Testing. Refactoring

Assoc. Prof. Arthur Molnar

Babeș-Bolyai University

2024

Overview

Program testing

What is testing?

Testing is observing the behavior of a program over many executions.

Program testing

- We execute the program for some input data and compare the result we obtain with the known correct result.
- **Questions**:
  - How do we choose input data?
  - How do we know we have run enough tests?
  - How do we know the program worked correctly for a given test? (known as the oracle problem)

Program testing

- Testing cannot prove program correctness, and cannot identify all defects in software. However, what it **can** prove is incorrectness, if at least one test case gives wrong results.
- **Problems with testing**
  - We cannot cover a function's input space
  - We have to design an oracle as complex as the program under test
  - Certain things are practically outside of our control (e.g. platform, operating system and library versions, possible hardware faults)

Testing Approaches

Exhaustive testing

- Check the program for all possible inputs.
- Impractical for all but mostly trivial functions.
- Sometimes used with more advanced techniques (e.g. symbolic execution) for testing small, but crucial sections of a program (e.g. an operating system's network stack)

Testing Approaches

Boundary value testing

- Test cases use the extremes of the domain of input values, typical values, extremes (inside and outside the domain).
- The idea is that most functions work the same way for most possible inputs, and to find most of those possibilities where functions use different code paths.

Testing Approaches

Random testing, pairwise (combinatorial) testing, equivalence partitioning

- And the list goes on...

Testing Methods

Black box testing

- The source code is not available (it is in a "black", non-transparent box)
- The selection of test case data for testing is decided by analyzing the specification.

White box testing

- The source code is readily available (it is in a transparent box) and can be consulted when writing test cases.
- Selecting test case data is done by analyzing program source code. We select test data such that all code, or all execution paths are covered.
- When we say *"have 95% code coverage"* (assignment bonus) it is white-box testing.

Demo

White and Black-box testing

Examine the test code in lecture.examples.ex40_black_box_white_box.py

Advantages and drawbacks

Black box testing

+ Efficient for large code-bases
+ Access to source code is not required
+ Separation between the programmer's and the tester's viewpoint
– You do not know how the code was written, so test coverage might be low, testing might be inefficient

Advantages and drawbacks

White box testing

- + Knowing about the code makes writing it **AND** testing it easier
- + Can help find hidden defects or to optimize code
- + Easier to obtain high coverage
- – Problems with code that is completely missing
- – Requires access to source code
- – Requires good knowledge of source code

White and Black-box testing

NB!

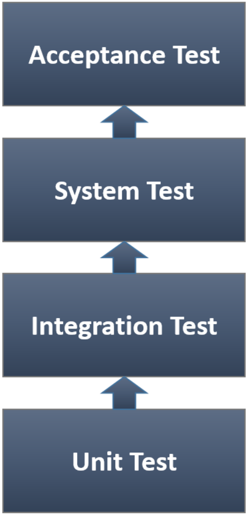
It's not a matter of which box is better, it's more like you have to make do with what you've got!

Testing levels

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

Testing levels



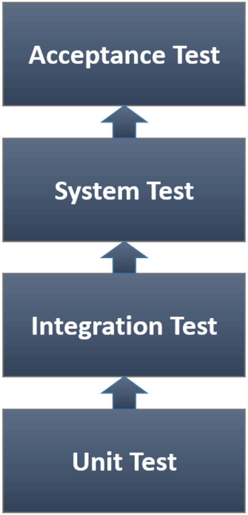
Unit Test

- Refers to tests that verify the functionality of a specific section of code, usually at function level.
- Testing is done in isolation. Test small parts of the program independently

Integration Test

- Test different parts of the system in combination
- In a bottom-up approach, it is based on the results of unit testing.

Testing levels



System Test

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

Acceptance Test

- Check that the system complies with user requirements and is ready for use

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, comparison of actual outcomes to predicted outcomes, setting up test preconditions

PyUnit - Python unit testing framework

The unittest[1] module supports:

- Test automation
- Sharing setup and shutdown code for tests
- Aggregation of tests into collections
- Independence of tests from the reporting framework (another instance of the *single responsibility principle*)

[1] `https://docs.python.org/3/library/unittest.html`

Demo

PyUnit

Run the unit test in lecture.examples.ex41_pyunit.py in an IDE that supports this (e.g. PyCharm)
NB! This has to be run as a unit-test, and not a regular Python program

PyUnit - Python unit testing framework

The unittest module supports:

- Tests are implemented using classes derived from **unittest.TestCase**
- Test methods should start with the characters **test**
- We now use special methods instead of **assert** statements directly - assertTrue(), assertEqual(), assertRaises() and many more[2]
- The setUp() and tearDown() methods are run before and after each test method, respectively.

