Lecture Notes
September 24, 2020

Working in a team: *finding bugs*

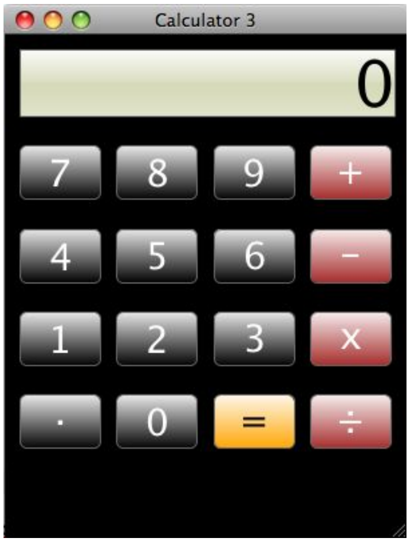
A large portion of this course is concerned with preventing, finding and fixing bugs, and defending against those you haven't yet found/fixed (security and reliability).

"Assignment I2"
https://courseworks2.columbia.edu/courses/104335/assignments/482608
focuses on finding bugs in your tic-tac-toe game, using both unit testing (JUnit) and a static analysis bug finder (SpotBugs). Both tools help find bugs in your code.

Bugs are not always in code. A bug could be in:

- Requirements
- Design
- Tests
- Documentation
- Configuration
- Environment
- User

Consider the following “bugs” in a simple calculator:



The tester enters a number, presses +, enters another number, presses =

- Nothing happens
- Computes the wrong answer

Calculator works correctly for some period of time or some number of computations, and then does nothing

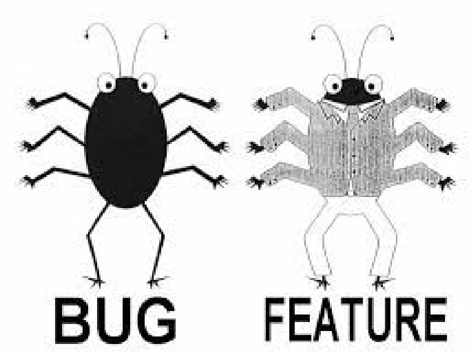
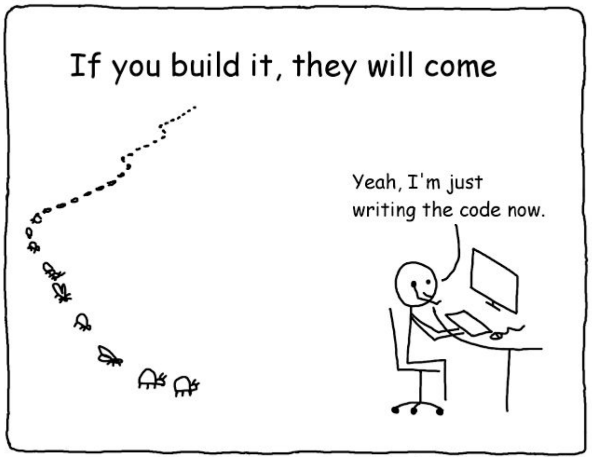
Calculator displays all 0’s and won’t do anything else

The calculator correctly adds, subtracts, multiplies and divides, but the tester found that if two operators are held down simultaneously, it appears to do square root

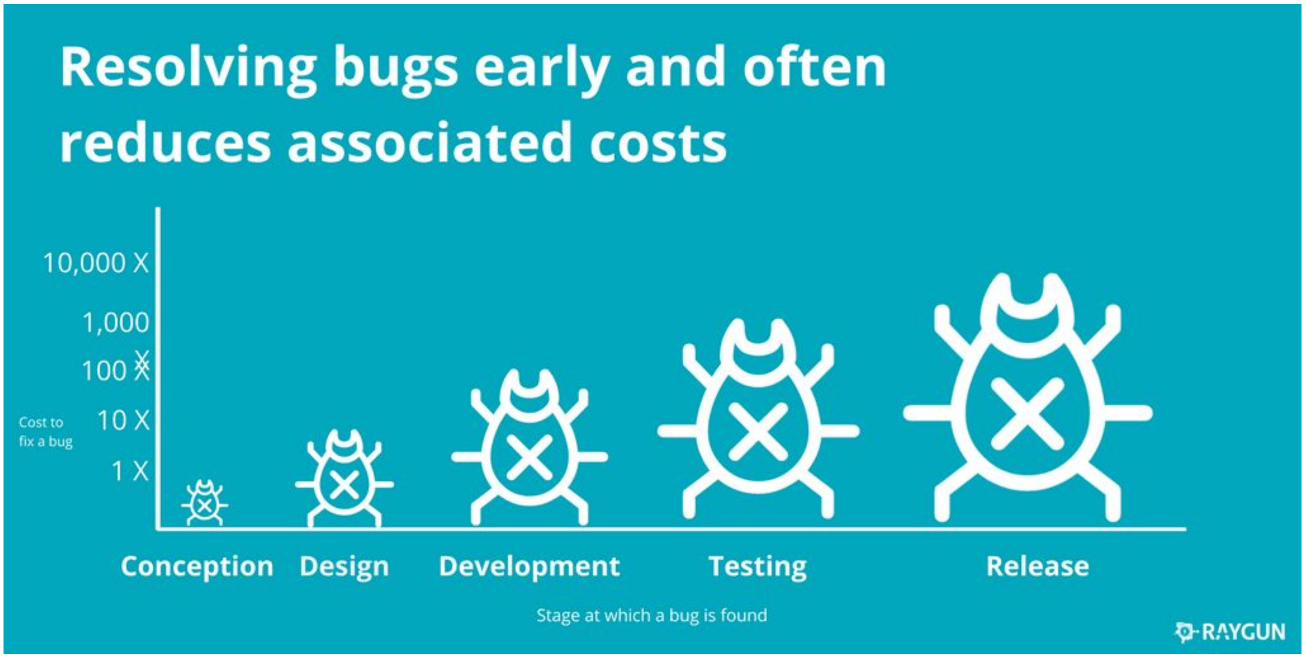
The calculator’s buttons are too small, the = key is in an odd place, the display is hard to read, …

Broad concept of what is a bug:

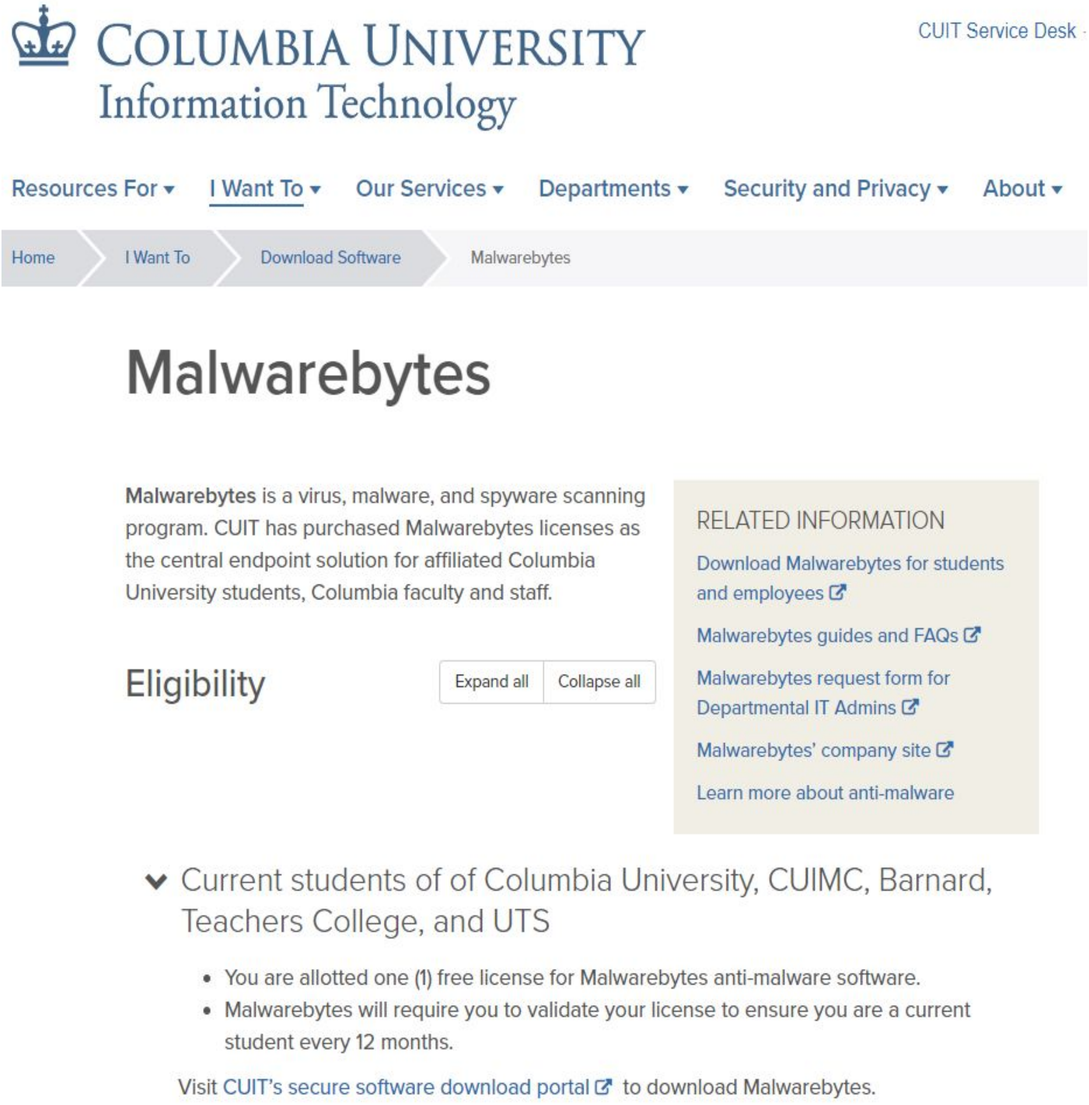
- Software doesn’t do something requirements say it should do
- Software does something requirements say it shouldn’t do
- Software does something that requirements don’t mention
- Software doesn’t do something that requirements don’t mention but should
- Software is difficult to understand, hard to use, slow, etc. - users will consider this a bug

