Unit Testing

Emina Torlak
emina@cs.washington.edu
Outline

- Software quality control
- Effective unit testing
- Coverage and regression testing
basics of software quality control
Errors and faults

Errors and faults

• **Error**: incorrect software behavior
  • Example: Software controller for the Ariane 5 rocket crashed (and so did the rocket).

Errors and faults

• **Error**: incorrect software behavior
  - Example: Software controller for the Ariane 5 rocket crashed (and so did the rocket).

• **Fault**: mechanical or algorithmic cause of error (bug)
  - Example: Conversion from 64-bit floating point to 16-bit signed integer value caused an exception.
  - Requirements specify desired behavior; if the system deviates from that, it has a fault.

Software quality control techniques

- **Fault avoidance**: prevents errors before the system is released.
  - reviews, inspections, walkthroughs, development methodologies, **testing, verification**

- **Fault tolerance**: enables the system to recover from (some classes of) errors by itself.
  - rollbacks, redundancy, mirroring
Showing the presence and absence of bugs ...
Showing the presence and absence of bugs ... 

Detects the presence of bugs by running the code on a few carefully chosen inputs.
Showing the presence and absence of bugs ...

Detects the presence of bugs by running the code on a few carefully chosen inputs.

Shows the absence of bugs on all possible inputs.

testing

verification
Common kinds of testing

- **Unit testing**: tests the behavior of an individual module (method, class, interface)
  - Black-box testing
  - White-box testing
- **System testing**: tests the behavior of the system as a whole, with respect to scenarios and requirements
  - Functional testing, integration testing
  - Performance, load, stress testing
  - Acceptance, usability, installation, beta testing
effective unit testing
Two rules of unit testing

• Do it **early** and do it **often**
  • Catch bugs quickly, before they have a chance to hide
  • **Automate** the process if you can

• Be **systematic**
  • If you thrash about arbitrarily, the bugs will hide in the corner until you're gone
Four basic steps of a test

1. Choose input data
   - *without* looking at the implementation: black box
   - *with* knowledge of the implementation: white box

2. Define the expected outcome

3. Run on the input to get the actual outcome

4. Compare the actual and expected outcomes
Four basic steps of a test

1. **Choose input** data
   - *without* looking at the implementation: black box
   - *with* knowledge of the implementation: white box
2. Define the **expected outcome**
3. Run on the input to get the **actual outcome**
4. **Compare** the actual and expected outcomes

This is hard! Need a set of test cases that is small enough to run quickly, yet large enough to cover [all] interesting program behaviors.
Choosing inputs: two key ideas
Choosing inputs: two key ideas

- Partition the input space
  - Identify **subdomains** with the same behavior
  - Pick one input from each subdomain
Choosing inputs: two key ideas

• Partition the input space
  • Identify subdomains with the same behavior
  • Pick one input from each subdomain

• Boundary values
  • Pick inputs at the edges of the subdomains.
  • Effective at finding corner case bugs:
    • off-by-one, overflow, aliasing, empty container
Partitioning the input space

// returns the maximum of a, b
public static int max(int a, int b) { ... }

• Partition into
  • a < b, a = b, a > b

• Pick an input from each class
  • (1, 2), (0, 0), (2, 1)
Partitioning the input space

// returns the maximum of a, b
public static int max(int a, int b) { ... }

• Partition into
  • a < b, a = b, a > b

• Pick an input from each class
  • (1, 2), (0, 0), (2, 1)

How would you partition the input space for
• BigInteger multiplication?
• Set intersection?
Choosing boundary values

// returns |x|
public static int abs(int a) { … }

• Partition into
  • $a < 0$, $a > 0$, $a = 0$ (boundary)

• Other boundary values
  • Integer.MAX_VALUE
  • Integer.MIN_VALUE
Choosing boundary values

// returns |x|
public static int abs(int a) { ... }

• Partition into
  • $a < 0, a > 0, a = 0$ (boundary)

• Other boundary values
  • Integer.MAX_VALUE
  • Integer.MIN_VALUE
Black box testing

• Explores alternate paths through the specification.
  • Module under test is a black box: interface visible, internals hidden.

