Software Development: Testing

CPSC 233: Introduction to Computer Science for Computer Science Majors II
Winter 2022

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Wednesday, 17 November 2021

UNIVERSITY OF CALGARY

Importance of Testing

- in large complex systems, **50%** of the systems development budget may be spent on testing
- this time should be reduced with modern design techniques on less complex systems, but it is still **very high.**
- Studies have shown that **virtually all non-trivial** software ships with errors!
- Thus, good testing is as important **(more?)** than programming

Psychological Problem of Acceptance of Testing

- we think if we're good, there will be no bugs. Therefore finding errors shows incompetence - who wants that?

- BUT everyone writes code with bugs

- Good programs have approximately 1 bug per 100 lines. So take the attitude that the more bugs you find, the BETTER tester you are.

UNIVERSITY OF CALGARY

When to Test

- **Throughout** the development lifecycle, not just at the end.

- **earlier** you find error **the better,** so test the design before coding ---> prevents errors

- Benefits:
  - **require less testing & debugging time**
  - **cost less**

Who should Test

- developers
  - know code so can be more efficient (e.g. no 2 tests which test exactly same stuff)
  - but have blind spots (i.e. if didn't realize system has to do X, will never test for that)
- **professional testers** (Q/A department)
  - include people from user department to test functionality

- Note - need both - they have different mind sets.
  - Programmer - **hopes not** to find bugs
  - Tester - **aggressively looking** for bugs (programmers will not like you)

How to Test

- **Exhaustive testing** (testing every possible input), would be ideal, but clearly impossible

- Instead, a **methodical approach** to testing is used: try to develop test cases to “cover all the bases”.

UNIVERSITY OF CALGARY

Black Box Testing

Black Box Testing

- assumes you **know nothing** of the internals of a program
- tests functionality
  - i. e. checks that program **satisfies requirement specifications**

---> checks for "blind spots" on part of designer.

- consider all types of input
- for each, divide it into **equivalence classes** - all data in each class is "equivalent" to each other
- Then choose test data such that at least one piece of data for each equivalence class is included.

UNIVERSITY OF CALGARY

Black Box Testing (cont'd)

- you must look at the positive cases (you expect program will work), as well as negative cases (you expect these to fail). **Junior programmers often are weak at testing all the negative cases**

e.g. Size of array to store courses enrolled in - # courses

Valid	Invalid
1 <= courses <= 5	< 1
	> 5

- Therefore need to include data from 3 test areas
  - Note: 8 is equivalent to 12 - i.e. if one handled properly other will be

Black box Testing (cont'd)

- BUT boundaries more likely to have errors than inside therefore:

  test e.g. 4, 5, 6.

  (Note - strictly speaking, this goes beyond black box testing -you might call it "grey box" testing - but because of the frequency of boundary errors, these extra tests should be included).

UNIVERSITY OF CALGARY

Equivalence Classes

Equivalence classes of test data

Partition possible input (and states) into categories

These categories are also known as ***equivalence classes***

- Test at least <u>one</u> data set from each class

UNIVERSITY OF CALGARY

12

Equivalence classes of test data

e.g.

```
If (n > 4 and n < 10) {
  //Do something
} else{
  //Do something else
{
```


13

Equivalence classes of test data

- in this case there are three equivalence classes:

We might choose 5 tests:

< 4 = 4 5 .. 9 = 10 > 10

14

White Box Testing

White Box Testing

- **look inside** at details of program to determine what to test, analyzing the flow of control
- **based on coverage testing**. The various test cases must "cover" the entire source code:
  - **ensure all statements are executed**
  - **ensure all expressions are evaluated**
  - **various paths** through the code must be considered

UNIVERSITY OF CALGARY

White Box Testing

- **look inside** at details of program to determine what to test, analyzing the flow of control
- **based on coverage testing**. The various test cases must “cover” the entire source code:
  - **ensure all statements are executed (weak) Statement Testing**
  - **ensure all choices/branches are evaluated (stronger) Conditional Testing**
  - **various paths** through the code must be considered (strongest) **(Path Testing)**

Testing and Debugging

Definition of Testing

testing = the process of detecting run-time errors (*bugs)* in code and
evaluating the functionality of the code (≈ logic errors)

- testing can tell you that you have bugs
- but it does not prove you don't have bugs

UNIVERSITY OF CALGARY

19

Debugging

- Some techniques for locating bugs:

1. use “trace messages” – print statements saying where you are, and values of some variables (most programmers start here)
2. use a debugger (built into IDEs, BREAKPOINTS!!!)
3. use “assertions” – statements that say what should be the case – if it is not true, program automatically gives error. Tool used in automatic testing (have to enable to have them run)

Debugging

- Testing helps discover bugs, i.e. you may know a bug exists, but not much more.

