Test driven development

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

October 2, 2015

Overview

How to write functions using test driven development (TDD)

TDD requires developers to create automated unit tests that clarify code requirements before writing the code itself.
When you create a new function (f), follow TDD steps:

- Add a test
  - **Define a test function (test_f()) which contains test cases written using assertions.**
  - Concentrate on the **specification of f**.
  - **Define f: name, parameters, precondition, post-condition, and an empty body.**
- Run all tests and see if the new one fails
  - Your program may have many functions, so **many test functions**.
  - At this stage, ensure the new **test_f() fails**, while **other test functions pass** (written previously).

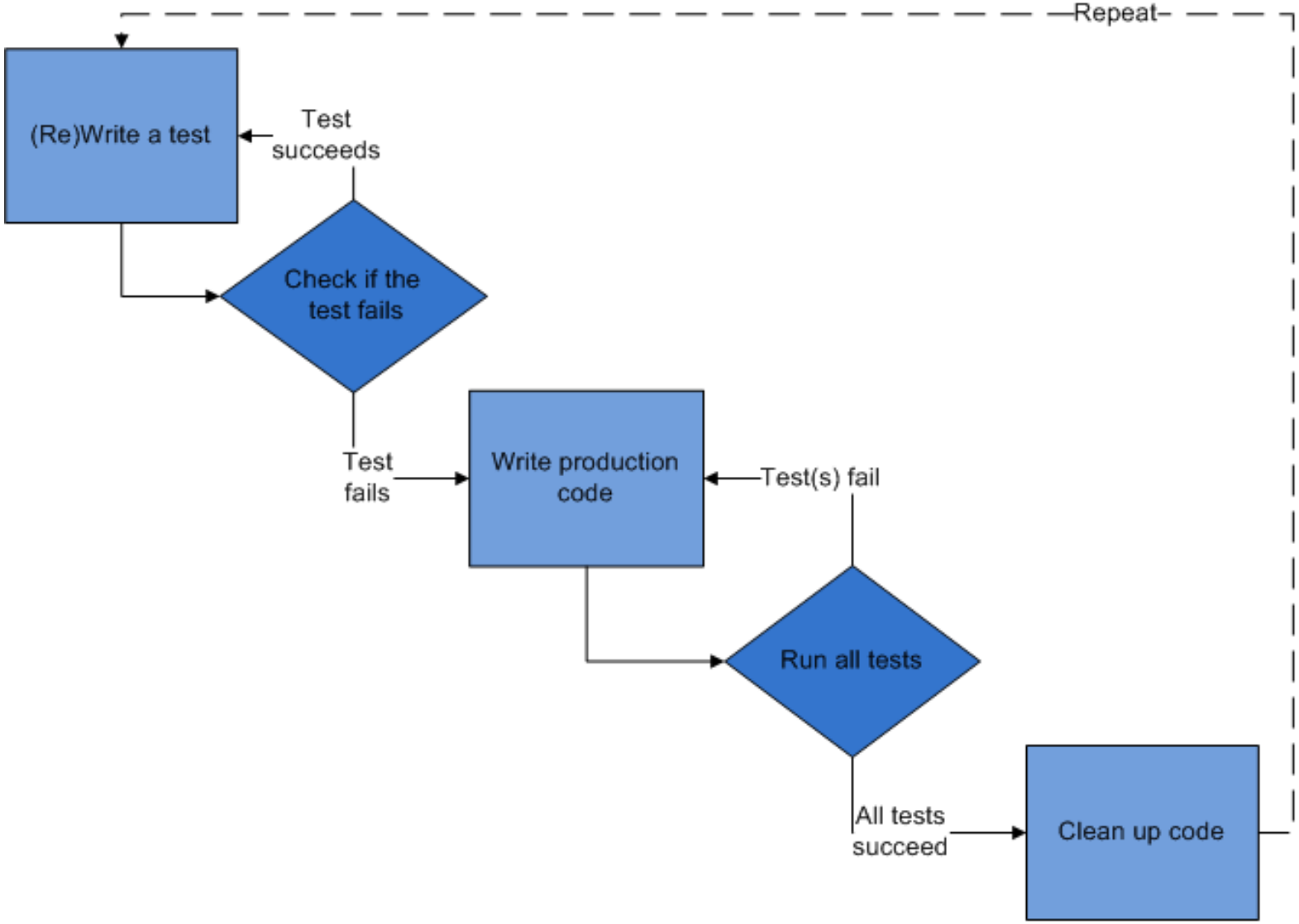
How to write functions using test driven development (TDD)

- **Write the body of f**
  - Now the specification of f is well written and you concentrate on **implementing the function according to pre/post-conditions** and on **passing all test cases written for f**.
  - **Do not concentrate on technical aspects (duplicated code, optimizations, etc).**
- Run all tests and see them succeed
  - Now, the developer is confident that the function meets the specification.
  - The final step of the cycle can be performed.
- Refactor code
  - Finally, you must clean up the code using refactoring techniques.