// If a >= b, returns a. Otherwise returns b.
public static int max(int a, int b) { ... }

• 3 paths, so 3 subdomains
  • (1, 2) => 2
  • (2, 1) => 2
  • (0, 0) => 0
Advantages of black box testing

• Process is not influenced by component being tested
  • Assumptions embodied in code not propagated to test data.

• Robust with respect to changes in implementation
  • Test data need not be changed when code is changed

• Allows for independent testers
  • Testers need not be familiar with code
Disadvantage of black box testing

• It will miss bugs in the implementation that are not covered by the specification
  • Control-flow details
  • Performance optimizations
  • Alternate algorithms for different cases
White box testing

- Explores alternate paths through the implementation
  - Module under test is a clear box: internals visible.

```java
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
    if (x>CACHE_SIZE) {
        for (int i=2; i<x/2; i++) {
            if (x%i==0) return false;
        }
        return true;
    } else {
        return primeTable[x];
    }
}
```

- Important transition at around $x = \text{CACHE\_SIZE}$
(Dis)advantages of white box testing
(Dis)advantages of white box testing

• Advantages
  • Finds an important class of boundaries.
  • Yields useful test cases.
  • In isPrime example, need to check numbers on each side of CACHE_SIZE
    • CACHE_SIZE-1, CACHE_SIZE, CACHE_SIZE+1
(Dis)advantages of white box testing

• Advantages
  • Finds an important class of boundaries.
  • Yields useful test cases.
  • In isPrime example, need to check numbers on each side of CACHE_SIZE
    • CACHE_SIZE-1, CACHE_SIZE, CACHE_SIZE+1

• Disadvantages
  • Tests may have the same bugs as implementation!
Properties of good and bad unit tests

• Tests should be self-contained and not depend on each other implicitly or explicitly.

• "Smells" (bad things to avoid) in tests:
  • Constrained test order
    • Test A must run before Test B.
  • Tests call each other
    • Test A calls Test B.
  • Mutable shared state
    • Tests A/B both use a shared object.
coverage and regression testing
Measuring test suite quality with coverage
Measuring test suite quality with coverage

• Various kinds of coverage
  • **Statement**: is every statement run by some test case?
  • **Branch**: is every direction of an if or while statement (true or false) taken by some test case?
  • **Path**: is every path through the program taken by some test case?
Measuring test suite quality with coverage

• Various kinds of coverage
  • **Statement**: is every statement run by some test case?
  • **Branch**: is every direction of an if or while statement (true or false) taken by some test case?
  • **Path**: is every path through the program taken by some test case?

• Limitations of coverage
  • Coverage is just a heuristic.
  • 100% coverage may not be achievable.
  • High-cost to approach the limit.
Measuring test suite quality with coverage

• Various kinds of coverage
  • **Statement**: is every statement run by some test case?
  • **Branch**: is every direction of an if or while statement (true or false) taken by some test case?
  • **Path**: is every path through the program taken by some test case?

• Limitations of coverage
  • Coverage is just a heuristic.
  • 100% coverage may not be achievable.
  • High-cost to approach the limit.

We will ask you to provide test-suite coverage metrics for your Feature-Complete Release.
Coverage measuring tools: EclEmma
Regression testing
Regression testing

- Whenever you find a bug
  - Store the input that elicited that bug, plus the correct output
  - Add these to the test suite
  - Check that the test suite fails
  - Fix the bug and verify the fix
Regression testing

• Whenever you find a bug
  • Store the input that elicited that bug, plus the correct output
  • Add these to the test suite
  • Check that the test suite fails
  • Fix the bug and verify the fix

• Why is this a good idea?
  • Ensures that your fix solves the problem.
  • Helps to populate test suite with good tests.
  • Protects against reversions that reintroduce bug:
    • It happened at least once, and it might happen again
Summary

• Unit testing helps
  • convince others that a module works;
  • catch problems earlier.

• Choose test data to cover
  • specification (black box testing)
  • code (white box testing)

• Testing can’t generally prove the absence of bugs, but it can increase quality and confidence in the implementation.