	 If you build it, they will come Yeah, I'm just writing the code now. cartoontester.blogspot.com © 2013

How much do bugs cost, and at which phase of SDLC?



Example of a real bug:



Malwarebytes

Malwarebytes is a virus, malware, and spyware scanning program. CUIT has purchased Malwarebytes licenses as the central endpoint solution for affiliated Columbia University students, Columbia faculty and staff.

RELATED INFORMATION

- Download Malwarebytes for students and employees
- Malwarebytes guides and FAQs
- Malwarebytes request form for Departmental IT Admins
- Malwarebytes' company site
- Learn more about anti-malware

Eligibility

Current students of of Columbia University, CUIMC, Barnard, Teachers College, and UTS

- You are allotted one (1) free license for Malwarebytes anti-malware software.
- Malwarebytes will require you to validate your license to ensure you are a current student every 12 months.

Visit CUIT's secure software download portal to download Malwarebytes.

Malwarebytes Endpoint Security Premium costs $40/year paid by Columbia University. Access is linked to your UNI. If I can get 6 licences at the cost of one, someone is losing $200 in revenue/year/user. If 10 people manipulate the vulnerability, it will cost about $2000 in revenue/year. You can calculate the rest.

Many bugs can be found by *static analysis*, which examines the code to find “generic” bugs and suspicious code patterns that appear in many programs. The static analysis tool does not know what your program is supposed to do, so cannot tell if it’s doing it right.

```
public class JavaProgram {
  public Integer next() {
    for(int i = p.length - 1; i >= 0;
      if(++p[i] ... n)
        p[i] = new Integer(0);
      else
        return p;
    }
    throw new NoSuchElementException();
```

Static analysis sometimes produces *false positives*, reporting bugs that could never happen during actual execution because it considers all paths through the program - which may not all be feasible at runtime

Is this a false positive or a real bug?

```
function foo (int x) {
     if (x<0) { do something buggy }
     else { do something not buggy }
}

function bar (int y) {
     if (y<0) return;
     foo(y)
}
```

There are also “generic” *dynamic* analyzers that don’t know what your code is supposed to do but can still find some bugs - particularly crashing bugs

For example, fuzzing takes existing test cases and “fuzzes” the inputs with semi-random changes, seeking to produce inputs that are:

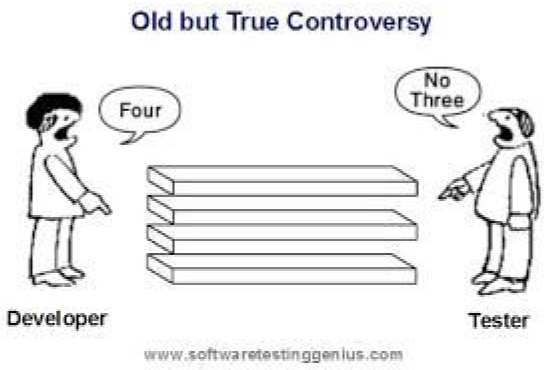
- structurally invalid
- structurally valid but semantically invalid

Example: Tictactoe game application [Postman & UI]

Both should be rejected by the application, via “input gatekeeper” code, but are often accepted (in itself a bug) and processed, which can trigger other bugs

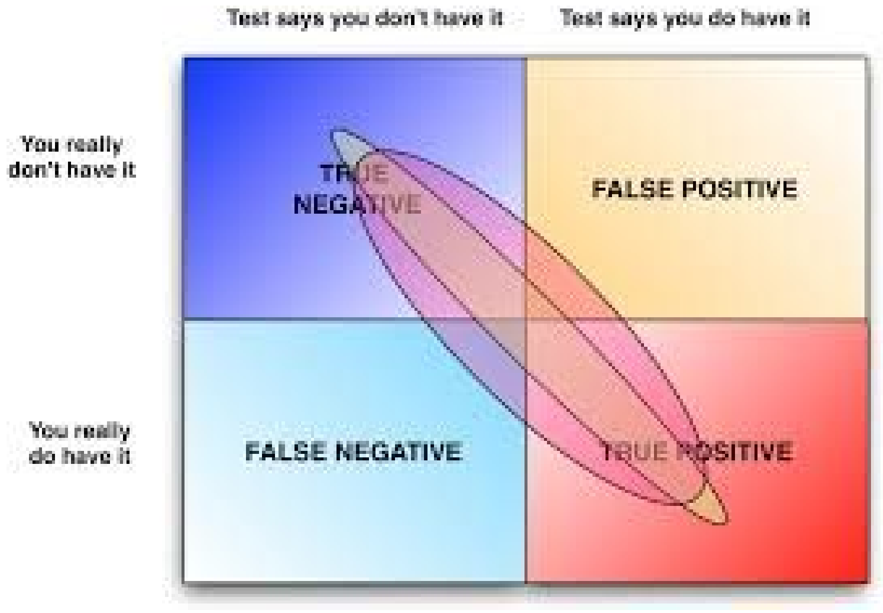
It is sometimes argued that the second case consists of false positives, because the invalid input would never have reached the internal code, beyond the gatekeeper, if it were not for the gatekeeper bug (*missing* code is a bug!)

To find application-specific bugs in your program, you also need to check the features, another form of dynamic analysis usually referred to as *software testing*



Bugs found during full program testing using valid inputs are never false positives, because not only *could* they happen during actual execution, they *did* happen

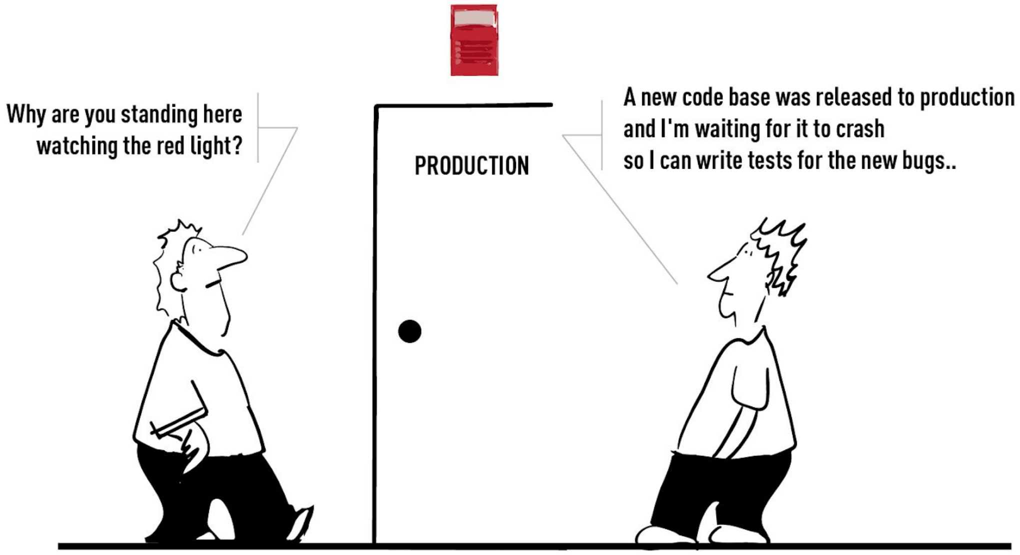
Both static and dynamic analysis exhibit *false negatives*, failing to report all bugs - there is <u>no</u> technique that can be guaranteed to find *all* bugs in a non-trivial program written in a conventional programming language. (Although it may be possible to prove certain properties.)