- You must also:
  - locate the error
  - explain the error's cause -> scientific hypothesis
  - correct the error -> scientific experiment
  - re-test -> analysis of experiment

UNIVERSITY OF CALGARY

Debugging

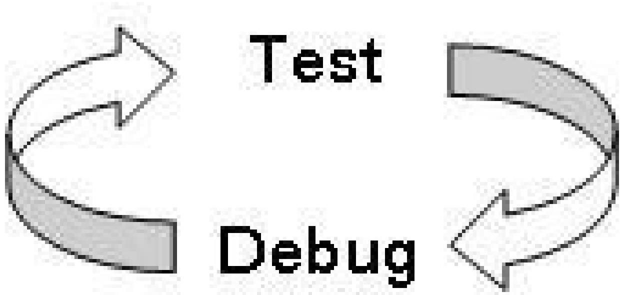
Note:

- The location of the error may not be the statement at which it manifests itself (e.g. if you return the pointer to a class variable, rather than a copy of that object, you won't get incorrect values until later in the program).
- A bug can be:
  - a simple programmer error (more easily fixed)
  - a design error (less easily fixed)

Debugging ≠ Testing

debugging = the process of correcting errors

- testing and debugging are cyclic

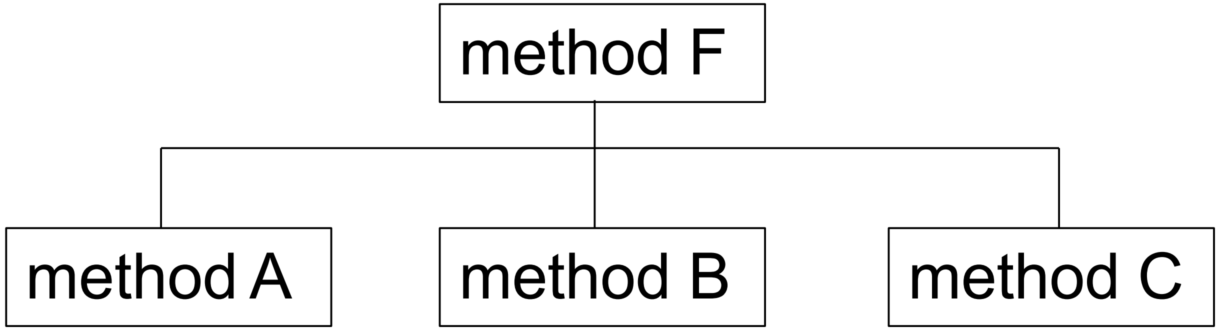


Modular Testing

Modular Testing

- if you write whole program and test it, and it doesn't work (e.g. infinite loop) very hard to find error
- **better to test each module separately** ---> much smaller bit of code to examine to find error.
- Most important concept: test each module individually as you implement!

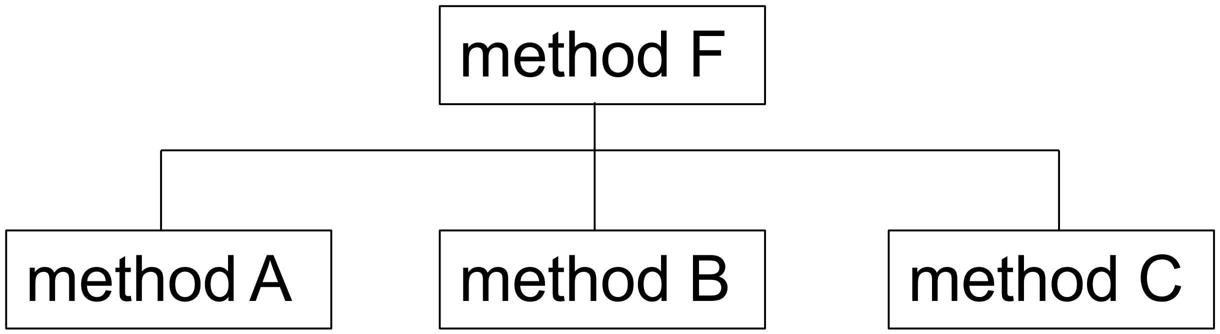
Modular Testing (cont'd)



- Test & debug method A.
- Test & debug method B.
- Test & debug method C.
- Finally, test method F.
- If it fails the testing then you can be (mostly) sure that the error is in F, and not a sub-method.

26

Modular Testing (cont'd)



- Test & debug method A.   (unit test)
- Test & debug method B. (unit test)
- Test & debug method C.   (unit test)
- Finally, test method F.   (integration test)
- If it fails the testing then you can be (mostly) sure that the error is in F, and not a sub-method.

Onward to … JUnit

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY