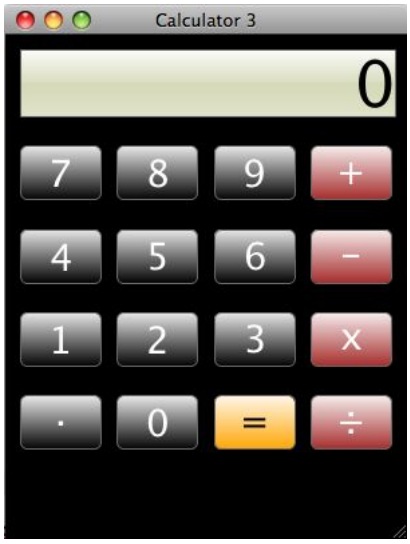
Lecture Notes
September 25, 2018

The purpose of testing is to find bugs: what is a bug?

Bugs are not always in code, could be in the requirements, design, tests or documentation

What does it mean for there to be a bug in:

- Requirements?
- Design?
- Tests?
- Documentation?



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

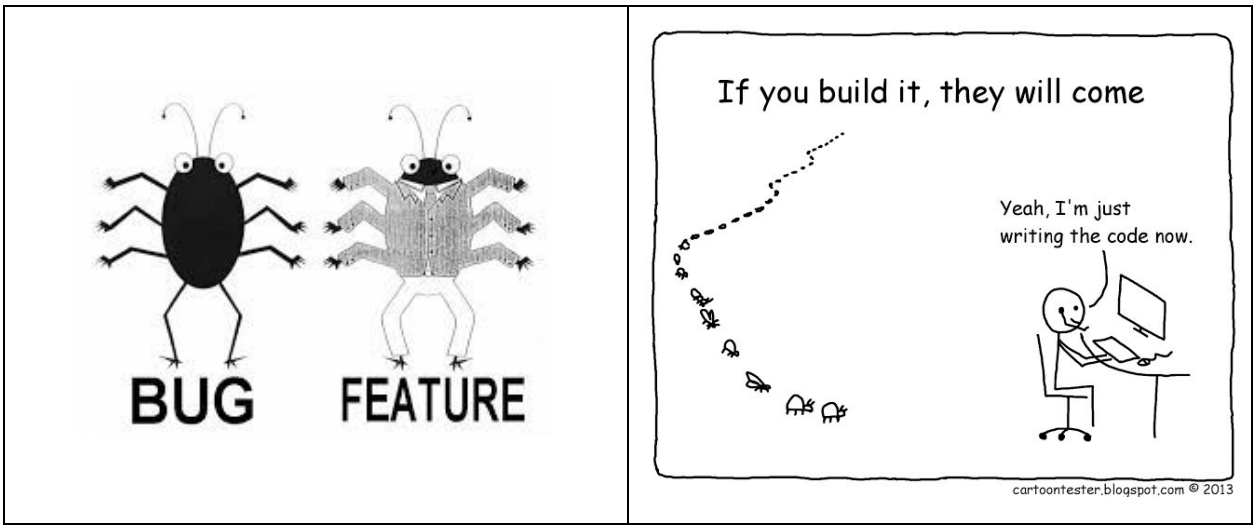
Calculator displays all 0's and won't do anything else

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 cannot be read in bright light, ...

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



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

Static analysis often 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

Static analysis can also be applied to bytecode or binaries, not necessarily to find bugs - may be used to detect malware

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

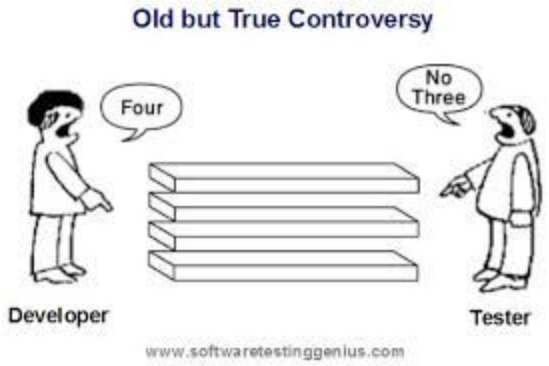
For example, fuzzing takes existing test cases and "fuzzes" the inputs with semi-random changes, seeking to produce inputs that are:

- structurally invalid
- structurally valid but semantically invalid

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

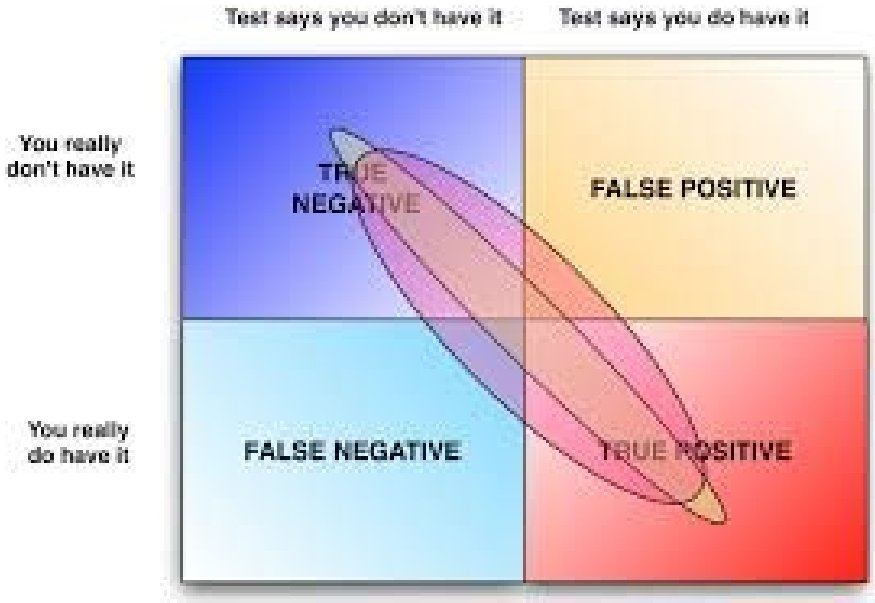
It is sometimes argued that the second case consists of false positives, because the invalid input would never have reached the internal code, beyond the gatekeeper, if it were not for the gatekeeper bug (*missing* code can be a bug!)

To find application-specific bugs in your program, you need to check the features, also a form of dynamic analysis but 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

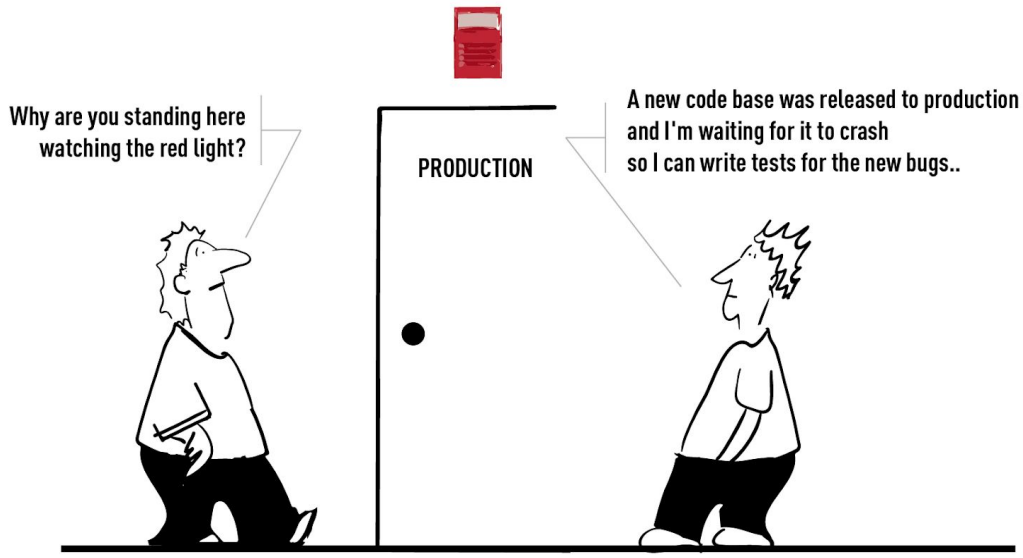
Both static and dynamic analysis exhibit *false negatives*, failing to report all bugs - there is no technique that can be guaranteed to find *all* bugs in a non-trivial program written in a conventional programming language



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.



Functional testing: Checks that the code does what it is supposed to do - often divided into unit, integration, system and acceptance testing - but does not necessarily check whether the code is fast, user friendly, or works on a wide range of platforms

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).

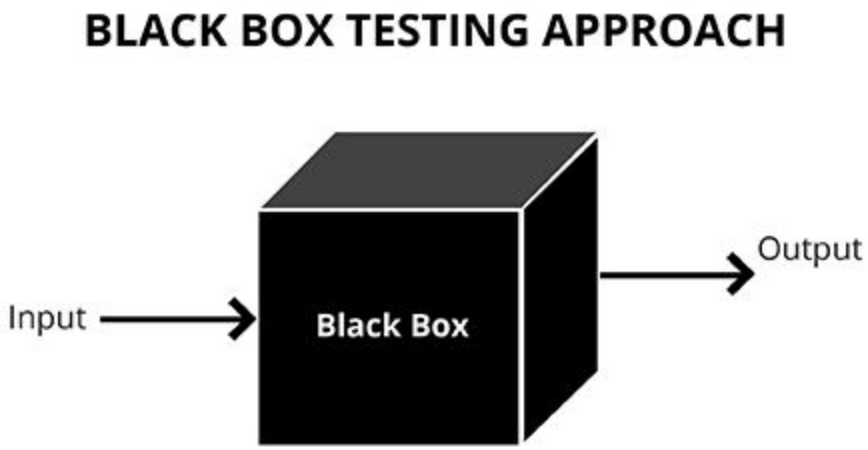
Integration testing - tests boundaries between "units", to check that the units do what they are supposed to do to fulfill their role in the design. Usually integrate only two, or a few, units at a time. So now when a unit would normally call other code, it indeed calls code in the selected other unit(s),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

