Lecture Notes
October 2, 2018

Pair assignment Practice with Bug Finders due before class today.

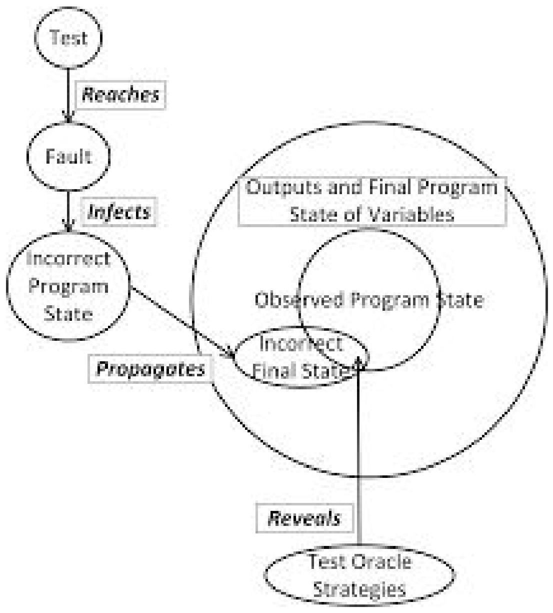
The next pair assignment Bug Hunt, due Tuesday, October 16, 10:10am, is similar - but looking for bugs in someone else's existing codebase.

Bug finder tools (both static and dynamic) usually find *generic* bugs, that can occur in lots of different kinds of programs.

But we also need to find application-specific (and design-specific) bugs. The easiest way to do this is with software testing.

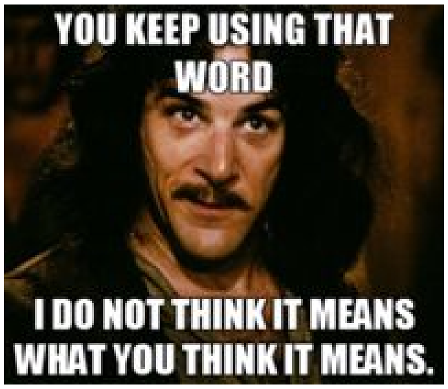
Recall the purpose of test cases is to detect the *presence* of bugs - it is impossible to use testing to verify the *absence* of bugs

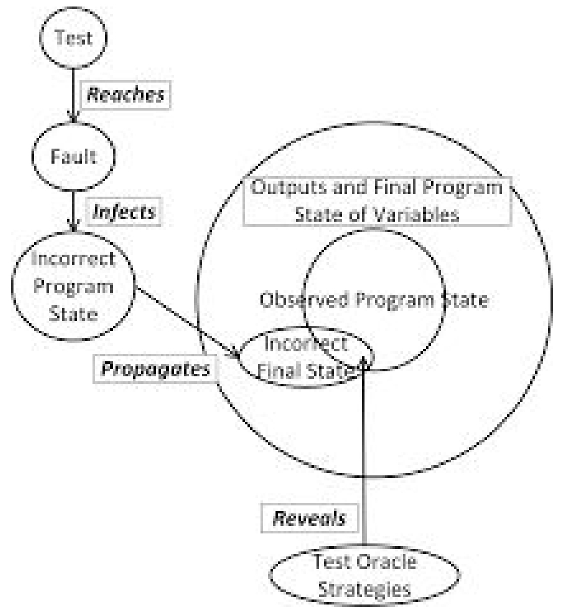
Say there is at least one bug lurking somewhere in your codebase, how does testing find it? Keep in mind we do not know where it is, and there are probably many - not just one.



First, at least one test must *reach* the location (or locations) in the code containing the bug. Or reach the location(s) where the code ought to be, in cases where the bug is due to missing code.

This may require intermediate "inputs" that are not parameters to the method under test, e.g., return values from stubs representing library calls, to force certain paths that have not already been exercised





After that part of the code has been executed (or the surrounding code when code is missing), the state of the program must be incorrect with respect to achieving its responsibilities to the program design. The program state has been *infected*.

