Program Testing. Refactoring.

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

September 19, 2015

Overview

Program testing

- **Testing** is observing the behavior of a program in many executions.
- Execute the program for some input data and observe if the results are correct for these inputs.
- Testing does not prove program correctness (only give us some confidence). On the contrary, it may prove its incorrectness if one execution give wrong results.
- Testing can never completely identify all the defects within software.

Testing methods

Exhaustive testing

- Check the program for all possible inputs.
- Impractical so we need to choose a finite number of test cases.

Black box testing

- The selection of input data for testing is decided by analyzing the specification.
- Distinct cases of the problem are decided and we use a test input data for each case

White box testing

- Select the test data by analyzing the text of the program. We select test data such that all the execution paths are covered.
- We test a function such that each statement is executed.

White box vs Black Box testing


```
def isPrime(nr):
    """
      Verify if a number is prime
      return True if nr is prime False if not
      raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

White box vs Black Box testing

Black Box

- test case for a prime/not prime
- test case for 0
- test case for negative number

```
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

White Box (cover all the paths)

- test case for 0
- test case fot negative
- test case for 1
- test case 3
- test case for prime (no divider)
- test case for not prime

```
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Unit testing

- Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level.
- Testing unit of code in isolation (functions). Test small parts of the program independently

Integration testing

- Considers the way program works as a whole.
- After all modules have been tested and corrected we need to verify the overall behavior of the program

Automated testing

Automated testing

- Test automation is the process of writing a computer program to do testing that would otherwise need to be done manually.
- Use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions

Test Driven Development (TDD)

TDD Steps:

1. Automated test cases
2. Writing specification (invariants, pre/post)
3. Throwing or not exceptions
4. Production code

Python unit testing framework (PyUnit)

unittestmodule supports:

- Test automation
- Sharing of setup and shutdown code for tests
- Aggregation of tests into collections
- Independence of the tests from the reporting framework

Python unit testing framework (PyUnit)


```
import unittest
class TestCase(unittest.TestCase):
    def setUp(self):
        #code executed before every testMethod
        val=StudentValidator()
        self.ctr=StudentController(val, StudentRepository())
        st = self.ctr.create("1", "Ion", "Adr", 1, "Cluj")

    def tearDown(self):
        #cleanup code executed after every testMethod

    def testCreate(self):
        self.assertTrue(self.ctr.getNrStudents()==1)
        #test for an invalid student
        self.assertRaises(ValidationEx,self.ctr.create,"1", "", "", 1, "Cj")

        #test for duplicated id
        self.assertRaises(DuplicatedIDException,self.ctr.create,"1", "I",
                                                                "A", 1, "j")
```

Python unit testing framework (PyUnit)


```
    def testRemove(self):
        #test for an invalid id
        self.assertRaises(ValueError,self.ctr.remove,"2")

        self.assertTrue(self.ctr.getNrStudents()==1)

        st = self.ctr.remove("1")
        self.assertTrue(self.ctr.getNrStudents()==0)
        self.assertEquals(st.getId(),"1")
        self.assertTrue(st.getName()=="Ion")
        self.assertTrue(st.getAdr().getStreet()=="Adr")


