COMS W4156 Advanced Software Engineering (ASE)

September 23, 2021

shared google doc for discussion during class

Quick Overview of Finding Bugs

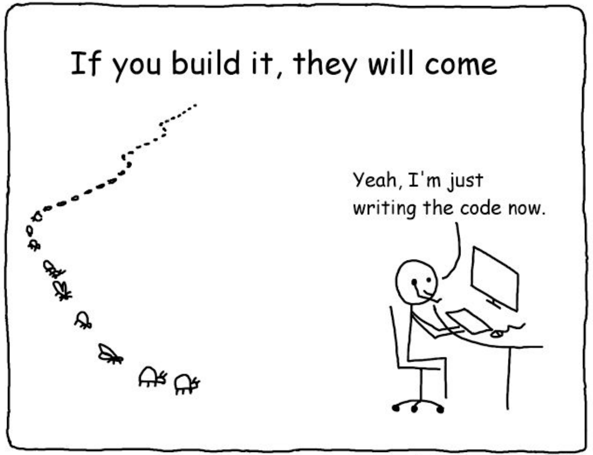
There is no shortage of bugs, every non-trivial program has bugs

Ideally find and fix them before your users do

Static Analysis

Testing

~~Formal Methods~~


If you build it, they will come
Yeah, I'm just writing the code now.

cartoontester.blogspot.com © 2013

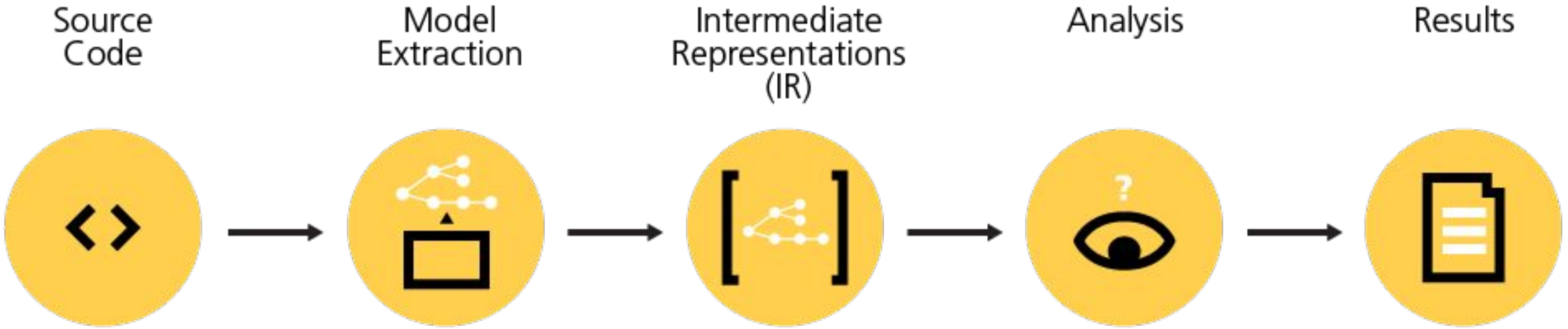
Static Analysis

Any analysis of code that does not actually execute the code is “static”

Includes style checkers: lack of compliance to coding standards does not mean there’s a bug, but does make it more likely for bugs to occur as multiple developers (mis-)understand and modify the code

But the term static analysis or static analyzer more commonly refers to code smell detectors and bug finders (often integrated in same tool)

© CanStockPhoto.com



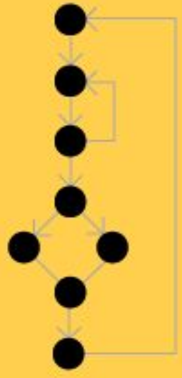
Source Code → Model Extraction → Intermediate Representations (IR) → Analysis → Results

Names Databases/Symbol Table

Name	Kind	Location
copy_item	function	item. c:25
item_cache	variable	itemc:10
color	parameter	pallette.c:23
header.h	file	chapes.c

Abstract Syntax Tree (AST)

Control Flow graph (CFG)



Call Graph

Static Analysis Bug Finders

Generic bug finder tools do not need to know anything about the intended functionality (features)