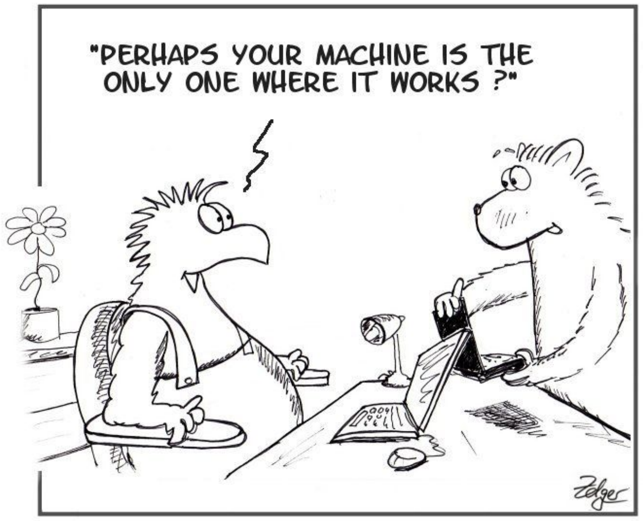
This infected program state must *propagate* through later code executed by the same test case, to cause some externally visible state or output to be incorrect.

Finally, the test case assertions (or human-checked logs, print statements, etc.) must *reveal* the incorrect aspects of the program state or output. Even if the buggy state is “visible”, it can’t actually be seen unless you look for it.

For professional software organizations, there is generally a substantial tool chain for software testing.

Continuous integration (CI) tools like [Jenkins] and [Travis CI] hook a version control system with build and testing tools that run after every commit to the shared repository, or nightly for very large codebases and test suites, so errors can be detected quickly

- **Don't break the build!**



It works on my machine

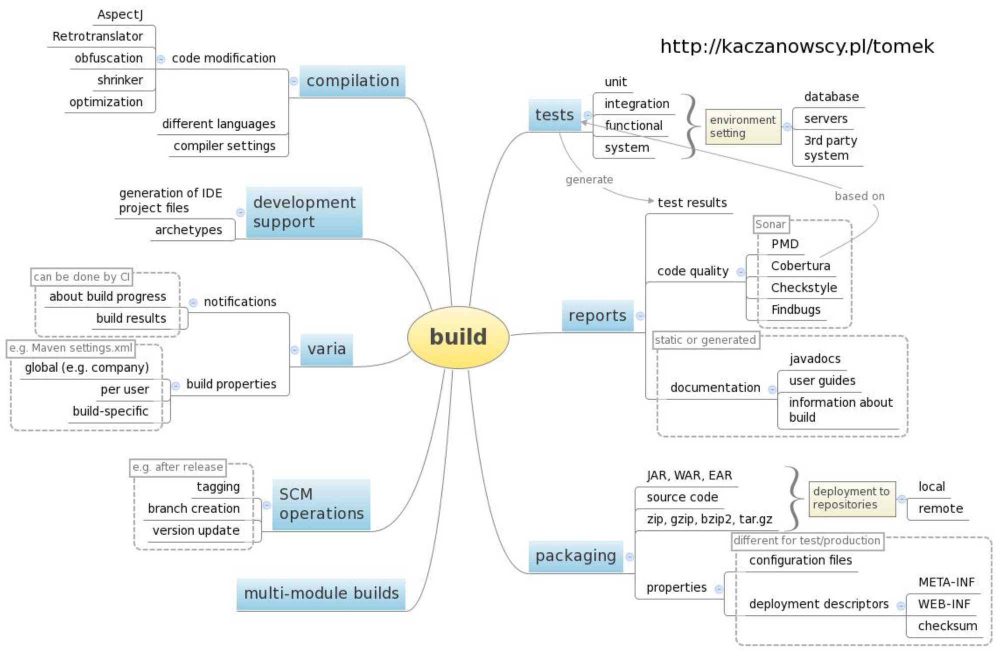
What does it mean to “break the build”?

