L2: Test-First Programming

Today
  o Testing
  o Choosing test cases
  o Blackbox vs. whitebox testing
  o Coverage
  o Stubs, drivers, oracles
  o Regression testing

Real Programmers Don’t Test (?

Here are the top-5 reasons why Louis Reasoner doesn’t want to test his code:

5) I want to get this done fast – testing is going to slow me down.
4) I started programming when I was 2. Don’t insult me by testing my perfect code!
3) Testing is for incompetent programmers who cannot hack.
2) We’re not Harvard students – our code actually works!
1) “Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them.” – an excerpt from a 6.170 student’s e-mail

The biggest problem for Louis is optimism. Change that perspective – look for things that can go wrong, rather than assuming Pollyannishly that all will go right.

Testing isn’t rocket science. Except when it is: a famous case was the Ariane 5 launch vehicle, designed and built for the European Space Agency in the 1990s. It self-destructed 37 seconds after its first launch. The reason was a control software bug that went undetected. Ariane 5’s guidance software was reused from the Ariane 4, which was a slower rocket. As a result, when the velocity calculation converted from a 64-bit floating point number (a double in Java terminology, though this software wasn’t written in Java) to a 16-bit signed integer (a short), it overflowed the small integer and caused an exception to be thrown. The exception handler had been disabled for efficiency reasons, so the guidance software crashed... and without guidance, the rocket did too. The cost of the failure was $1 billion... a cost that might have been prevented with better testing.

In PS1, we have a sequence of steps feeding each other; a stack of methods building on each other. The optimistic hacker’s approach to PS1 might build the whole thing before testing any part of it. Contrast with the engineer’s approach: isolate each component, build it and test it soundly in isolation, then plug together.

Why Testing is Hard

we want to
   know when product is stable enough to launch
   deliver product with known failure rate (preferably low)
   offer warranty?

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright © MIT.
but

- it’s very hard to measure or ensure quality in software
- residual defect rate after shipping:
  - 1 - 10 defects/kloc (typical)
  - 0.1 - 1 defects/kloc (high quality: Java libraries?)
  - 0.01 - 0.1 defects/kloc (very best: Praxis, NASA)

example: 1Mloc with 1 defect/kloc means you missed 1000 bugs!

exhaustive testing is infeasible

- space is generally too big to cover exhaustively
- imagine exhaustively testing a 32-bit floating-point multiply operation, $a \times b$
  - there are $2^{64}$ test cases!

statistical testing doesn’t work for software

- other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer defect rate for whole lot
- many tricks to speed up time (e.g. opening a refrigerator 1000 times in 24 hours instead of 10 years)
- gives known failure rates (e.g. mean lifetime of a hard drive)
- but assumes continuity or uniformity across the space of defects, which is true for physical artifacts
- **this is not true** for software
  - overflow bugs (like Ariane 5) happen abruptly
  - Pentium division bug affected approximately 1 in 9 billion divisions

often confused, but very different

(a) problem of **finding** bugs in defective code
(b) problem of showing **absence** of bugs in good code

approaches

- testing: good for (a), occasionally (b)
- reasoning: good for (a), also (b)

theory and practice

- for both, you need grasp of basic theory
- good engineering judgment essential too

what are we trying to do?

- find bugs as cheaply and quickly as possible

reality vs. ideal

- ideally, choose one test case that exposes a bug and run it
- in practice, have to run many test cases that “fail” (because they don’t expose any bugs)

in practice, conflicting desiderata

- increase chance of finding bug
- decrease cost of test suite (cost to generate, cost to run)

**design testing strategy carefully**

- know what it’s good for (finding egregious bugs) and not good for (security)
- complement with other methods: code review, reasoning, static analysis
- exploit automation (e.g. JUnit) to increase coverage and frequency of testing
- do it early and often: **test-first programming**
Basic Notions

what’s being tested?
- unit testing: individual module (method, class, interface)
- subsystem testing: entire subsystems
- integration, system, acceptance testing: whole system

how are inputs chosen?
- random: surprisingly effective (in defects found per test case), but not much use when most inputs are invalid (e.g. URLs)
- systematic: partitioning large input space into a few representatives
- arbitrary: not a good idea, and not the same as random!

how are outputs checked?
- automatic checking is preferable, but sometimes hard (how to check the display of a graphical user interface?)

how good is the test suite?
- coverage: how much of the specification or code is exercised by tests?

when is testing done?
- test-first programming: tests are written first, before the code
- regression testing: a new test is added for every discovered bug, and tests are run after every change to the code

essential characteristics of tests
- modularity: no dependence of test driver on internals of unit being tested
- automation: must be able to run (and check results) without manual effort

Choosing Test Cases

arbitrary testing is not convincing
- “just try it and see if it works“ won’t fly

exhaustive testing is not feasible
- even a simple int → int function requires billions of runs to test all inputs

key problem: choosing a test suite systematically
- small enough to run quickly
- large enough to validate the program convincingly