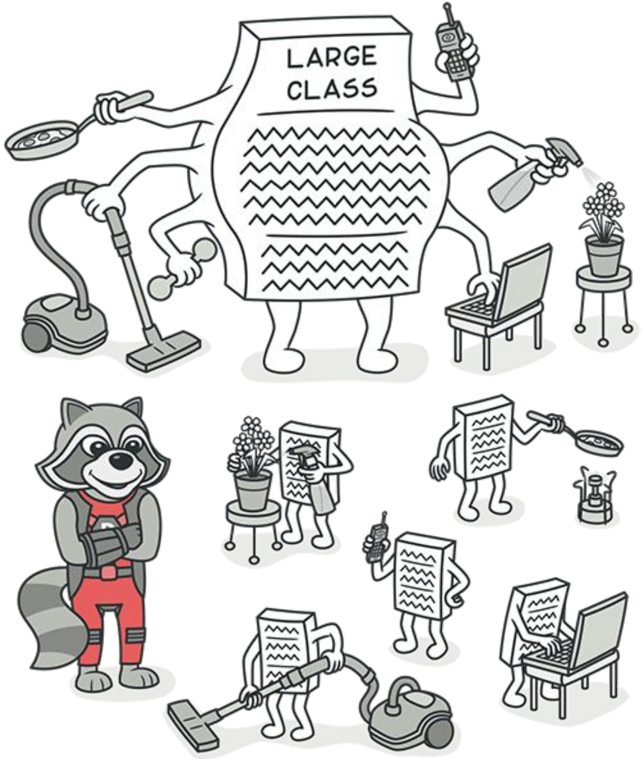
Instead look for “patterns” in the code:

- Code Smells do not necessarily imply bugs, but tend to lead to bugs as multiple developers (or your future self) modifies the code
- Resource leaks where resources (e.g., memory, database connections) are not freed on every code path reachable from an allocation
- Code does not check/handle every possible error code or exception that can be returned by an API
- Code uses “blacklisted” library, service, framework with known security vulnerabilities
- Others...

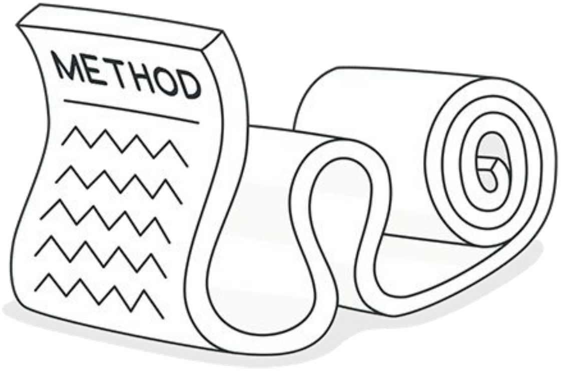
Code Smells

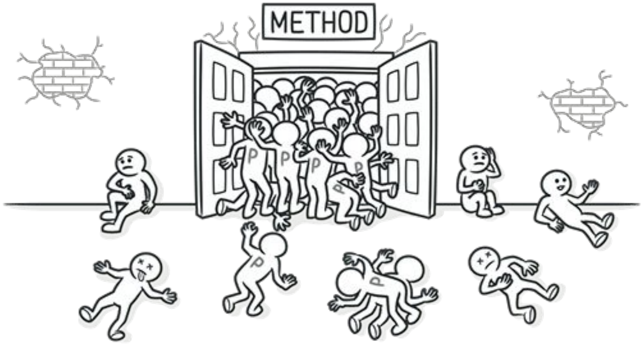
Some smells are obvious and could be addressed by style checkers, e.g., “bloater” classes, methods, parameter lists that are too big

