



软件工程

测试驱动的开发

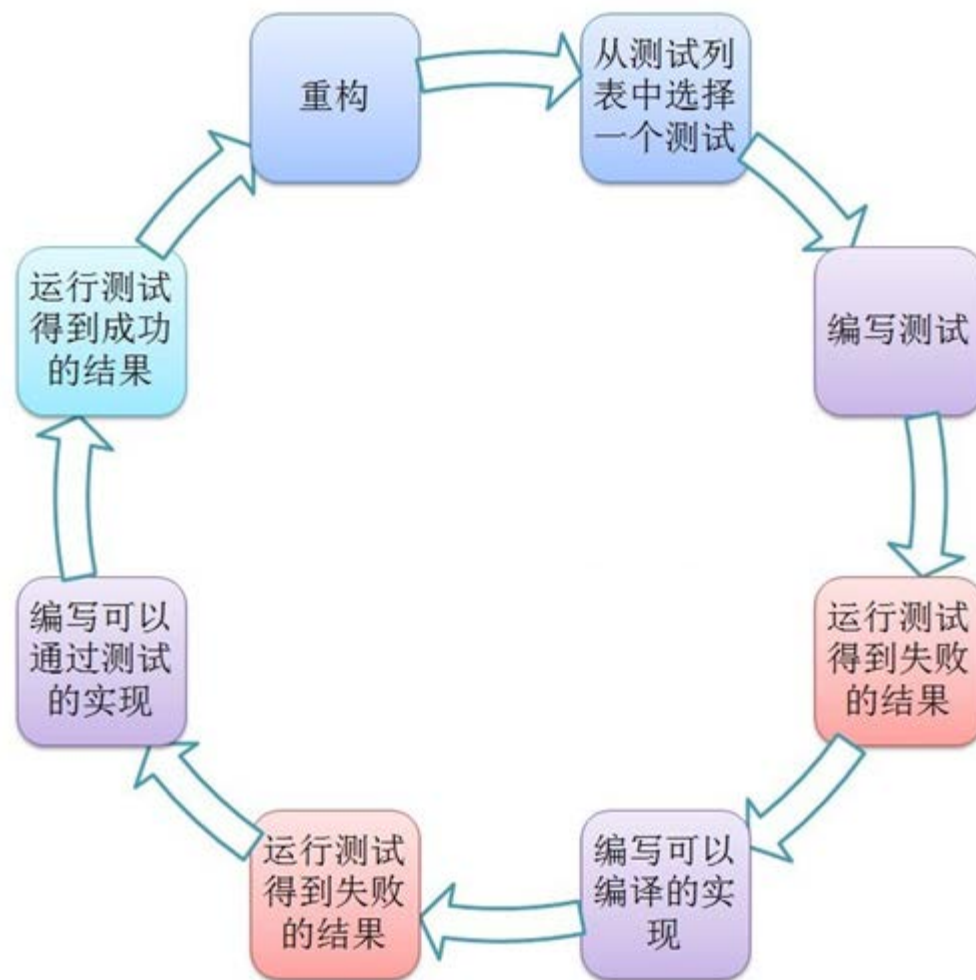
刘铭

2018年3月5日

主要内容

- 1.TDD的基本思想与过程
- 2.TDD实例之一：Fibonnaci数列
- 3.TDD实例之二：保龄球记分器
- 例子：<http://phpunit.de/manual/3.7/en/test-driven-development.html>
- <http://stackoverflow.com/questions/787172/what-is-a-good-sample-class-to-demonstrate-tdd>
- Ken Beck. Test-Driven Development: by Example (《测试驱动开发：实战与模式解析》，机械工业出版社，2013年9月，ISBN 9787111423867)

TDD的基本过程



测试驱动的开发(TDD)

- **测试驱动开发(Test-driven development):** XP中倡导的程序开发方法, 先写测试程序, 然后编码实现。
- **戴两顶帽子思考的开发方式, 测试驱动着整个开发过程:**
 - **测试驱动代码的设计和功能的实现:** 先戴上实现功能的帽子, 在测试的辅助下, 快速实现其功能;
 - **测试驱动代码的再设计和重构:** 再戴上重构的帽子, 在测试的保护下, 通过去除冗余的代码, 提高代码质量。
- **TDD所追求的目标: 代码整洁可用(Clean code that works)**
 - 除非缺乏某个功能将导致测试失败, 否则就拒绝在程序中实现该功能;
 - 除非由于缺少某行代码将导致测试失败, 否则就拒绝在程序中增加哪怕一行代码;
 - 首先编写失败的测试表明需要一项功能, 再逐渐地增加那项功能使测试通过。

测试驱动的开发(TDD)