Test Driven Development (TDD)

- Steps to apply TDD:
  1. Create automated test cases
  2. Run the test (will fail)
  3. Write the minimum amount of code to pass that test
  4. Run the test (will succeed)
  5. Refactor the code

Test Driven Development (TDD)



Step 1 - Create automated test cases

- When you work on a task (work-item) start by creating a test function
- Work item: Compute the greatest common divider

```
def test_gcd():
    """
      test function for gdc
    """
    assert gcd(0, 2) == 2
    assert gcd(2, 0) == 2
    assert gcd(2, 3) == 1
    assert gcd(2, 4) == 2
    assert gcd(6, 4) == 2
    assert gcd(24, 9) == 3
```

Step 1 - Create automated test cases

- Concentrate on the specification of f.

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers, a>=0; b>=0
    return an integer number, the  greatest common divisor of a and b
    """
    pass
```

Step 2 - Run the test (will fail)


```
#run the test - invoke the test function
test_gcd()
```

```
Traceback (most recent call last):
  File "C:/curs/lect3/tdd.py", line 20, in <module>  test_gcd()
  File "C:/curs/lect3/tdd.py", line 13, in test_gcd
    assert gcd(0, 2) == 2
AssertionError
```

- Validates that the test function is working correctly and that the new test does not mistakenly pass without requiring any new code.
- It rules out the possibility that the new test will always pass, and therefore be worthless

Step 3 - Write the minimum amount of code to pass that test

- Concentrate on implementing the function according to pre/post-conditions and on passing all test cases
- Do not concentrate on technical aspects (duplicated code, optimizations, etc).

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers, a>=0; b>=0
    return an integer number, the  greatest common divisor of a and b
    """
    if a == 0:
        return b
    if b == 0:
        return a
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Step 4 - Run the test (will succeed)


```
>>> test_gcd()
>>>
```

If all test cases pass, the programmer can be confident that the code meets all the tested requirements.

Step 5 - Refactor the code

Clean up the code using refactoring techniques. **How?**

Refactoring

- **Code refactoring** is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" [5].
- **Code smell** is any symptom in the source code of a program that possibly indicates a deeper problem:
  - **Duplicated code**: identical or very similar code exists in more than one location.
  - **Long method**: a method, function, or procedure that has grown too large.

Refactoring: Rename method/variable

Rename a variable or a method name to something meaningful

```
def verify(k):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    l = 2
    while l<k and k % l>0:
        l=l+1
    return l>=k
```

```
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    div = 2  #search for divider
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the
    # number itself than nr is prime
    return div>=nr;
```

Refactoring: Extract Method

- You have a code fragment that can be grouped together.
- Turn the fragment into a method whose name explains the purpose of the method.

Refactoring: Extract Method


```
def startUI():
    list=[]
    print list
    #read user command
    menu = """
           Enter command:
              1-add element
              0-exit
           """
    print(menu)
    cmd=input("")
    while cmd!=0:
        if cmd==1:
            nr=input("Give element:")
            add(list, nr)
            print list
        #read user command
        menu = """
            Enter command:
               1-add element
               0-exit
            """
        print(menu)
        cmd=input("")

startUI()
```

```
def getUserCommand():
    """
     Print the application menu
     return the selected menu
    """
    menu = """
           Enter command:
              1-add element
              0-exit
           """
    print(menu)
    cmd=input("")
    return cmd

def startUI():
    list=[]
    print list
    cmd=getUserCommand()
    while cmd!=0:
        if cmd==1:
            nr=input("Give element:")
            add(list, nr)
            print list
        cmd=getUserCommand()

startUI()
```

Refactoring: Substitute Algorithm

- You want to replace an algorithm with one that is clearer.
- Replace the body of the method with the new algorithm.

```
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    div = 2  #search for divider
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the
    # number itself than nr is prime
    return div>=nr;
```

```
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    for div in range(2,nr):
        if nr%div == 0:
            return False
    return True
```

Calculator - procedural version

References

1. *The Python language reference. http://docs.python.org/py3k/reference/index.html*
2. *The Python standard library. http://docs.python.org/py3k/library/index.html*
3. *The Python tutorial. http://docs.python.org/tutorial/index .html*
4. Kent Beck. *Test Driven Development: By Example. Addison-Wesley Longman, 2002*. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development
5. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also http://refactoring.com/catalog/index. html