COMP 121
Software Engineering

Testing

Spring 2021

(Inspiration from Ben Liblit and Mike Ernst)
Introduction

• Software is hard to write!
  ▪ And like any human activity, we all make mistakes when building software

• Bugs in software can have major, real-world consequences
  ▪ For an ongoing list, see Paul G. Neumann, ACM Risks Forum, http://www.csl.sri.com/users/neumann/#3
  ▪ A few famous examples next…
Therac-25 Radiation Therapy Machine

- Massive radiation overdoses killed or seriously injured patients (1985-1987)
  - New design removed hardware interlocks
    - All safety checks done in software
  - Equipment control task not properly synchronized
- Error missed in testing
  - Bug only triggered if operator changed setup too quickly
  - Didn’t happen during testing because operators didn’t have enough practice yet to do this
Mars Polar Lander

- 290kg robotic spacecraft lander launched in 1999
- Lander failed to reestablish communication after descent phase
- Most likely cause: engine shut down too early
  - Legs deployed led to sensor falsely indicating craft had touched down, yet it was 40m above surface
- Error traced to a single line of code
  - Known that leg deployment could lead to a bad sensor reading, but never addressed
Ariane 5 Failure

• In 1996, Ariane 5 launch vehicle failed 39s after liftoff
  ▪ Caused destruction of over $100 million is satellites!

• Cause of failure
  ▪ To save money, inertial reference system (SRC) from Ariane 4 reused in Ariane 5
  ▪ SRI tried to compute a floating point number out of range to an integer; issued error message (as an int); that int was read by the guidance system, causing nozzle to move accordingly
  ▪ The backup system did the same thing
  ▪ Result was rocket moved toward horizontal
  ▪ Vehicle than had to be destroyed

• Ultimate cause: Ariane 5 has more pronounced angle of attack than Ariane 4
  ▪ The out of range value was actually appropriate
Software Quality Assurance (QA)

- Testing: run software, look for failures
  - Limits: risk of missing behaviors due to inadequate test suite
- Code reviews: manual review of program text
  - Limits: informal, uneven, easy to miss issues
- Software process: development/team methodology
  - Limits: one step removed from the code
- Static analysis: assess source code without running it
  - Limits: hard to scale, typically has many false positives
- Program verification: prove program correct
  - Limits: very difficult, very expensive, not scalable
- …and many more!
No Single QA Approach is Perfect

“Beware of bugs in the above code; I have only proved it correct, not tried it.” — Donald Knuth, 1977

“Program testing can be used to show the presence of bugs, but never to show their absence!” — Edsgar Dijkstra, Notes on Structured Programming, 1970

- Most popular QA approach? Testing + code review
  - Static analysis has made huge inroads recently, but is a drop in the bucket compared to testing
  - Verification is on the horizon, but is still out of reach for most systems
  - We’ll focus on testing in this class
Levels of Testing

• Unit testing: One component at a time
  ▪ A component could be a method, class, or package
  ▪ If test fails, defect localized to small region
  ▪ Done early in software lifecycle, ideally when/before component is developed, and whenever it changes

• Integration/system testing: The whole system together
  ▪ Ensures components work together correctly
  ▪ Possible even if system not complete, as long as there’s some end-to-end slice of its functionality

• Other testing terms
  ▪ “Acceptance test” — test system against user requirements
  ▪ “Regression test” — make sure new version of software behaves identically to old version
Automated Unit Testing with JUnit

- xUnit test frameworks for language x
  - Original was SUnit (Smalltalk), by Kent Beck (1989)
  - JUnit popularized the approach
- Easy to build
  - “Never in the annals of software engineering was so much owed by so many to so few lines of code.” — Martin Fowler
- Key: test cases run and checked automatically
  - This means we can run them early and often
- Testing terminology:
  - System Under Test (SUT) — doesn’t need definition!
  - Test case — code that runs part of SUT and checks result
    - Test cases can pass or fail, no gray areas
  - Test suite — a set of test cases
Installing JUnit 4

• Download junit
  ▪ Documentation here: https://junit.org/junit4/

• Download hamcrest
  ▪ https://search.maven.org/artifact/org.hamcrest/hamcrest/2.1/jar

• Add them to your CLASSPATH
  ```bash
  # bash, both files in $HOME/java
  # add the following as a single line to .bash_profile
  export CLASSPATH=$HOME/java/junit-4.13-beta-2.jar:
  $HOME/java/hamcrest-2.1.jar:.
  ```

• Test to see if junit is available
  ```bash
  $ java org.junit.runner.JUnitCore
  JUnit version 4.13-beta-2
  ...
Basic JUnit Example

# run with "java org.junit.runner.JUnitCore ListTests"
import static org.junit.Assert.*;
import org.junit.*;
import java.util.*;

public class ListTests {
    @Test public void testAdd() {
        List<Object> l = new LinkedList<>();
        Object o = new Object();
        l.add(o);
        assertTrue("list should contain o", l.contains(o));
    }
    @Test public void testIsEmpty() {
        List<Object> l = new LinkedList<>();
        assertTrue("list should be empty", l.isEmpty());
    }
}
Things to Notice

• A test case in JUnit is just a class
  ▪ Test methods are annotated with @Test
    - Java annotations begin with @, can be examined via reflection

• Each test method has one or more assertions
  ▪ From org.junit.Assert
    ▪ assertTrue, assertFalse, assertEquals, assertNull, etc

• Running tests shows passes (.) and failures (E)
  ▪ Failures come with backtrace
  ▪ Test methods run in deterministic but undefined order
    - Make sure success/failure does not depend on ordering!
  ▪ Why does it report the running time?
    - For large projects, running all tests take significant amount of time
    - Might need to be selective about which tests are run when
Tips for Assertions

• Use assertEquals etc rather than assertTrue
  ▪ Will get a more useful message if case fails
  ▪ Note: first arg to assertEquals is expected value

• Always put messages in assertions

• You can add helper methods for your own kinds of assertions, e.g.,
  ▪ <E> assertListContains(List<E> expected, E elt)
  ▪ assertApproxEqual(double expected, double actual, double delta)
    - Check expected-delta ≤ actual ≤ expected+delta
Tips for Test Cases

• Ideally, each test case should check one thing
  ▪ Makes it easier to understand what went wrong if test fails

```java
class ListTests {
    @Test void testAdd() { ... }
    @Test void testRemove() { ... } ...
    /* Rather than one large test */
}

• But you can break this rule as needed

```java
@Test void testContains {
    List l1 = ..., l2 = ...;
    assertTrue(l1.contains(1));
    assertFalse(l2.contains(1));
}
```
• Test cases fail if they throw an (uncaught) exception
  ▪ JUnit will catch the exception and keep running other tests

• If test cases catch exceptions, be specific

```java
@Test
t void testRemoveErr() {
    List l1 = ...;
    try {
        l1.remove(-1);
        fail("Removed at position -1?!");
    }
    catch (IndexOutOfBoundsException e) { }
}
```
Test Fixtures

• Creating objects per-test can be painful
  ▪ Sometimes, tests need complex web of objects
    - Expensive to reallocate for every test, leads to duplicate code

• A test fixture is an initial set of objects/state of the world for running a set of test cases
  ▪ Test fixtures are “set up” before tests are run
  ▪ They are “torn down” after tests are run
    - E.g., to close files

• JUnit supports four test fixtures annotations
  ▪ @BeforeClass, @AfterClass — methods to run once per test case class
  ▪ @Before, @After — methods to run once per test method
Test Fixtures Example

- Be careful if you mutate fixtures
  - If you do, use `@Before/@After` instead of `*Class` varieties
- Make sure `tearDown` releases all resources
  - Even in the presence of exceptions

```java
class LinkedListTest {
    List<Integer> l; BufferedReader f;

    @BeforeClass void setUp() {
        l = new LinkedList<Integer>();
        l.add(1); l.add(2); l.add(3);
        f = ... 
    }

    @AfterClass void tearDown() {
        f.close();
    }
} 
```
Test Automation

• JUnit tests are completely automated
  - Run from a single command line invocation
  - Test results checked automatically, without human intervention
    - Critically: tests must be repeatable; avoid non-determinism!

• Drawback: Adds cost
  - Have to write code and tests together
  - Have to ensure tests and code remain in sync over time

• Major benefits
  - Tests can be run often
  - Code maintenance and evolution becomes much safer
    - Rerunning tests after making a change provides a lot of confidence that the change was correct
Regression Testing

• Key idea: When you find a bug
  ▪ Write a test that exhibits the bug
  ▪ Always run the test when code changes
  ▪ ⇒ ensures bug doesn’t reappear

• Helps ensure *forward progress*
  ▪ Ideally, old bugs never reemerge
  ▪ But even if they do, you’ll find them quickly

• Note that automation is key
  ▪ Set of test cases increases over time
  ▪ Without automation, would be too hard to re-execute
Nightly Builds

• Want to run tests as often as possible
  ▪ If bug appears after small code change, easy to attribute bug to that change
  ▪ If bug appears after 1,000 code changes or very big change, tracking down the problem is harder
• But, often too expensive to run all tests on every save
  ▪ Especially as project gets large
• Split tests into two groups
  ▪ *Smoke tests* that make sure nothing is horribly wrong
    - These tests run quickly, not exhaustive
    - Run these all the time
  ▪ Full test suite less often
    - Once per night, once per week, etc
Constructing a Test Suite

• Combine tests from different classes
  ▪ To create set of smoke tests, nightly tests, etc
  ▪ (Example from JUnit documentation)

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestFeatureLogin.class,
    TestFeatureLogout.class,
    TestFeatureNavigate.class,
    TestFeatureUpdate.class
})
public class FeatureTestSuite {
    // class is empty, used only for annotations
}
```
Labeling Tests with Categories

```java
public interface TSmoke { /* category marker */ }

public class A {
    @Test public void a() { ... }
    @Category(TSmoke.class) @Test public void b() { ... }
}

@RunWith(Categories.class)
@IncludeCategory(TSmoke.class)
@SuiteClasses({A.class})
public class SmokeTestSuite {
    // Will run A.b but not A.a
}
```

- Enables flexible groups of tests
Continuous Integration

- Continuous integration (CI) = developers merge changes often
  - Typically by pushing to central version control repository
  - Helps ensure different changes do not conflict
- Creates a natural testing workflow: test before push
  - Helps maintain invariant that main branch tests succeed
- Many CI systems support this model
  - Image from Travis CI:
Record-and-Replay Testing

• What about testing GUIs?
  ■ Can unit test individual methods
  ■ But how do we test clicking buttons etc?
  ■ Standard approach: record and replay manual tests

• Key challenges
  ■ Test recording is fragile
    - Either tightly tied to UI or dependent on OS hooks for keyboard/mouse
  ■ Test replay is fragile
    - Breaks if UI changes
    - If record \((x,y)\) coordinates, breaks with different screen layouts etc
  ■ Note: manual testers would adapt to these conditions
Developing Test Cases

• Now that we know how to run tests, how do we come up with those test cases?
  ▪ This is a hard problem!

• Two main approaches:
  ▪ Derive tests from specification (black box testing)
    - Pros: This is what we actually want the program to do!
    - Cons: Specs are notoriously incomplete; specs don’t necessarily tell you every place the code could go wrong
  ▪ Derive tests from implementation (white/glass box testing)
    - Pros: This is the code we’re actually running!
    - Cons: If our code is completely missing some key property that’s in the spec, we might not even know to test it
    - Cons: Tests might be overly specific to this particular implementation

• In practice need to look at both!
Black Box Testing Approaches

- Look only at specification, not at code
Consider Each Path in Spec

- Look at the spec and consider conditional branches

```java
// Return true if x in a, else return false
boolean contains(int[] a, int x);
```

- Two “paths” through spec
  - One test where x in a, one test case where x not in a
  - Maybe another one: what if x appears twice in a?

```java
// Return maximum of a and b
int max(int a, int b)
```

- Three paths through spec
  - if a<b returns b; if a>b — returns a; if a=b — returns a

- In all cases, actual tests will need concrete values
  - E.g., test max with (3, 4) to cover first case
Consider Edge Cases

• Anticipate common off-by-one errors or forgetting something special that has to happen at the beginning or end of a range

```java
// Return true if x in a, else return false
boolean contains(int[] a, int x);
```

• What if `x` is the first element? What if it’s the last element?
• What if `a` is empty?

• In Java, consider whether `null` should be handled
  • What if `a` is null?
Consider Aliasing

What happens if \texttt{src} and \texttt{dst} are same object?
- This is \textit{aliasing} and it’s easy to forget! Watch out for this

Other useful cases (for other methods)
- \texttt{null}
- Circular lists

```java
// modifies: src, dst
// effects: removes all elts of src and appends them in reverse order to end of dst
<E> void appendList(List<E> src, List<E> dst) {
    while (src.size()>0) {
        E elt = src.remove(src.size()-1);
        dst.add(elt);
    }
}
```
Black Box Testing Advantages

• Process not influenced by tested component
  ▪ Code’s assumptions not propagated to test suite
  ▪ Tests are all about using redundancy to find mistakes
  ▪ To create useful redundancy, avoid strict duplication

• Robust with respect to implementation changes
  ▪ Shouldn’t need to change black box tests when code changed

• Allows testers to be independent
  ▪ Testers need not be familiar with code
  ▪ Tests can be developed before writing code
Glass/White Box Testing

• Look at implementation
• Focus on features not described by spec
  ▪ Concrete decisions not made in the abstract spec
    - E.g., data structure implementation decisions, max sizes of arrays, separate handling for cases that are combined in the spec
  ▪ Performance optimizations
    - E.g., “fast path” vs. “slow path” in code, like when a value is in a cache vs. not
  ▪ If you can, details of error handling
• We actually used glass box testing earlier!
  ▪ Even when looking at just the spec, needed to guess where programmer likely make mistakes
Coverage Criteria

• Common metric for test suite quality: coverage
  ▪ Goal: test suite covers all possible program behaviors
    - If test suite doesn’t cover some behavior, we aren’t testing for bugs in it!
  ▪ Hypothesis: high coverage means few mistakes remain in program

• But what is a behavior? Probably not measurable

• Instead: Structural coverage testing
  ▪ Divide a program into elements (e.g., methods, statements)
  ▪ Coverage of a test suite is

\[
\frac{\text{# of elements executed by suite}}{\text{# elements in program}}
\]
Statement Coverage

int select(int[] a, int n, int x) {
    int i=0;
    while (i<n && a[i]<x) {
        if (a[i]<0) {
            a[i]=-a[i];
        }
        i++;
    }
    return 1
}

- Consider test (n=1 a[0]=-7 x=9)
  - Covers all statements
  - But, doesn’t consider case where a[i]<0
Condition Coverage

`int select(int[] a, int n, int x) {
    int i=0;
    while (i<n && a[i]<x) {
        if (a[i]<0) {
            a[i]=-a[i];
        }
        i++;
    }
    return 1;
}

• Add test (n=1 a[0]=7 x=9)
  - Covers all branches (all edges in the graph)
  - But, for i<n&&a[i]<x, has cases where i<n, i≥n, a[i]<x, but no case where a[i]≥x is checked
  - I.e., the branches due to short-circuiting are not covered
Path Coverage

• Execute every path through the program
  - Challenge 1: Which paths are *realizable*, i.e., could occur at runtime
    - Often not obvious from looking at the program text
    - So it’s hard to know how many of the possible paths have been covered
  - Challenge 2: Acyclic programs can have exponential number of paths
    - `if(...) {...} else {...}; if(...) {...} else {...}; if(...) {...} else {...};` has eight paths
  - Challenge 3: Programs with loops might have an unbounded number of paths
    - E.g., a program that reads data from the network and processes it in a loop
  ⇒ Path coverage is not a common metric
Code Coverage Limitations

- Code coverage seems to work well in practice
  - And test suites with low coverage are probably bad in other ways
- But, 100% coverage does not mean no bugs
  - And, 100% coverage not achievable in practice
    - Common to reach 85% coverage
    - Safety-critical software should get 100% statement coverage (feasible)
  - Are remaining statements unreachable (dead code)? Just hard to get to? Hard to know for sure.
- Reality: time and money are limited
  - Should we spend money testing code or adding new features that our customers want?
- Where should we direct testing effort?
  - “High risk” code (= bugs could cause severe damage)
In Practice...

- Statement coverage is most common criterion used
  - Many coverage tools provide *basic block* coverage
    - Basic block = sequence of non-branching statements that can only be entered from the first statement

- *Branch coverage* is another kind of coverage, related to condition coverage
  - Condition = separate && and || into their own branches, branch = treat the ... in if (...) as a block

- *Modified condition/decision coverage* is a more complicated version of condition coverage, sometimes comes up
Two Rules of Testing

1. Test early and often
   - Catch bugs as soon as possible after a code change
   - Use automation to make running tests simple
   - Regression testing has a big payoff
     - When you find a bug: (1) write a test for it; (2) show the test fails; (3) fix the bug; (4) show the test passes

2. Be systematic
   - Bugs will hide in whatever you don’t test, so try to test as much as you can
   - Make your tests as small and separate as possible so that when they fail, you can figure out what happened
   - Writing tests helps you think about the programming problem you’re trying to solve
Refactoring
Motivation

- Old-style software design process: “waterfall model”

- Developed and popularized in 1970’s–1980’s
Pros and Cons of Waterfall Model

• Some good properties
  ▪ Provides structure to the software engineering process
  ▪ Lots of emphasis of careful thought and design early on

• But, critical bad properties
  ▪ Requirements often not known in advance
    - Customers don’t really know what they want
  ▪ Designs often need to be changed
    - Changing requirements
    - Implementation challenges due to unforeseen design issues

• Result: Strict adherence to waterfall leads to inappropriate designs
  ▪ Makes code harder to understand, maintain, and extend
Refactoring

• New approach to software design (*part of extreme programming*)
  - Come up with a reasonable first design
  - But then be willing to change and evolve design over time

• *Refactoring* enables safe design changes
  - Assumption: we have a comprehensive, automatically runnable test suite
  - Then we can divide code changes into two sorts:
    - Bug fixes or feature additions that modify functionality
      - We expect some tests to break; fix them and add new tests
    - Refactoring
      - Change code design, but do not change behavior
        - All refactorings should be *semantics preserving*
      - Implies can rerun all exists tests to ensure change works!
Replace Number with Constant

- New code is more readable
- Can avoid typos if we reuse magic number several times
- Might want to add more digits of pi later

```java
double area(double r) {
    return 3.14 * r * r;
}
```

```java
static final double pi = 3.14;
double area(double r) {
    return pi * r * r;
}
```
Other Refactorings

• Lots of refactoring options available.
  ▪ Some IDEs even have support for automating refactoring.

• Examples
  ▪ Move method from one class to another.
  ▪ Move method from a subclass to a superclass or the reverse.
  ▪ Extract code sequence into its own method.
  ▪ Replace conditional branching with dynamic dispatch.
  ▪ Group together a long parameter list into an object containing those values.
Code Smells

• *Code smells* are coding patterns or idioms that suggest something might be wrong here
  ▪ I.e., might point to code that should be refactored

• Examples
  ▪ Every time I make change X, I have to make lots of little changes to different classes
  ▪ There’s a class but it doesn’t seem to be useful any more
  ▪ The code has excess generality that’s not currently used
  ▪ Classes rely on too many details of each other
  ▪ Subclass doesn’t use features of superclass
Practicalities

• Ideal time to refactor: When you want to make a change and the current design impedes the change
  ▪ Much easier to do cost-benefit analysis of refactoring

• Be cautious of refactoring buggy code
  ▪ The refactoring might not go as planned if you don’t truly understand the bug

• Be cautious if you keep refactoring the same code over and over
  ▪ Probably need to do more careful thinking of its design

• Refactorings tend to be small changes; so they can’t fix every problem
  ▪ If the design is really bad, sometimes you just need to throw it away and start again