Test-to-pass: initial testing makes sure the software minimally works (“smoke test”)

Test-to-fail: most testing is trying to find bugs, so they can be fixed before deployment.

Much more expensive to fix a bug found by a user than to fix it before the user gets a chance to encounter it.


Why are you standing here watching the red light?
PRODUCTION
A new code base was released to production and I'm waiting for it to crash so I can write tests for the new bugs..

Functional testing: Checks that the code does what it is supposed to do - often divided into unit, integration, system testing

Non-functional testing: compatibility testing, performance testing, penetration and other security testing, fault injection, UI/UX testing, etc.

Unit testing - tests that individual methods, classes, modules, packages or other “units” do what they are supposed to do to fulfill their role in the design.

Unit tests aim to test a method or class *in isolation from the rest of the program*. They call the method, or a series of methods in the same class, from a test harness, not from the other code in the rest of the program.

When a unit itself would normally call other code or otherwise interact with the environment outside the unit, it instead calls stubs (state verification) or mocks (behavior verification). We will cover unit testing and stubs/mocks later in course.

Integration testing - tests boundaries between “units”, to check that the units do what they are supposed to do to fulfill their role in the design. Usually integrate only two, or a few, units at a time. So now when a unit would normally call other code, it indeed calls code in the selected other unit(s), but still uses stubs or mocks for all other code.

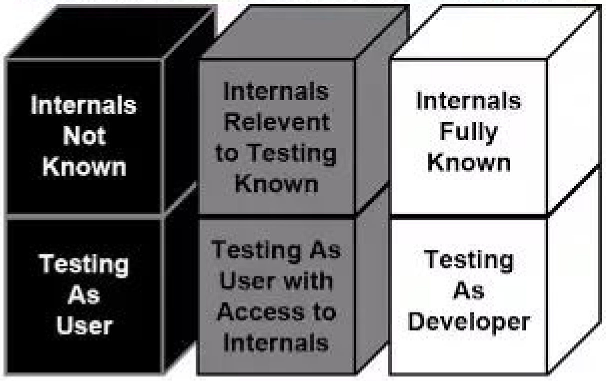
Integration testing may also refer to testing the full program’s interfaces to external libraries, systems, databases, etc.

System testing - aka end-to-end testing, tests the full system from external user-visible entry points, to check that the code fulfills the customer’s requirements - often done by independent testers, not the developers.

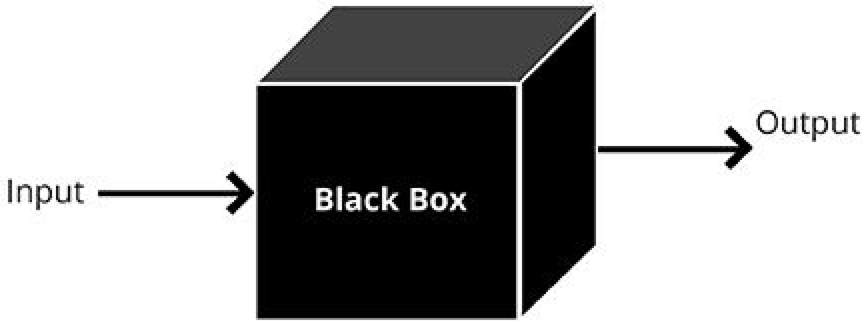
Acceptance testing and beta testing are special cases of system testing.