Often means the class or method is trying to do too many things

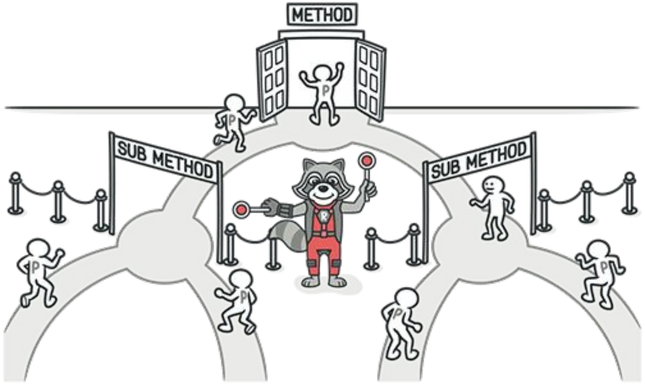


Bloaters


METHOD


METHOD
P

SubMethod
SubMethod
SubMethod
SubMeth
R


METHOD
SUB METHOD
SUB METHOD
P
R

Code Smell Detectors

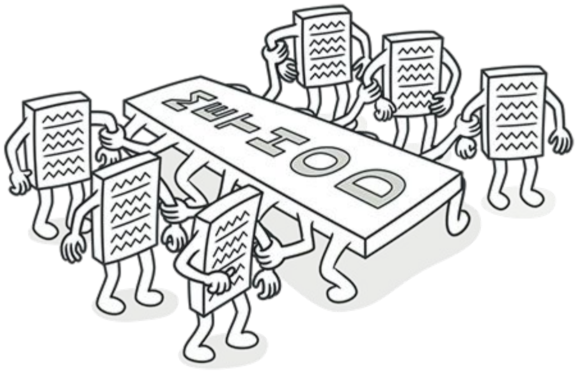
Some smells are less obvious, usually not addressed by coding style rules

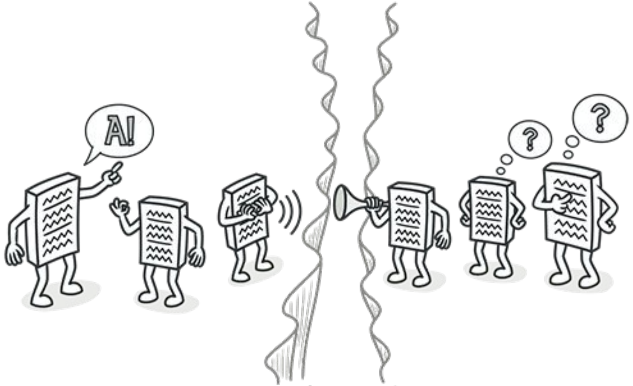
Feature envy - a “coupler” class that uses public methods and fields of another class excessively, perhaps more than it uses its own

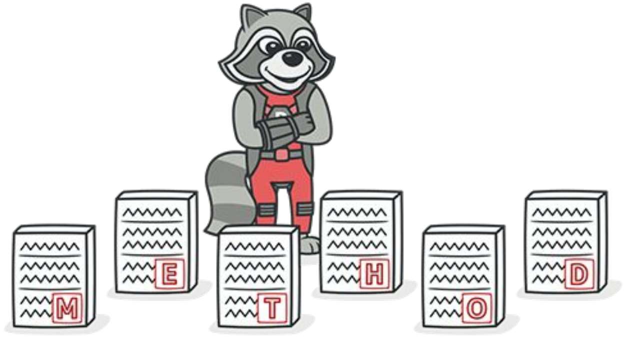
Inappropriate intimacy - a “coupler” class that has dependencies on what should be *internal* fields and methods (implementation details) of another class

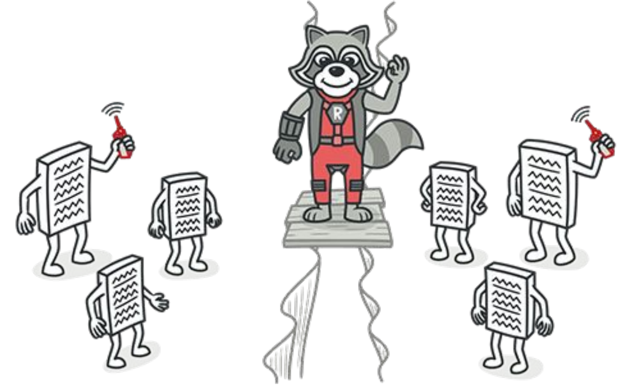
Coupling is the opposite of information hiding (or encapsulation), means changes in one code unit can force a ripple effect of changes to other code units

Coupling







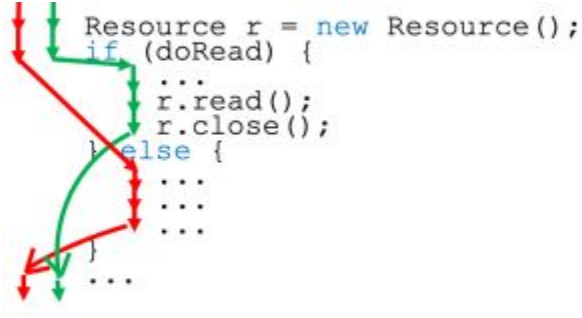


```
public class Phone {
  private final String unformattedNumber;
  public Phone(String unformattedNumber) {
     this.unformattedNumber = unformattedNumber;
   }
  public String getAreaCode() {
     return unformattedNumber.substring(0,3);
   }
  public String getPrefix() {
     return unformattedNumber.substring(3,6);
   }
  public String getNumber() {
     return unformattedNumber.substring(6,10);
   }
}
```