Most if not all software organizations have some form of automated “build”, which is how they put together their entire application for end-to-end system testing and for deployment to users

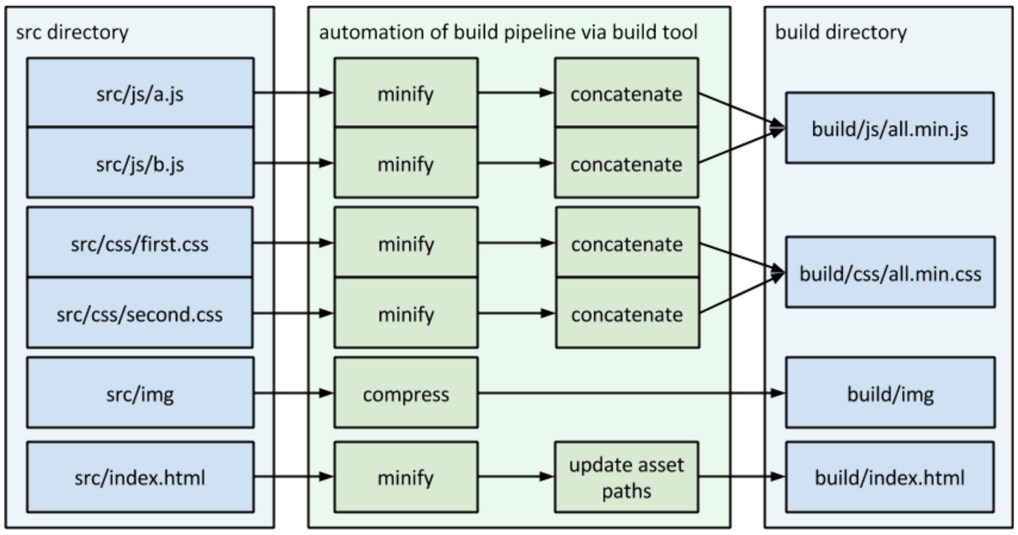
The build used for testing may not be identical to what will be deployed, e.g., the testing build often includes additional logging/debugging facilities whereas the deployment build is “optimized” for end-users (but that final build should also be tested!)

The earliest widely-known build tool is make, developed for compile and link of C and other 1970s Unix languages - updated versions are in common use today

Today’s build tools, including make, automate much more than compile/link and apply even to non-compiled languages - e.g., run package manager to update all dependencies and/or package up your code to be installable by others from a package manager



A *build configuration* is a repeatable, reproducible, standard sequence of steps so developers do not have to remember the steps and everyone uses exactly the same steps



Build tools provide some notation for writing down these steps

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Given this “makefile”, what does “make” do?

What does “make clean” do? Why?

A more typical makefile looks like:

```
src = $(wildcard *.c)
obj = $(src:.c=.o)
LDFLAGS = -lGL -lglut -lpng -lz -lm
myprog: $(obj)
    $(CC) -o $@ $^ $(LDFLAGS)
clean:
    rm -f $(obj) myprog
```

Java Build Tools: Ant vs Maven vs Gradle

Gradle supports multiple languages, not just Java; most languages have at least one language-specific build tool

Non-trivial software has *dependencies* on other software (even trivial software usually has dependencies on standard libraries)

