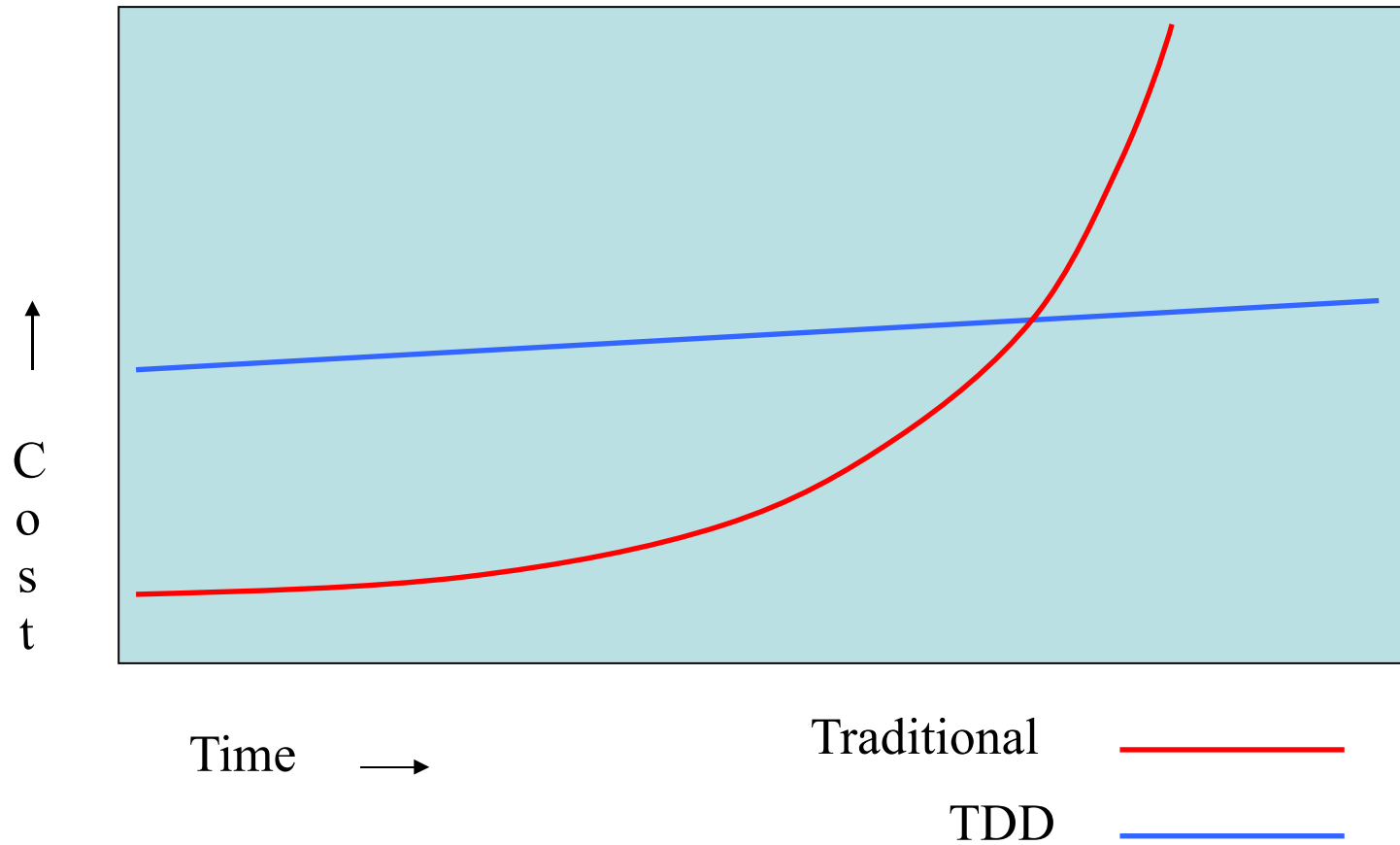
Characteristics of TDD

- Makes software development predictable on
 - Reliability
 - Scheduling, development cost
- Results in “clean code that works”
 - [Ron Jeffries]
- TDD is generally a white-box unit-testing mechanism
- Taking small steps prevents bugs and the need for debugging
- Design optional; will emerge from the tests if necessary

Design not necessary

- First step in the procedure will always be to identify a small change to be made
- That change can be identified from
 - a formal design specification,
 - a requirement spec,
 - a user story (use case),
 - or an ad-hoc informal request from a user.
- All tests are saved forever, and are a record of requirements.
 - The tests replace the requirements and design specs





Cost of development



Rules for developers

- Unit testing is not separable from coding
- Start as simply as possible
- Write new code **ONLY** if a test is failing
 - The tests provide the reason for writing a line of code
 - Write a failing test before writing a line of code
- Eliminate duplication of code and simplify code ruthlessly
 - Fewer lines of code mean fewer tests to write and maintain, prevents mushrooming of the test base
- **ALL** tests are saved in the automated regression test suite

Technique

- Initially the program works 
- Add a test that calls a new (unimplemented) feature; you get a syntax error because the method isn't even defined yet. 
- Add a stubbed version, which fails the test (presumably). 
- Fix it and all tests run. 
- Refactor towards a better design
- Run the test again
 - “Proves” that the better code is still correct

Technique 2

- Identify a “smallest possible” change to be made
- Implement test and (the one line of) code for that change (see previous slide)
- Run *all* tests
- Save test and code together in source control system
- Repeat

Elements of TDD unit tests

- Testing and reporting tool (xUnit)
- Test suites (groups of tests)
- Tests
- Mock resources
- Test library (assert implementations, etc.)
- Product-specific setup library

Why does TDD work?