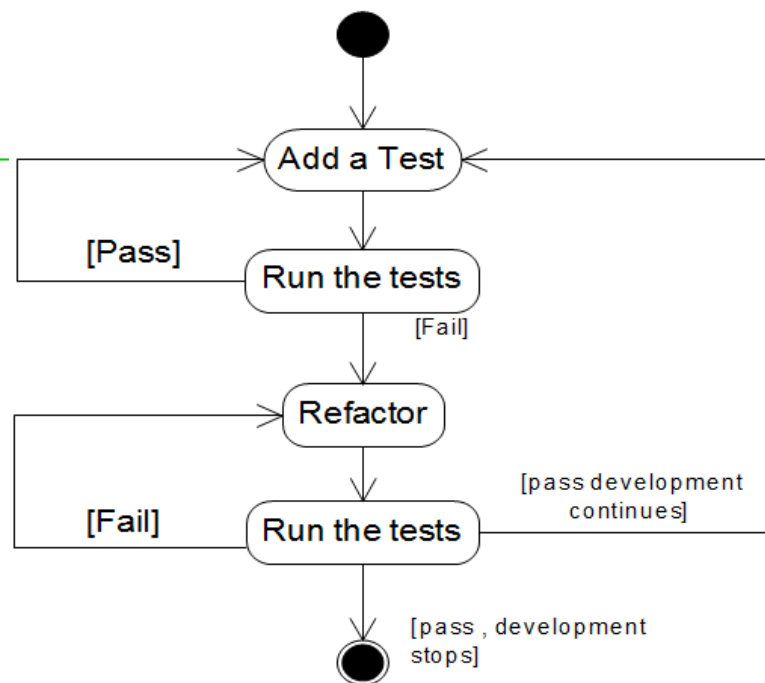
■ 在TDD中：

- 只有自动测试失败时，我们才重写代码；
- 消除重复设计，优化设计结构。
- 必须自己写测试程序(测试很多、很频繁，不能依赖他人写测试)
- 开发环境必须能够迅速响应很小的变化；

■ TDD的阶段：

- 不可运行：写一个不能工作的测试程序，一开始甚至不能编译；
- 可运行：尽快让其工作，为此可以在程序中使用一些不合情理的方法；
- 重构：消除在程序工作中产生的重复设计，优化设计结构。

■ TDD的口号：不可运行→可运行→重构



TDD的优势

- 程序中的每一项功能都有测试来验证它的操作的正确性；
- 从程序调用者的视角来观察要编写的程序，首先关注程序的接口，设计出易于调用的软件；
- 迫使自己把程序设计为可测试的；
- 把测试作为一种无价的文档，一个测试用例就像一套范例，帮助其他程序员如何使用代码。

依赖关系与重复设计

- A和B，在不修改A的情况下无法改动B，则B依赖于A；
 - 例子：使用某个数据库厂商的SQL语言，导致系统依赖该数据库产品，无法移植到其他数据库系统。
- TDD目标：编写一个对我们有用的测试而无需改动代码；
- 消除程序中的重复设计就是消除依赖关系（TDD的第二条规则）。
- 减少测试程序和代码之间的耦合度。

Make Test Work

- **The three approaches to making a test work cleanly are:**
 - Fake it
 - It's good for your morale to see your tests succeed.
 - Triangulation
 - It's useful when you are not definitely sure what the correct abstraction is.
 - Obvious implementation
 - You should only apply the obvious implementation directly only if you are sure it will work.



Fake It

- What is your first implementation once you have a broken test?
 - Return a constant.
- Once you have the test running, gradually transform the constant into an expression using variables.

Fake It

- **Why would you do something that you know you will have to rip out?**
- **Having something running is better than not having something running.**
 - Psychological — Having a green bar feels good (as opposed to having a red bar).
 - When the bar is green, you know where you stand.
 - You can refactor from there with confidence.
 - Scope control — Programmers are good at imagining all sorts of future problems.
 - Focus - Starting with one concrete example and generalizing from there prevents you from prematurely confusing yourself with extraneous concerns.
 - Focusing on the next test case is easier, knowing that the previous test is guaranteed to work.

Fake It

- Does Fake It violate the rule that says you don't write any code that isn't needed?
- **Not really - because in the refactoring step you are eliminating duplication of data between the test case and the code.**

Usually you use this when you cannot tell how to implement certain functionality, or your previous steps were too large and you cannot figure out what went wrong.
Something that works is better than something that doesn't work!

Triangulate

- **How do you most conservatively drive abstraction with tests?**
 - Abstract only when you have two or more examples.
- **Suppose we want to write a function that will return the sum of two integers. We write:**

```
public void testSum() {
    assertEquals(4, plus(3, 1));
}

private int plus(int augend, int addend) {
    return 4;
}
```

You set different cases in which the test should work and find the “generic” form of it later.