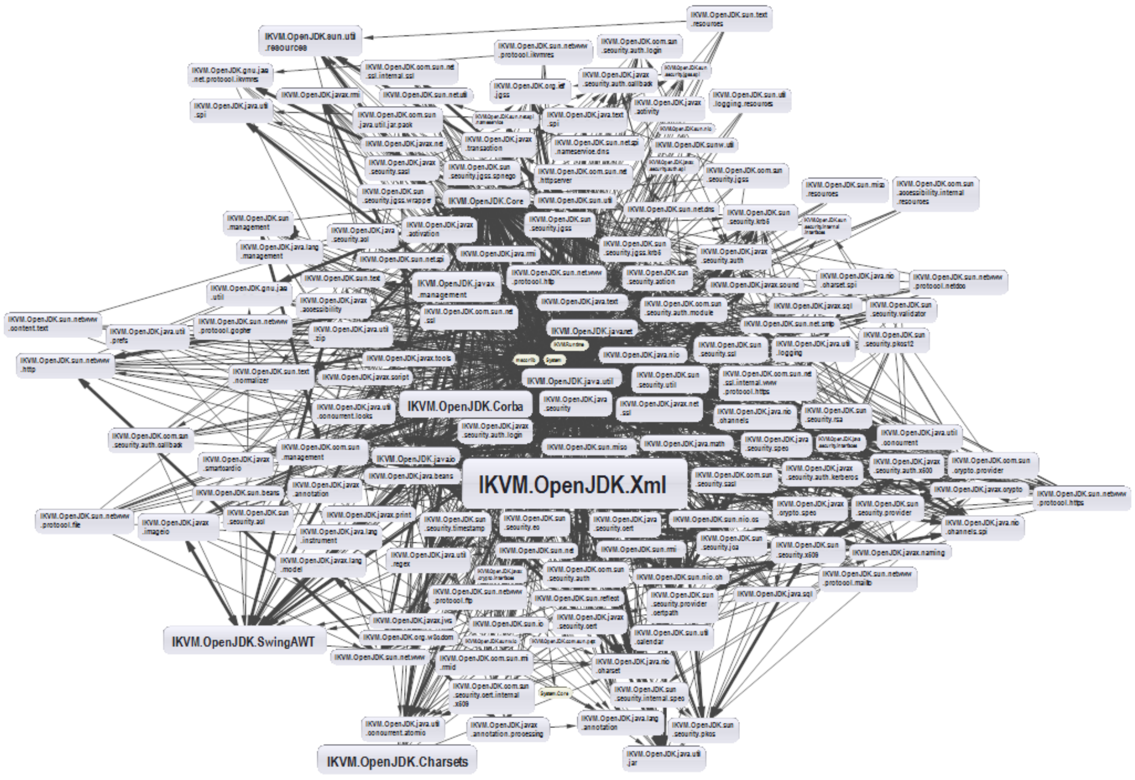
When you organize your own codebase into multiple components - classes, modules, packages, libraries, etc. - then some of your components are likely to be *dependent* on your other components

When you reuse components or other code and/or data resources written by others, your code *depends* on those resources

For example, if you write an application that requires authentication and authorization (e.g., login userid and password, maybe 2-factor), instead of implementing this capability yourself, you should use a widely-used package written by some expert(s). Then your program depends on that package.

Dependencies may be *transitive*, A depends on B depends on C depends on D.

Highly interconnected dependencies and/or reliance on specific versions of the external components ("DLL hell" on Windows) is **bad**. Why?



In order for your software to run, the compiler or runtime environment needs to know what all the dependencies are and where to find them

Package managers (e.g., maven central, pip, npm) keep track of the third-party components that your codebase uses, beyond those that “come with” the language platform

They utilize the “metadata” associated with each package - the software's name, description of its purpose, version number, vendor, checksum, and a list of dependencies

Many of them work like a mobile app store:

- Download third-party libraries and keep up to date
- Based on ecosystem with central repository
- Ensures placed in right spot on your file system
- Can also remove and clean up
- Eliminates the need for manual installs and updates

Hopefully also ensure integrity and authenticity by verifying digital certificates and checksums

The term *dependency* is also used in software engineering to refer to the *coupling* code smells - the degree to which your component depends on the "innards" of another component, as opposed to its interface, so that if the other code changes internally you also need to change your own code.

Dependency may also mean a relationship among work tasks, such that you cannot do one task until another task has been completed. Ideally, dependencies among tasks by different software developers should be minimized, so they can work in parallel.