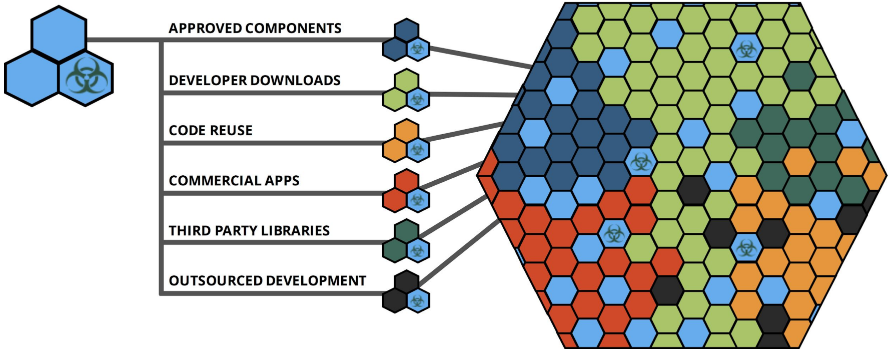
```
public class Customer…
  private Phone mobilePhone;
  public String getMobilePhoneNumber() {
    return "(" +
      mobilePhone.getAreaCode() + ") " +
      mobilePhone.getPrefix() + "-" +
      mobilePhone.getNumber();
  }
```

```java
public class Phone {
  private final String unformattedNumber;
  public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
  private String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
  private String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
  private String getNumber() {
      return unformattedNumber.substring(6,10);
   }
  public String toFormattedString() {
      return "(" + getAreaCode() + ") " + getPrefix() + "-" +
getNumber();
   }
}
```

```java
public class Customer…
  private Phone mobilePhone;
  public String getMobilePhoneNumber() {
      return mobilePhone.toFormattedString();
   }
```



```
Resource r = new Resource();
if (doRead) {
    ...
    r.read();
    r.close();
} else {
    ...
    ...
    ...
}
...
```



APPROVED COMPONENTS

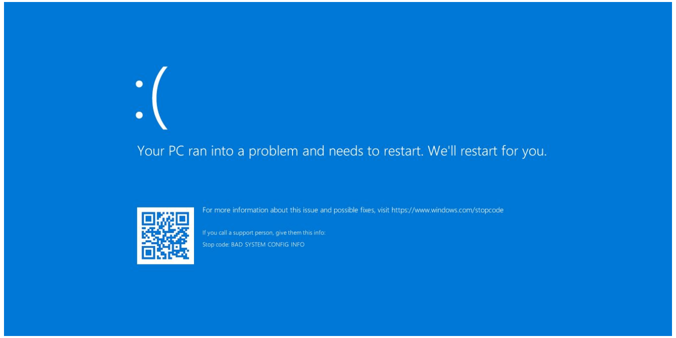
DEVELOPER DOWNLOADS

CODE REUSE

COMMERCIAL APPS

THIRD PARTY LIBRARIES

OUTSOURCED DEVELOPMENT



:(

Your PC ran into a problem and needs to restart. We'll restart for you.

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: BAD SYSTEM CONFIG INFO

Unsanitized User Input

```
if (loginSuccessful) {
  logger.severe("User login succeeded for: " + username);
} else {
  logger.severe("User login failed for: " + username);
}
```

Say user has entered username “guest”, the log gets

```
May 15, 2020 2:19:10 PM
java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: guest
```

Say the user entered username
“guest
May 15, 2020 2:25:52 PM
java.util.logging.LogManager$RootLogger log
SEVERE: User login succeeded for:
administrator“

The log gets

```
May 15, 2020 2:19:10 PM
java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: guest
May 15, 2020 2:25:52 PM
java.util.logging.LogManager log
SEVERE: User login succeeded for:
administrator
```

Dereferencing a Null Pointer

```
int a, b, c; // some integers
int *pi;     // a pointer to an integer

a = 5;
pi = &a; // pi points to a
b = *pi; // b is now 5
pi = NULL;
c = *pi; // this is a NULL pointer dereference
```