- **If we are triangulating to the right design, we have to write:**

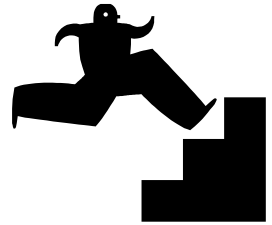
```
public void testSum() {
    assertEquals(4, plus(3, 1));
    assertEquals(7, plus(3,4));
}
```

- When we have the second example, we can abstract the implementation of plus():

```
private int plus(int augend, int addend)
{
    return augend + addend;
}
```

Obvious Implementation

- How do you implement simple operations?
 - Just implement them.
 - Fake It and Triangulation are teensy-weensy tiny steps.
 - Sometimes you are sure you know how to implement an operation. Go ahead. (plus example)
 - If you are getting surprised by red bars - go to smaller steps.
 - If you know what to type, and you can do it quickly, then do it.
- BUT**
- By using only Obvious Implementation, you are demanding perfection of yourself.
 - Psychologically, this can be a devastating move.
 - What if what you write isn't really the simplest change that could get the test to pass?
 - What if your partner shows you an even simpler one?



TDD的缺陷

- 开发者可能只完成满足了测试的代码，而忽略了对实际需求的实现➔结对编程
- 会放慢开发实际代码的速度，特别对于要求开发速度的原型开发造成不利。
- 对于GUI、DB和Web应用而言，构造单元测试比较困难，如果强行构造单元测试，反而给维护带来额外的工作量。
- 开发更为关注用例和测试案例，而不是设计本身。
- 测试驱动开发会导致单元测试的覆盖度不够，可能缺乏边界测试。当代码完成以后还是需要补充单元测试，提高测试的覆盖度。



2. Fibonacci数列的TDD案例



Starting

- **We are driven by scenarios:**
 - “we want the system to be able to...”
- **What do we want the function to do?**
- **Write a class and place the testing code:**

```
public class TestFib extends TestCase {  
    public void testFibonacci() {  
        assertEquals(0, fib(0));  
    }  
}
```

TODO List

- fib(0) == 0

Make it compile

- **Add minimal code to make it compile**

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

- **Run the test code**
- **Green...**

```
public class TestFib extends  
    TestCase {  
    public void testFibonacci()  
    {  
        assertEquals(0, fib(0));  
    }  
}
```

Add another test

- **Add another test method:**

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

- **There is no value to writing testFibonacciOfOneIsOne.**

- **We settle for**

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

- **Run the test code**
- **Red...**

TODO List

- ~~fib(0) == 0~~
- fib(1) == 1

Make it Green

- Add minimal test to make it green
- A chosen alternative: treat 0 as a special case

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

- Run the test code
- Green...

TODO List

- ~~fib(0) == 0~~
- fib(1) == 1

Remove Duplications

- This time duplications are in the test (not in the code)
- It will only get worse as we add new cases
- Remove them (refactoring)

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]));  
}
```

- Run the test code (to make sure we did not ruin anything)
- **Green...**

And another one

- To make sure we are done..

```
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 the test code
- **RED!** (good to have a bug)

TODO List

- ~~fib(0) == 0~~
- ~~fib(1) == 1~~
- ~~fib(2) == 1~~
- fib(3) == 2

Make it Green

- Add minimal test to make it green
- The chosen alternative:
 - Applying the same strategy as before (treating smaller inputs as special cases)

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

- Run the test code
- Green...


TODO List

- ~~fib(0) == 0~~
- ~~fib(1) == 1~~
- ~~fib(2) == 1~~
- fib(3) == 2

Refactoring


- Where did we get the 2 (return 2) from?

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



fib(n-1)

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



fib(n-2)

Refactoring

- **Now we have**

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

- Generalize for n=2

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

- **We are done!**



3. 保齡球比賽積分的TDD案例



保龄球比赛积分的TDD案例

- <http://www.objectmentor.com/resources/articles/xpepisode.htm>
- 一个极短的讨论，确定需要写一个保龄球计分程序，顺便画了一个简单的类图，把验收测试单也画了出来。
- 在编码期间，他们在不停的寻找对象和方法的蛛丝马迹，不是靠想，而是靠代码和测试进行尝试。
 - 对象总是在测试中创建，测试中修改，相对来说测试的修改就少了不少；
 - 在思考了实际使用后，测试也会做一些调整，构想对象不同使用方式，使得测试不停的增加；
 - 新增加的测试如果无法通过，又促使对象代码的修改；
 - 如果对象代码变得冗长，方法变了味道，发现了便重构，重构后往往会带来对象和方法修改的迹象甚至建立新对象和新方法。
- 如此反复直到大家都觉得程序的意思很清楚，测试很全面且正常，代码很美观。



結束

2018年3月5日