Functional and some non-functional testing divided into black box vs. white box testing, often also grey box

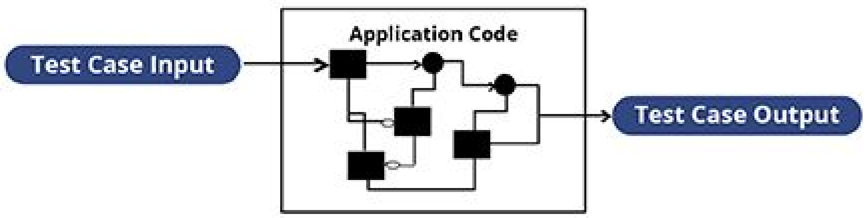
Differences Between Box Testing Types


Internals Not Known
Testing As User
Internals Relevent to Testing Known
Testing As User with Access to Internals
Internals Fully Known
Testing As Developer

BLACK BOX TESTING APPROACH



WHITE BOX TESTING APPROACH



Black box - tester considers what the box is supposed to do, not how it does it, doesn’t look inside the box

For example, for unit testing, the box might be a method. The tester provides input parameters and checks the outputs - the method might contain one line of code or might invoke a complicated graph of method calls, doesn’t matter for black box testing.

For integration testing, the tester considers two or more boxes and the possible communication paths between them - providing inputs and checking outputs on those paths

For system testing, the box is the full program. The tester provides external inputs and checks the externally visible outputs.

External inputs include error messages. Error messages (or lack thereof) are typically the last thing developers think about but the first thing users notice when software does not work as users expected.

White box - tester leverages full knowledge of what is inside the box.

Unit testing is often implicitly white box, since typically done by the same developers who coded the box.

White box is necessary to achieve *coverage* - track what code has and has not already been exercised by previous tests, then construct additional test inputs specifically to force execution of previously unexercised code

- ➢Coverage cannot detect *missing* code

Gray box (or grey box) - tester looks at intermediate products between boxes (e.g., network I/O) and leftover side-effects after box is done executing (e.g., temporary files, database connections left open)

For all blackbox and whitebox test cases, check any error status, exceptions, logs, etc. produced by one box that are visible to other boxes or to human users (or administrators) and external systems.

All these forms of testing will be discussed more deeply later in course.

“Assignment I2”
https://courseworks2.columbia.edu/courses/104335/assignments/482608
Assignment 2 builds on Assignment 1. Please make sure you have completed Assignment 1 before starting this assignment. In addition to using unit testing (JUnit), coverage tracking (Emma), and a static analysis bug finder (SpotBugs) to identify bugs in your application, you need to fix the bugs!

Shirish will now give a second Eclipse tutorial, focused on what you need to do for Assignment 2.

There will be one more assignment for this individual project, to add persistent storage of an in-progress game, with one more corresponding tutorial.

“Assignment T0”: https://courseworks2.columbia.edu/courses/104335/assignments/486855 This is the initial team assignment, to form a team and choose which programming language your team will use (C/C++, Java, Javascript, Python).

It’s not due until October 15, after the individual project and first assessment, but you should start forming teams.

There is a “search for teammates” thread in piazza that you can use to look for teammates https://courseworks2.columbia.edu/courses/104335/external_tools/1456

If time permits, at the end of each of the next few class sessions I will put everyone in breakout groups so you can introduce yourself and meet each other. Today (or next class if no time today) there is a short exercise to work on in your breakout groups, and then volunteers will report their group’s ideas to the class. Each breakout group should have its own zoom chat, so you can discuss via chat if you cannot use a microphone.

Recall the small drone that flies around an in-person classroom, just below the ceiling, and uses its sensors to determine which students are “paying attention”.

We previously considered how to do acceptance testing and beta testing assuming adult users. Today, consider how to do beta testing with live users in real ***middle school*** classrooms with real students (mostly ***ages 11-13***) and a real class in progress.

- Discuss how the change in target audience might change how the drone identifies attention.
- Discuss how this might affect testing.

Please ***return to the main room*** to report after the breakout rooms close (after 10 minutes if they don’t close).