Key Idea #1: Partition the Input Space
input space is very large, but program is small
- so behavior must be the “same” for whole sets of inputs
ideal test suite

- identify sets of inputs with the same behavior
- try one input from each set

multiply : BigInteger x BigInteger \rightarrow BigInteger

- partition BigInteger into:
  \[ \text{BigNeg, SmallNeg, -1, 0, 1, SmallPos, BigPos} \]
- pick a value from each class
  \[ -265, -9, -1, 0, 1, 9, 265 \]
- test the \( 7 \times 7 = 49 \) combinations

max : int x int \rightarrow int

- partition into:
  \[ a < b, a = b, a > b \]
- pick value from each class
  \[ (1, 2), (1, 1), (2, 1) \]

intersect : Set x Set \rightarrow Set

- partition Set into:
  \[ \emptyset, \text{singleton, many} \]
- partition whole input space into:
  this = that, this \( \subseteq \) that, this \( \supseteq \) that, this \( \cap \) that \( \neq \) \( \emptyset \), this \( \cap \) that = \( \emptyset \)
- pick values that cover both partitions
  \[ \{\},{} \quad \{\},\{2\} \quad \{\},\{2,3,4\} \]
  \[ \{5\},{} \quad \{5\},\{2\} \quad \{4\},\{2,3,4\} \]
  \[ \{2,3\},{} \quad \{2,3\},\{2\} \quad \{1,2\},\{2,3\} \]

Key idea #2: Boundary testing

- include classes at boundaries of the input space
  - zero, min/max values, empty set, empty string, null
- why? because bugs often occur at boundaries
  - off-by-one errors
- forget to handle empty container
- overflow errors in arithmetic

Blackbox vs. whitebox

**black box testing**
- choosing test data only from spec, without looking at implementation

**glass box (white box) testing**
- choosing test data with knowledge of implementation
  - e.g. if implementation does caching, then should test repeated inputs
  - if implementation selects different algorithms depending on the input, should choose inputs that exercise all the algorithms
- must take care that tests don’t depend on implementation details
  - e.g. if spec says “throws exception if the input is poorly formatted”, your test shouldn’t check specifically for a NullPointerException just because that’s what the current implementation does
- good tests should be modular -- depending only on the spec, not on the implementation

```java
public static int max(List<Integer> l) { ... }
```

**Coverage**

Three kinds of coverage:

- all-statements: is every statement run by some test case?
- all-branches: if every direction of an if or while statement (true or false) taken by some test case?
- all-paths: is every possible combination of branches – every path through the program – taken by some test case?

specification coverage vs. code coverage
a typical code coverage tool (EclEmma plugin for Eclipse):

---

industry practice

- all-statements is common goal, rarely achieved (due to unreachable code)
- all branches if possible: safety critical industry has more arduous criteria (e.g., “MCDC”, modified decision/condition coverage)
- all-paths is infeasible

using coverage for white-box testing

- generate black-box test cases until code coverage is sufficient

Test frameworks

driver

- just runs the tests
- must design unit to be drivable!
- eg: program with GUI should have API

stub

- replaces other system components
- allows reproducible behaviours (esp. failures)

oracle

- determines if result meets spec
- preferably automatic and fast
varieties: computable predicate (e.g. is the result odd?), comparison with literal (e.g. must be 5), manual examination (by a human)

in regression testing, can use previous results as “gold standard”

Test-first programming

write tests before coding

specifically, for every method or class:
1) write specification
2) write test cases that cover the spec
3) implement the method or class
4) once the tests pass (and code coverage is sufficient), you’re done

writing tests first is a good way to understand the spec

think about partitioning and boundary cases

if the spec is confusing, write more tests

spec can be buggy too
  • incorrect, incomplete, ambiguous, missing corner cases
  • trying to write tests can uncover these problems

Regression Testing

whenever you find and fix a bug

store the input that elicited the bug

store the correct output

add it to your test suite

why regression tests help

helps to populate test suite with good test cases
  • remember that a test is good if it elicits a bug – and every regression test did
    in one version of your code

protects against reversions that reintroduce bug

the bug may be an easy error to make (since it happened once already)

test-first debugging

when a bug arises, immediately write a test case for it that elicits it

once you find and fix the bug, the test case will pass, and you’ll be done
How to avoid debugging

first defense against bugs is to make them impossible

- Java makes buffer overflow bugs impossible
- static typing eliminates many runtime type errors
- immutable objects like Strings and URLs can be passed around and shared without fear that they will be modified
- immutable (final) references guarantee that you won’t

if we can’t prevent bugs, we can try to localize them to a small part of the program