There are also control flow and data flow *dependencies* between program elements (e.g., statements, expressions, variables) within a single code unit

Continuous integration (and "breaking the build") normally includes automated testing, not just compilation and packaging

There may be *hidden* dependencies within the application code and/or within the testing code that affect CI

A "flaky test" is a test case that produces different results on different test suite executions *even when (apparently) nothing changed.* What could cause this?

PASS
FAIL
toptal

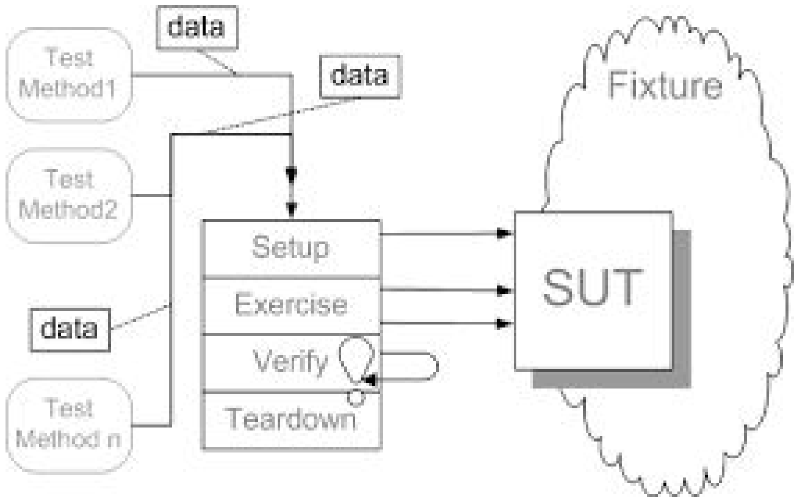
```
Assert.AreEqual(
    GetTimeOfDay(),
    "Morning" );
```

The most common causes of flaky tests are dependencies on test order (due to dependencies among application code and/or testing code) and dependencies on external resources

Test classes should be *independent*, so can run in any order, or in parallel, to get same result

A test class should not have dependencies on any other test class running beforehand, e.g., the N-1st test leaving the application state in the form expected for the Nth test. Any test cases intended to share application state and run in a specific order should be included in the same test class.

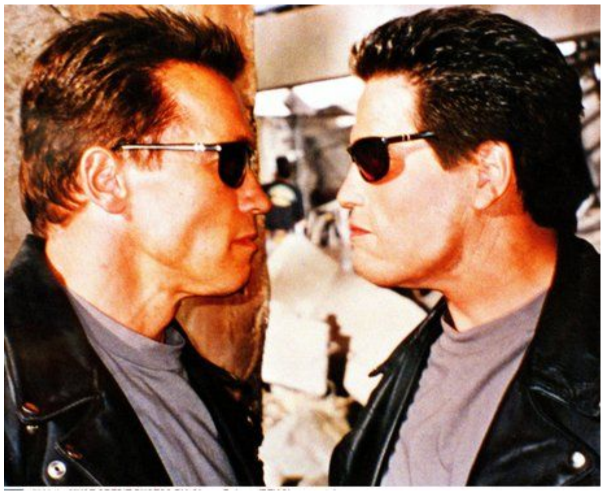
This is the motivation for associating setup/teardown code with each test case (or test class).



But setup and teardown are often incomplete (and may have their own bugs). Finding all hidden dependencies among test cases is extremely expensive - for n test cases, in worst case O(2^n). Why?

Rather than finding all hidden dependencies, some testing frameworks prevent any hidden dependencies from affecting test results by restarting/rebooting the environment in between test classes just in case - also extremely time-consuming, might take 10ms to run each test and 2s to restart JVM, much longer to reboot Linux and most other OSes, but only O(n)

One way to avoid flaky tests due to dependencies within the codebase and on external resources is to use "test doubles" (as in stunt doubles)



Multiple kinds of test doubles:

- Dummies
- Stubs
- Spies
- Fakes
- Mocks

Dummy objects are passed around but never actually used except to fill parameter lists

```
private class FooDummy
implements Foo
{
    public String bar() {
        return null;
    }
}
```

```
public class FooCollectionTest
{
    @Test
    public void it_should_maintain_a_count()
    {
        FooCollection sut = new FooCollection();
        sut.add(new FooDummy);
        sut.add(new FooDummy);
        assertEquals(2, sut.count());
    }
}
```

Stubs are fake objects or individual methods that implement a required interface with pre-programmed behavior - usually simply returning canned values.