if __name__ == '__main__':
    unittest.main()
```

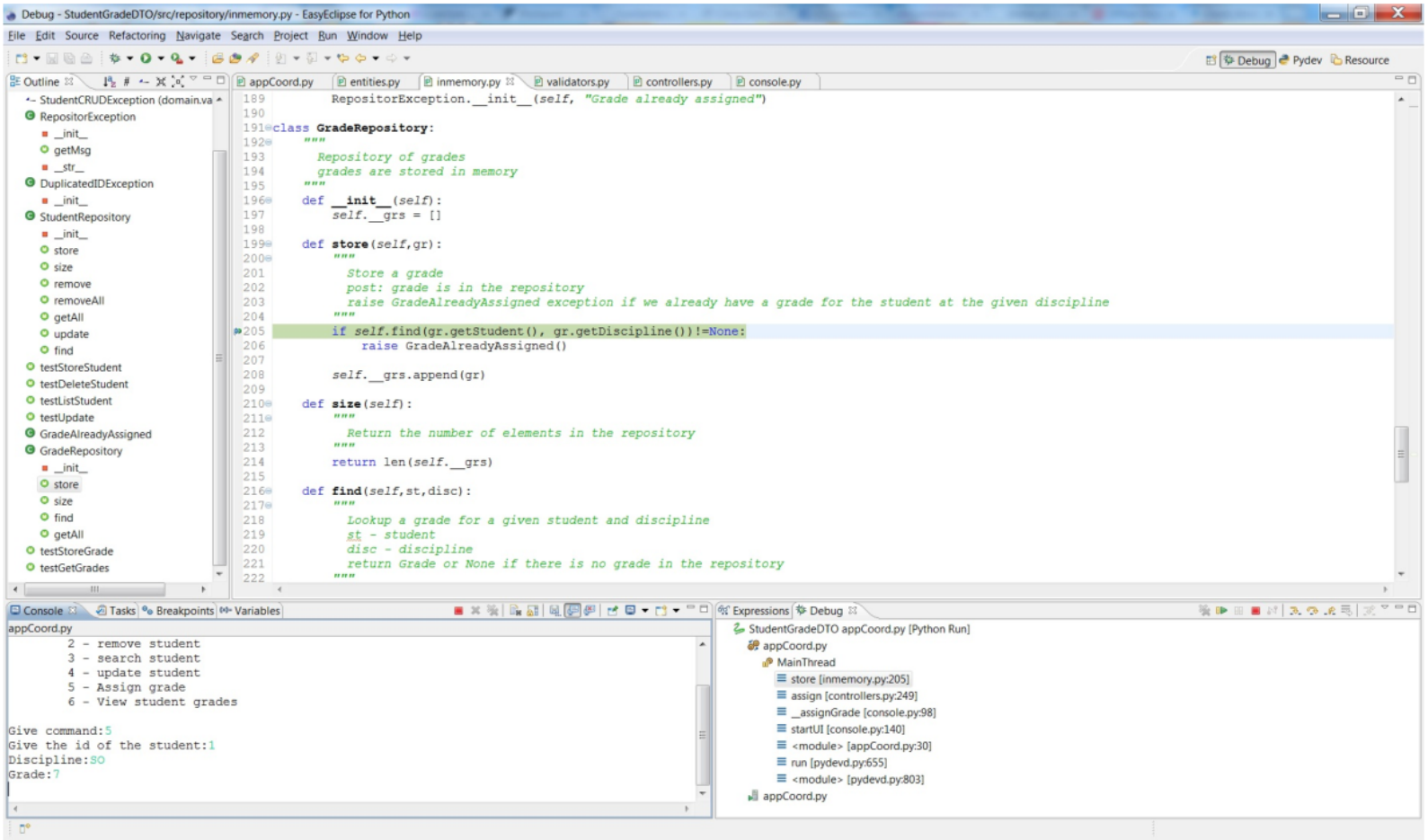
Debugging

Debugging - the activity that must be performed when testing indicates the presence of errors, to find errors, and rewrite the program with the purpose of eliminating the errors.

- Using print statements
- Using the IDE

Debugging is the most unpleasant activity. Debugging must be avoided!

Eclipse debug perspective - Example



Eclipse debug perspective

Debug view

- View the current execution trace (stack trace)
- Execute step by step, resume/pause execution

Variables view

- View variable values

Program inspection

- Anyone can write code that a computer can understand. Good programmers
- Write code that humans can understand!
- Programming style consist of all the activities made by a programmer for producing products easy to read, and easy to understand, and the way in which these qualities are achieved

Program inspection

- Readability is considered the main attribute of style.
- A program, like any publication, is a text must be read and understand by another programmer. The element of coding style are:
  - Comments
  - Text formatting (indentation, white spaces)
  - Specification
  - Good names for entities (classes, functions, variables) of the program
    - Meaningful names
    - Use naming conventions

Naming conventions

- Class names: Student, StudentRepository
- Variable names: student, nrElem (nr_elem)
- Function names: getName, getAddress, storeStudent (get_name,get_address, store_student)
- constants: MAX

Whatever convention you use, use it **consistently**.

Refactoring

- Refactoring is the process of changing the software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
- It is a disciplined way to clean up the code that minimizes the chances of introducing bugs.
- When you need to add a new feature to the program, and the program's code is not structured in a convenient way for adding the new feature, first refactor the code to make it easy to add a feature, then add the feature

Why refactoring

- Refactoring improves the design of the software
- Refactoring makes software easier to understand
- Refactoring helps you find bugs
- Refactoring helps you program faster

Bad smells

When we need to refactor the code

- Duplicated code
- Long method
- Large class
- Long parameter list
- Comments
- Divergent change - One class is commonly changed in different ways for different reasons

Refactoring methods

- **Rename Method** - *The name of a method does not reveal its purpose*.
- **Consolidate Conditional Expression** - *You have a sequence of conditional tests with the same result*. Combine them into a single conditional expression and extract it.
- **Consolidate Duplicate Conditional Fragments** - *The same fragment of code is in all branches of a conditional expression*. Move it outside of the expression.

Refactoring methods

- **Decompose Conditional** - *You have a complicated conditional (if-then-else) statement*. Extract methods from the condition, then part, and else parts.
- **Inline Temp** - *You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings*. Replace all references to that temp with the expression.
- **Introduce Explaining Variable** - *You have a complicated expression*. Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

Refactoring methods

- **Remove Assignments to Parameters** - *The code assigns to a parameter*. Use a temporary variable instead.
- **Remove Control Flag** - *You have a variable that is acting as a control flag for a series of boolean expressions*. Use a break or return instead.
- **Remove Double Negative** - *You have a double negative conditional*. Make it a single positive conditional

Refactoring methods

- **Replace Nested Conditional with Guard Clauses** - *A method has conditional behavior that does not make clear what the normal path of execution is*. Use Guard Clauses for all the special cases.
- **Replace Temp with Query** - *You are using a temporary variable to hold the result of an expression*. Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

Refactoring classes

- **Encapsulate Field** - *There is a public field*. Make it private and provide accessors.
- **Replace Magic Number with Symbolic Constant** - *You have a literal number with a particular meaning*. Create a constant, name it after the meaning, and replace the number with it.
- **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 classes

- **Move Method** - *A method is, or will be, using or used by more features of another class than the class on which it is defined.* Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
- **Move Field** - *A field is, or will be, used by another class more than the class on which it is defined.* Create a new field in the target class, and change all its users.