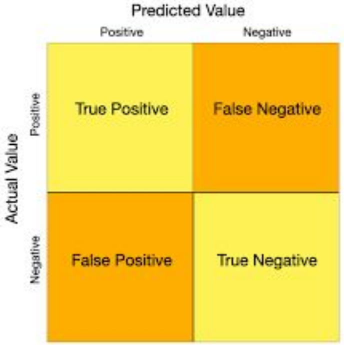
```
int a, b, c; // some integers
int *pi;     // a pointer to an integer

a = 5;
pi = &a; // pi points to a
b = *pi; // b is now 5
pi = abracadabra(a, b);
c = *pi; // could pi be a NULL pointer?
```

You're dereferencing a null pointer!

Static Analysis vs. Testing

Static analysis finds bugs that *might* happen with some input

Testing finds bugs that *did* happen with a specific input, but it’s infeasible to consider all possible inputs so misses many bugs (false negatives)

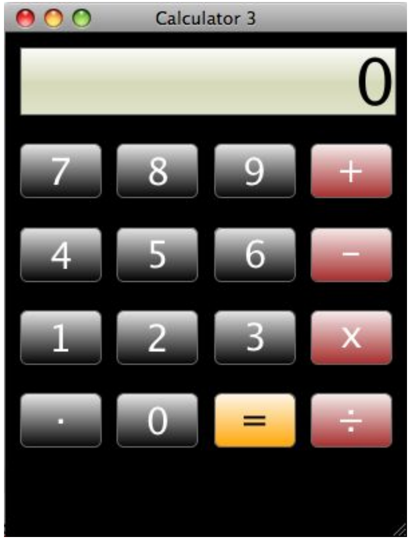


		Predicted Value	
		Positive	Negative
Actual Value	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

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)
}
```

What Kinds of Bugs Can Be Found By Testing



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

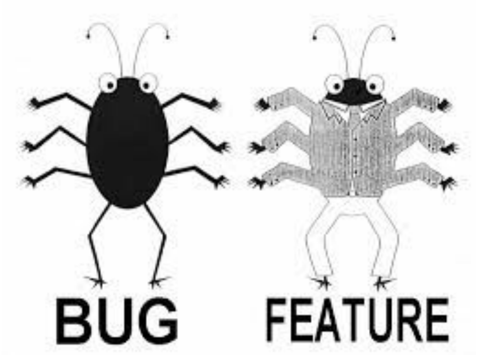
Calculator displays all 0’s and won’t do anything else

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, ...

What Kinds of Bugs Can Be Found By Testing

- 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.



Is This a Bug or a Feature?

The tester enters a number, presses +, enters another number, presses = ● Nothing happens ● Computes the wrong answer	Probably a bug
Calculator works correctly for some period of time or some number of computations, and then does nothing	Feature - trial period over, now you have to pay for it
Calculator displays all 0’s and won’t do anything else	Feature - battery low
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	Undocumented feature
The calculator’s buttons are too small, the = key is in an odd place, the display is hard to read, ...	The user will consider these bugs even though the developers may not

Getting Started with Testing

Test-to-Pass vs. Test-to-Fail

Initial testing makes sure the software minimally works with typical inputs (“smoke test”)

But most testing is trying to find bugs, which means testing with valid inputs designed to trigger “corner cases” and with invalid inputs

Test Granularity

Unit - there should be at least three tests, per parameter, for every non-trivial method

System - there should be at least three tests for every entry point

Why three?

How to Choose Test Inputs

Typical valid inputs

Enter age: ______25______________

Atypical valid inputs

Enter age: ______114_____________

Invalid inputs

Enter age: ______3.14159________

Equivalence classes, boundary conditions and fuzzing covered later in semester

Where do Invalid Inputs come from?

Users

Network

Devices

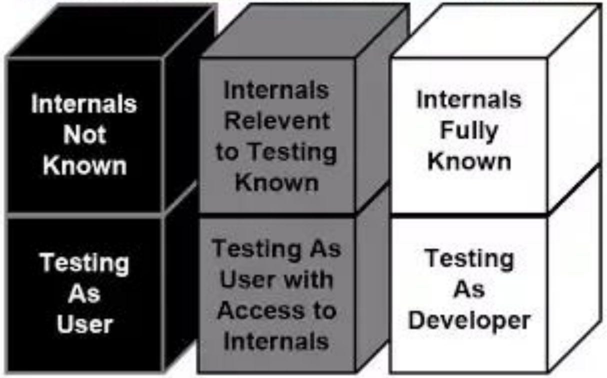
Databases

Files

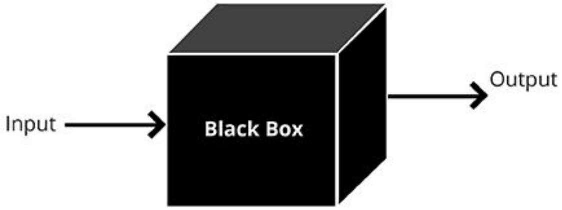
...

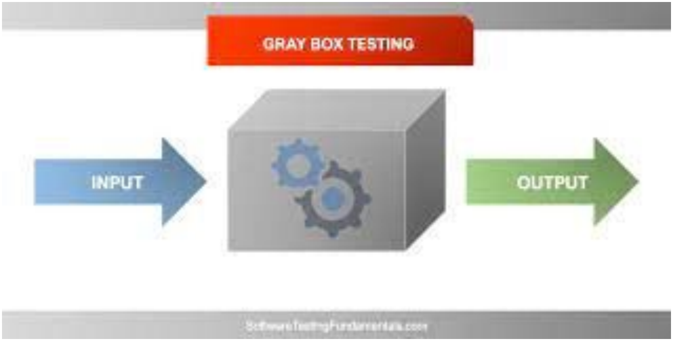
Approaches to Testing

Differences Between Box Testing Types

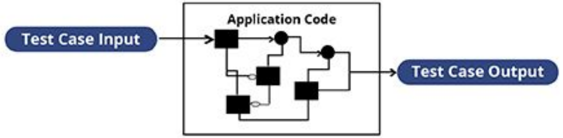

Internals Not Known
Internals Relevent to Testing Known
Internals Fully Known
Testing As User
Testing As User with Access to Internals
Testing As Developer

BLACK BOX TESTING APPROACH





WHITE BOX TESTING APPROACH



Blackbox matches Testing Intuition, so...

Why greybox?

Some bugs are not immediately visible externally, e.g., in-memory side-effects, resource leaks, corrupted data

Check logs, databases, file system, network traffic

Why whitebox?

If you never ran this part of the code, how can you have any confidence that it works?

Helps with choosing inputs intended to reach specific parts of the code

Coverage

At minimum, unit testing and system testing collectively exercise every statement - much easier to force with unit than system testing

Better: Exercise every branch

Stricter coverage discussed later in semester

Coverage may check missing conditional cases but can never detect entirely missing code

Beware the missing else!