- Avoid making actual requests to a server
- Feed the system with known data, forcing a specific code path

```
private class FooStub
implements Foo
{
    public String bar()
    {
        return "baz";
    }
}
```

```
public class FooCollectionTest
{
    @Test
    public void it_should_return_joined_bars()
    {
        FooCollection sut = new FooCollection();
        sut.add(new FooStub);
        sut.add(new FooStub);
        assertEquals("bazbaz", sut.joined());
    }
}
```

A spy is like stub, but can maintain state and keep records - e.g., number of times it was called with which arguments, usually replacing methods of a real object

Fake objects have working implementations, but usually a lighter-weight implementation that would not be suitable for production - e.g., an in-memory database

Mocks combine fakes with spies on the fake object

```
public class FooCollectionTest
{
   @Test
   public void it_should_return_joined_bars()
   {
      Foo fooMock = mock(Foo.class); // instance
      when(fooMock.bar()).thenReturn("baz", "qux"); // behavior

      FooCollection sut = new FooCollection();
      sut.add(fooMock);
      sut.add(fooMock);

      assertEquals("bazqux", sut.joined());
      verify(fooMock, times(2)).bar(); // verify
   }
}
```

Mocking can prevent some tests from being flaky - e.g., using an in-memory database avoids timeouts and other issues that can arise with a disk-based (or over-the-network) database

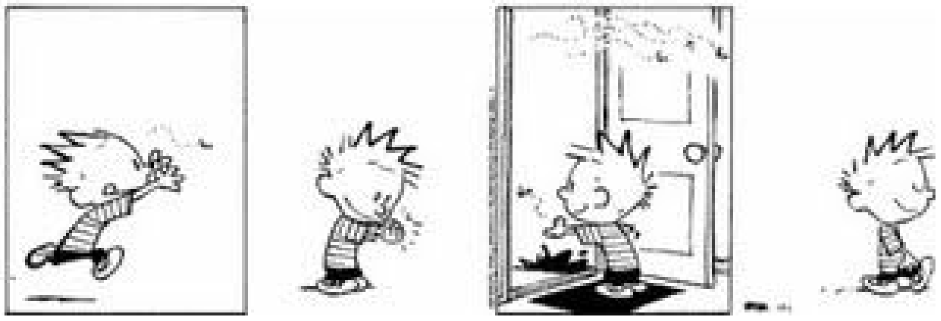
Sometimes apparently flaky tests are caused by the dreaded Heisenbug	

Sometimes the same bug is "fixed", then reappears, over and over - a special form of regression

Continuous integration often distinguishes between the smaller test suite run after every commit, usually only testing the specific code that changed, vs. the full *regression test suite* - which may take hours to run, so many organizations do a full build/test every night

A *regression* is when fixing a bug or adding a new feature causes previously working code to break, or "regress", sometimes code that seems completely unrelated in another part of the codebase (because there was a hidden dependency)

Regression:
"when you fix one bug, you
introduce several newer bugs."



To detect regression, need to re-run *all* test cases that previously passed - and even tests that previously failed, in case they now fail in different ways or unexpectedly pass

Read for Thursday:
textbook, chapter 3 Test Automation

Revised Project Proposal due next Tuesday, October 9, 11:59pm.

Next pair assignment Bug Hunt due Tuesday, October 16, 10:10am.

In-class exercise if time permits: Sit with your team, discuss ideas for revised project proposal. IA mentors should spend a few minutes with each team.