- Encourages “divide-and-conquer”
- Programmers are *never* scared to make a change that might “break” the system
- The testing time that is often squeezed out of the end of a traditional development cycle *cannot* be squeezed out.

eXtreme Programming

What is XP?

- Who is behind XP?
 - Kent Beck, Ward Cunningham, Ron Jeffries
- Short definition
 - lightweight process model for software development
- What's in the name?
 - code is in the centre of the process
 - practices are applied extremely
- What is new in XP?
 - none of the ideas or practices in XP are new
 - the combination of practices and their extreme application is new

Practices

- XP is based on the extreme application of 12 practices (guidelines or rules) that support each other:
 - Planning game
 - Frequent releases
 - System metaphor
 - Simple design
 - Tests
 - Refactoring
 - Pair programming
 - Collective code ownership
 - Continuous Integration
 - Forty-hour week
 - On-site customer
 - Coding standards

Planning Game

- **Pieces:** user stories
- **Players:** customer & developer
- **Moves:**
 - **User story writing**
 - requirements are written **by the customer** on small index cards
 - user stories are written in business language
 - and describe things that the system needs to do
 - each user story is assigned a **business value**
 - Example (payroll system):
 - An employee making \$10 an hour works four hours of overtime on Friday and two on Sunday. She should receive \$60 for the Friday and \$40 for the Sunday
 - for a few months projects there may be 50-100 user stories²

Planning Game (2)

- **Moves:**
 - **Story estimation**
 - each user story is assigned a cost **by the developer**
 - cost is measured on ideal weeks (1-3 weeks)
 - a story is split **by the customer** if it takes longer than 3 weeks to implement
 - **Commitment**
 - customer and developer decide which user stories constitute the next release
 - **Value and Risk first**
 - developer orders the user stories of the next release so that
 - more valuable or riskier stories are moved earlier in the schedule
 - a fully working (sketchy) system is completed (in a couple of weeks)

Frequent Releases

- The development process is highly iterative
- A release cycle is usually up to 3 months
- A release cycle consists of iterations up to 3 weeks
- In each iteration the selected user stories are implemented
- Each user story is split in programming tasks of 1-3 days
- small and frequent releases provide frequent feedback from the customer

Tests

- Tests play the most important and central role in XP
- Tests are written before the code is developed
 - forces concentration on the interface
 - accelerates development
 - safety net for coding and refactoring
- **All** tests are automated (test suites, testing framework)
- If user stories are considered as the requirements then Tests can be considered as the specification of the system
- 2 kinds of test:
 - Acceptance tests (functional tests)
 - clients provide test cases for their stories
 - developers transform these in automatic tests
 - Unit tests
 - developers write tests for their classes (before implementing the classes)
 - All unit tests must run 100% successfully all the time

Refactoring

- **Change it even if it is not broken!**
- Process of improving code while preserving its function
- The aim of refactoring is to
 - make the design simpler
 - make the code more understandable
 - improve the tolerance of code to change
- The code should not need any comments
 - There is no documentation in XP
 - The code and the user stories are the only documents
- Useful names should be used (system metaphor)
- Refactoring is continuous design
- Remove duplicate code
- Tests guarantee that refactoring didn't break anything that worked!

Pair programming

- Two programmers sit together in front of a workstation
 - one enters code
 - one reviews the code and thinks
- “Pair programming is a dialog between two people trying to simultaneously program and understand how to program better”,
Kent Beck
- Second most important practice after tests
- Pairs change continuously (few times in a day)
 - every programmer knows all the aspects of the system
 - a programmer can be easily replaced in the middle of the project
- Costs 10-15% more than stand-alone programming
- Code is simpler (fewer LOC) with less defects (15%)
- Ensures continuous code inspection (SE)

Collective code ownership

- The code does not belong to any programmer but to the team
- Any programmer can (actually **should**) change any of the code at any time in order to
 - make it simpler
 - make it better
- Encourages the entire team to work more closely together
- Everybody tries to produce a high-quality system
 - code gets cleaner
 - system gets better all the time
 - everybody is familiar with most of the system

Continuous integration

- Daily integration at least
- The whole system is built (integrated) every couple of hours
- XP feedback cycle:
 - develop unit test
 - code
 - integrate
 - run all units tests (100%)
 - release
- A working tested system is always available

40 hour week

- “Overtime is defined as time in the office when you don’t want to be there” *Ron Jeffries*
- Programmers should not work more than one week of overtime
- If more is needed then something is wrong with the schedule
- Keep people happy and balanced
- Rested programmers are more likely to refactor effectively, think of valuable tests and handle the strong team interaction

On-site customer

- User stories are not detailed, so there are always questions to ask the customer
- The customer must always be available
 - to resolve ambiguities
 - set priorities
 - provide test cases
- Customer is considered part of the team

Coding standards

- Coding standards make pair programming and collective code ownership easier
- Common name choosing scheme
- Common code formatting

Listen-Test-Code-Design

- Traditional Software Lifecycle:
 - Listen - Design - Code - Test
- XP lifecycle
 - Listen - Test - Code - Design
- **Listen** to customers while gathering requirements
- Develop **test** cases (functional tests and unit tests)
- **Code** the objects
- **Design** (refactor) as more objects are added to the system

Requirements

- small teams (up to 10-15 programmers)
- common workplace and working hours
- all tests must be automated and executed in short time
- on-site customer
- developer and client must commit 100% to XP practices

XP is successful because...

- XP can handle changing customer requirements, even late in the life cycle
- XP stresses customer satisfaction; it delivers
 - what the customer needs
 - when the customer needs it
- XP emphasises team work
- XP is fun