```
if (condition) { do something }
else { do something else }
// some other code is here
// some other code should be here but isn’t
```

```
if (condition) { do something }
// some other code is here
// some other code should be here but isn’t
```

Testing Automation Concepts

It may be possible to run each test manually, and track coverage manually, but most industry testing uses some push-button tool

Continuous Integration = Rebuild the system and run the whole test suite every time the codebase changes (or periodically, e.g., every night)

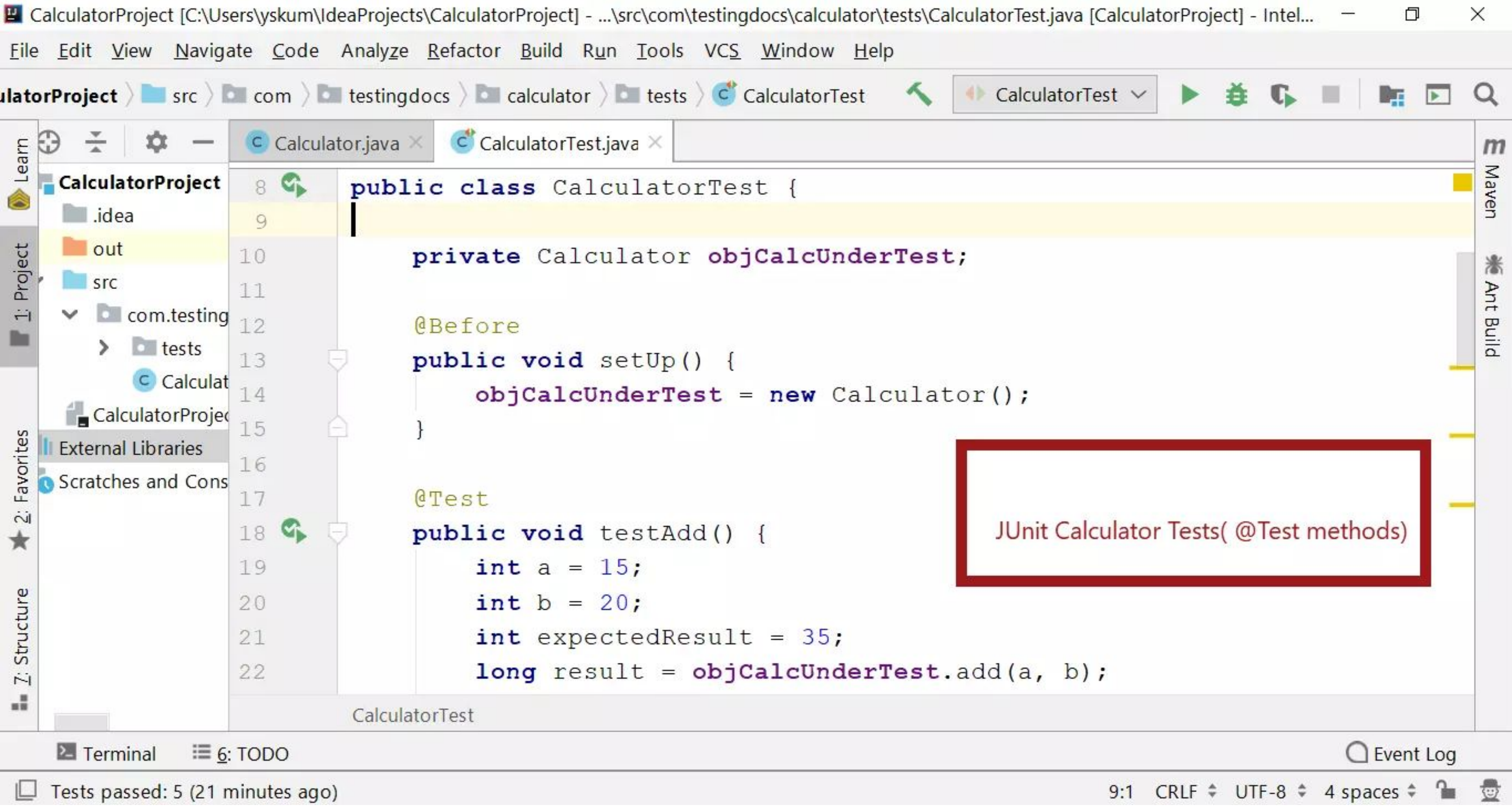