[2] `https://docs.python.org/3/library/unittest.html#assert-methods`

Automated testing

Discussion

How can we know when our test are "good enough" ?

The Coverage module

One (of the simpler) ways is to use code coverage

- Measure how much of the entire code was executed during the tests
- 0% coverage means no lines of code were executed
- 100% means **ALL** lines of code were executed at least once
- There exist tools which can measure and report this automatically

The coverage module

- PyCharm Professional can be used to gather coverage information by installing the *coverage*[3] module.

[3] `https://coverage.readthedocs.io/en/coverage-5.3/`

The coverage module

... or we can use it in command line

1. pip install coverage # installs the **coverage.py** module
2. open a **cmd/terminal** into your project's folder
3. **coverage** run -m unittest discover -p *.py && **coverage** report[4]
4. **coverage** html produces pretty printed output

[4] `https://stackoverflow.com/questions/47497001/python-unit-test-coverage-for-multiple-modules`

Test Driven Development Steps

Test Driven Development (TDD)

TDD requires developers to create automated unit tests that clarify code requirements before writing the code.

- Steps to apply TDD[5]:
  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

[5] Kent Beck. *Test Driven Development: By Example. Addison-Wesley Longman, 2002*. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development

Writing functions for TDD

1. Create 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.

Writing functions for TDD

2. Run all tests and see that the new one fails
    - Your program has many functions, so it will also have many **test functions**
    - At this stage, ensure the new test_f() fails, while previously written test function pass
    - This shows that the test is actually executed and that it tests the correct function

Writing functions for TDD

3. Write the body of function f()
    - Writing the test before the function obliged you to clarify its specification
    - Now you concentrate on correctly implementing the function code
    - At this point, do not concentrate on technical aspects such as duplicated code or optimizations

Writing functions for TDD

4. Run all tests and see them succeed
   - Re-run the test you created at step 1
   - Now, you can be confident that the function meets its specification

Writing functions for TDD

5. Refactor code
    - **Code refactoring** is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"[6].
    - **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.

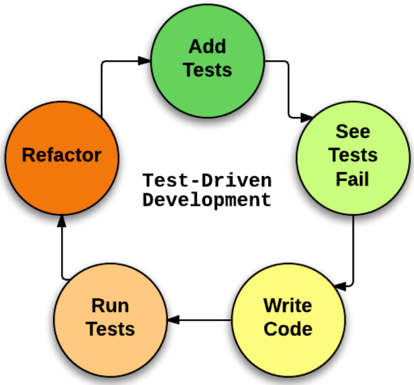
[6] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also http:// refactoring.com/catalog/index. html

Writing functions for TDD

Discussion

How do I know my tests are good enough?

Test Driven Development (TDD)



[7]

[7] http://joshldavis.com/2013/05/27/difference-between-tdd-and-bdd/

Demo

Test Driven Development

Check out the examples in lecture.examples.ex42_tdd_1.py and lecture.examples.ex43_tdd_2.py

Thoughts on TDD

- TDD is designed to take you out of the mindset of writing code first, and thinking later
- It forces you to **think** what each part of the program has to do
- It makes you analyse boundary behaviour, how to handle invalid parameters before writing any code

Program inspection

- Anyone can write code that computers understand. It's about writing code that humans also understand!
- Programming style consist of all the activities made by a programmer for producing code easy to read, 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 understood 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

- Specific to each language, for Python they are encoded in the PEP-0008[8]
- Class names use camel case notation: Student, StudentRepository
- Variable names: student, nr_elem
- Function names: get_name, get_address, store_student
- constants are capitalized: MAX_LENGTH

[8] https://www.python.org/dev/peps/pep-0008

Refactoring

Refactoring

The process of changing the software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.

- It is a disciplined way to clean up 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

- Improves the design of the software
- Makes software easier to understand
- Helps you find bugs
- Helps you program faster

Bad smells

When is refactoring needed?

- Duplicated code
- Long method/class
- Long parameter list (more than 3 parameters is seen as unacceptable)
- Comments

Sample code to refactor

The following file contains some examples of code that is good candidate for refactoring lecture.examples.ex44_refactoring.py

Refactoring methods

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

Refactoring methods

4. **Decompose Conditional** - *You have a complicated conditional (if-then-else) statement*. Extract methods from the condition, then part, and else parts.
5. **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.
6. **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

7. **Remove Assignments to Parameters** - *The code assigns to a parameter*. Use a temporary variable instead.
8. **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.
9. **Remove Double Negative** - *You have a double negative conditional*. Make it a single positive conditional

Refactoring methods

10. **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.
11. **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

12. **Encapsulate Field** - *There is a public field*. Make it private and provide accessors.
13. **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.
14. **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

15. **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.
16. **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.