- fail fast: the earlier a problem is observed, the easier it is to fix
- assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation
  - in Java: `assert boolean-expression`
  - note that you must enable assertions with -ea
- unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (or in the test driver)
- regression testing: run tests as often as possible when changing code.
  - if a test fails, the bug is probably in the code you just changed

when localized to a single method or small module, bugs may be found simply by studying the program text

Code review

other eyes looking at the code can find bugs

code review

careful, systematic study of source code by others (not original author)
analogous to proofreading an English paper
look for bugs, poor style, design problems, etc.
formal inspection: several people read code separately, then meet to discuss it
lightweight methods: over-the-shoulder walkthrough, or by email
many dev groups require a code review before commit

code review complements other techniques

code reviews can find many bugs cheaply
also test the understandability and maintainability of the code
three proven techniques for reducing bugs: static checking, code reviews, testing
An example of test-first programming

First, the spec:

```java
/**
 * Find the first occurrence of x in sorted array a.
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {...}
```

Then, the test cases:

```java
// Testing strategy for i = search(x, a):
//     partition the space of (x, a, i) as follows
// x: neg, 0, pos
// a.length: 0, 1, 2+
// a.vals: neg, 0, pos; all same, increasing;
// i: 0, middle, n-1, -1

• // x=2, a=[-1, 1, 3], i=-1
• // x=0, a=[0], i=0
• // x=1, a=[-1, 1, 3], i=1
• // x=-1, a=[-1, 1, 3], i=0
• // x=3, a=[-1, 1, 3], i=n-1
• // x=2, a=[-1, 1, 3], i=-1
• // x=1, a=[1, 1, 1], i=0
```

Then, a simple implementation: (tests the tests! also gives us a slow oracle in case we need it later)

```java
/**
 * Find the first occurrence of x in sorted array a.
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {
    for (int i = 0; i < a.length; ++i) {
        if (x == a[i]) {
            return i;
        }
    }
    return -1;
}
```

Now, some attempts at the real binary-search implementation. Let’s do it recursively:
public static int search(int x, int[] a) {
    int mid = a.length/2;
    if (x < a[mid]) {
        binarySearch(x, left half of a)
    } else if (x > a[mid]) {
        binarySearch(x, right half of a)
    } else {
        return mid; // because x == a[mid], i.e. we found it!
    }
}

OK, we need to strengthen the induction hypothesis to avoid copying the array:

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}

Broken! Let’s adjust the spec of our private method:

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}

Broken! Let’s adjust the spec of our private method:

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}
private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    ...
}

public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length-1);
}

And now add a test for an empty range:

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    if (lo >= hi) {
        return -1; // range has dwindled to nothingness
    }

    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}

Still broken! Not handling the case of multiple matches correctly. Let's fix:

    } else {
        // x == a[mid]... we found it, but check if it's the first
        if (x == a[mid-1]) {
            // not the first! search lower half
            return binarySearchInRange(x, a, lo, mid);
        } else {
            return binarySearchInRange(x, a, lo, mid);
        }
    }
}

Still broken! In fact, we've experienced a regression – tests that used to be passing are now failing because of our change. Regressions happen! Run all your test cases after every change you make!

Need to be careful about a[mid-1], because mid might be 0. Finally:

    } else {
        // x == a[mid]... we found it, but check if it's the first
        if (mid > 0 && x == a[mid-1]) {
            // not the first! search lower half
            return binarySearchInRange(x, a, lo, mid);
        } else {
            return mid; // it's the first
        }
    }
}

Here's the final code. Left as an exercise to the reader to tidy up binarySearchInRange() and make it simpler and more compact. The fact that you have a working test suite makes this tidying far safer!
Without a test suite, attempts to simplify logic can easily introduce bugs. Just run it after every little change you make.

```java
public static int search(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length-1);
}

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    if (lo >= hi) {
        return -1; // range has dwindled to nothingness
    }

    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it, but check if it's the first
        if (mid > 0 && x == a[mid-1]) {
            // not the first! search lower half
            return binarySearchInRange(x, a, lo, mid);
        } else {
            return mid; // it's the first
        }
    }
}
```

**Summary**

Thinking about our three main measures of code quality (safe from bugs, easy to understand, ready for change), the techniques in this lecture have mainly addressed safety. Readiness for change was considered by writing tests that only depend on behavior in the spec. Testing doesn't measure or improve ease of understanding, but we talked about code review, which does.

**testing matters**
- you need to convince others (and yourself) that your code works
- testing generally can't prove absence of bugs, but can increase quality by reducing bugs

**test early and often**
- unit testing catches bugs before they have a chance to hide
- automate the process so you can run it frequently
- regression testing will save time in the long run

**be systematic**
- use input partitioning, boundary testing, and coverage
- regard testing as a creative design problem

**use tools and build your own**
- automated testing frameworks (JUnit) and coverage tools (EclEmma)
- design modules to be driven, and use stubs for repeatable behavior