Test fixture = setUp() before tests run, tearDown() after tests run

Test case = runs code with specific set of inputs and checks results

Test suite = collection of test cases

Test runner = tool that runs your tests, either specific tests or “discovers” all the test cases for some unit

Assert methods = special functions and operators used in checking results, e.g., assertEqual(), assertRaises(), assertTrue()

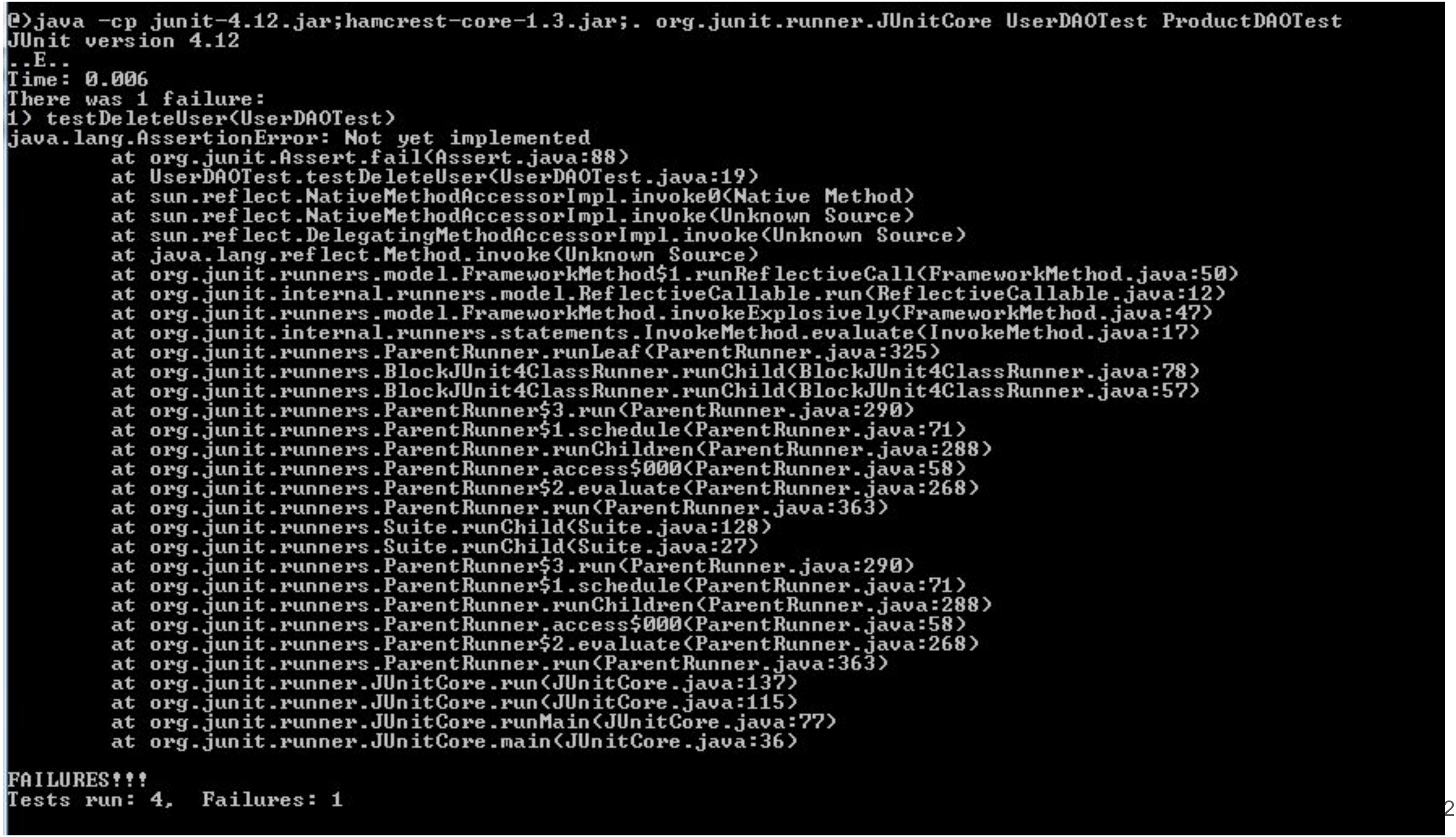


CalculatorProject [C:\Users\yskum\IdeaProjects\CalculatorProject] - ...\src\com\testingdocs\calculator\tests\CalculatorTest.java [CalculatorProject] - Intel...