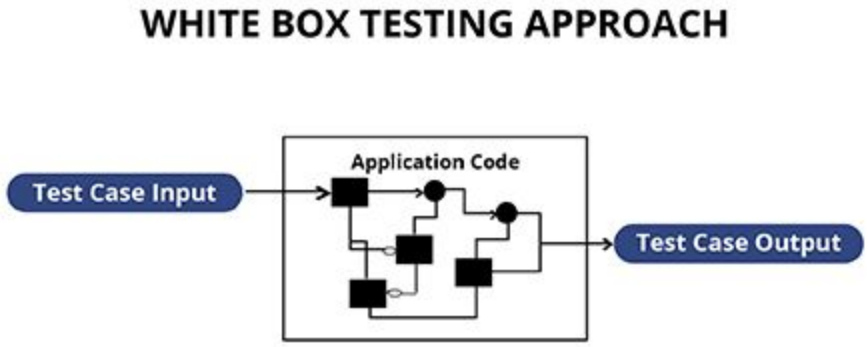
Integration testing may also refer to testing the full program's interfaces to external libraries, systems, databases, etc.

System testing - aka end-to-end testing, tests 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 - system testing in customer's environment with real-world users and (sometimes) real-world workloads, beta testing

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





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.

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 same developers who coded the box.

At integration and system testing levels, white box consideration 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

Let's say we intend to test this function:

```
public String concat(boolean append, String a,String b) {
        String result = null;
        if (append) {
           result = a + b;
                        }
        return result.toLowerCase();
    }
```

We might have this test case:

```
 @Test public void testStringUtil() {
        String result = stringUtil.concat(true, "Hello ", "World");
        System.out.println("Result is "+result);
    }
```

It exercises every line of code in the function. Is this enough testing? Why or why not?

Let's say we add this test case, which is exactly the same except the boolean append argument is false rather than true.

```
@Test public void testStringUtil() {
        String result = stringUtil.concat(false, "Hello ", "World");
        System.out.println("Result is "+result);
    }
```

What happens when we run this test case?

Notice we had two separate test cases for true vs. false. Since testing the same method with a different input, it would be better to have *one* test case and some way to parameterize it with alternative inputs.

We would like to test this code with multiple different arguments n, at least one odd and one even. How can we do that without writing two (or more) separate test cases?

```
public class MathChecker {
      public Boolean isOdd(int n) {
         if (n%2 != 0) {
            return true;
         } else {
            return false;
                 }
         }
   }
```

Parameterized test cases:

```
@RunWith(Parameterized.class)
 public class MathCheckerTest {
      private int inputNumber;
      private Boolean expected;
      private MathChecker mathChecker;

      @Before
      public void setup(){
      mathChecker = new MathChecker();
      }

    // Inject via constructor
    public MathCheckerTest(int inputNumber, Boolean expected) {
       this.inputNumber = inputNumber;
       this.expected = expected;
    }

    // Could read test data from a file rather than hardcoding
    @Parameterized.Parameters
    public static Collection<Object[]> getTestData() {
       return Arrays.asList(new Object[][]{
             {3, true},
             {6, false},
       });
    }

    @Test
    public void testisOdd() {
       System.out.println("Running test for:"+inputNumber);
       if (mathChecker.isOdd(inputNumber) = expected) {
             System.out.println("Success");
       } else {
             System.out.println("Failure");
       }
    }
 }
```

Ideally, testing is *automated* to remove human tedium and error, reproducible, faster, more likely to be done.

Consider this variant of concat:

```
public class StringUtil {
    public String concat(String a,String b) {
        return a + b;
    }
}
```

Here is one approach to testing concat:

```
@Test
    public void testStringUtil_Bad() {
     String result = stringUtil.concat("Hello ", "World");
     System.out.println("Result is "+result);
    }
```

Why would it be difficult to automate this test?

What should we do instead?

An automated testing system would always treat the first test as "pass", since there are no assertions, and a human has to check. (Or string manipulations on output redirected to a file, but that is outside the automated testing system.)

Consider concat again:

```
public class StringUtil {
    public String concat(String a,String b) {
        return a + b;
    }
}
```

Here is another approach to testing concat:

```
@Test
    public void testStringUtil_Good() {
     String result = stringUtil.concat("Hello ", "World");
     assertEquals("foobar", result);
    }
```

This test will "fail" if the method returns the wrong result, without human intervention.

More on automated testing tools on Thursday...

In-class exercise if time permits: Sit with your team, discuss ideas for preliminary project proposal. This is also a good time to ask me if you have any questions or concerns about your project ideas.

Read for Thursday: (these are each only one page long)

- Unit Testing
- Refactoring
- Continuous Integration
- Test Driven Development
- Burndown Charts
- Cone of Uncertainty
- Management by Miracle
- Three Simple Truths

Team assignment Preliminary Project Proposal due Thursday, 11:59pm.

Pair assignment Practice with Bug Finders due next Tuesday, October 2, 10:10am.