```java
public class CalculatorTest {

    private Calculator objCalcUnderTest;

    @Before
    public void setUp() {
        objCalcUnderTest = new Calculator();
    }

    @Test
    public void testAdd() {
        int a = 15;
        int b = 20;
        int expectedResult = 35;
        long result = objCalcUnderTest.add(a, b);
```

JUnit Calculator Tests(@Test methods)

Tests passed: 5 (21 minutes ago)



```
@>java -cp junit-4.12.jar;hamcrest-core-1.3.jar;. org.junit.runner.JUnitCore UserDAOTest ProductDAOTest
JUnit version 4.12
..E..
Time: 0.006
There was 1 failure:
1) testDeleteUser(UserDAOTest)
java.lang.AssertionError: Not yet implemented
        at org.junit.Assert.fail(Assert.java:88)
        at UserDAOTest.testDeleteUser(UserDAOTest.java:19)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
        at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)
        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
        at org.junit.runners.Suite.runChild(Suite.java:128)
        at org.junit.runners.Suite.runChild(Suite.java:27)
        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:115)
        at org.junit.runner.JUnitCore.runMain(JUnitCore.java:77)
        at org.junit.runner.JUnitCore.main(JUnitCore.java:36)

FAILURES!!!
Tests run: 4,  Failures: 1
```

Individual Mini-Project

Three parts:

1. Implementing a simple game (intentionally due tomorrow, the day after program change period ends)
2. Testing the game (due September 29)
3. Saving game state (due October 6)

Note we added the requirement to submit a <= 2-minute demo video for each part (this will be used only for grading)

See Connect Four

Team Project

Team Formation

After teams have been formed, you will propose your own project (within constraints, most notably you will implement a service not an app) and proceed to develop/test/demo the project in two iterations

Preliminary Project Proposal