The BUGS Lecture
Avoiding, Finding and Eliminating Bugs

Saman Amarasinghe
Spring 2013
Today’s Topics

Impact of Bugs
Avoiding Bugs
Detecting Bugs: Testing
Killing the Bugs: Debugging
Very Hard Bugs
Real Programmers Don’t do Bugs!

Reasons why real programmers don’t worry about bugs

5) Bugs happens to incompetent programmers who cannot hack.
4) I started programming when I was 2. Don’t insult me!
3) we’re not Harvard students – our code actually works!
2) I want to get this done fast – testing and defensive programming is going to slow me down.
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
BUGS ARE EVERYWHERE!
AND THEY CAN DO BAD THINGS
Who Says Software is Buggy?

Ariane 5: most technically advanced rockets in the world

Ariane 5 self-destructed 37 seconds after launch
reason: a control software bug that went undetected

- conversion from 64-bit floating point to 16-bit signed integer caused an exception
  - because the value was larger than 32767 (max 16-bit signed integer)
- but the exception handler had been disabled for efficiency reasons
- software crashed ... rocket crashed ... total cost over $1 billion
Another Prominent Software Bug

Mars Polar Lander crashed

- sensor signal falsely indicated that the craft had touched down when it was still 130 feet above the surface.
- descent engines shut down prematurely... and it was never heard from again.

the error was traced to a single bad line of code

- Prof. Nancy Leveson: these problems "are well known as difficult parts of the software-engineering process"... and yet we still can’t get them right.
The Opportunity and the Challenge

Programs don’t have to obey any physical laws
➢ You are free to do anything!

When there are no laws, anarchy follows!
➢ You have to be really careful

In building a large physical system
➢ The laws of nature can help isolate and compartmentalize the world

No such luck in programming systems!
The Challenge

**we want to**

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

**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: NASA)
- example: 1 Mloc with 1 defect/kloc means you have 1000 bugs!
Defensive Programming

first defense against bugs is to make them impossible

- Java makes buffer overflow bugs impossible

second defense against bugs is to not make them

- correctness: get things right first time

third defense is to make bugs easy to find

- local visibility of errors: if things fail, we'd rather they fail loudly and immediately – e.g. with assertions

fourth defense is extensive testing

- uncover as many bugs as possible

last resort is debugging

- needed when effect of bug is distant from cause
The situation: bugs fall down from the kitchen ceiling and end up in a pot with soup.

Means of maintaining quality:

1. Check soup for bugs. If a bug found, remove it, or pour out the whole pot.
2. Keep the pot’s lid closed most of the time, in such a way minimizing the possibility for bugs to fall into the pot.
3. Clean the kitchen.

Variant 1 – reactive quality through testing of software
Variant 2 – built-in quality through defensive programming
Variant 3 – proactive quality through type safe language
AVOIDING BUGS
First Defense: Impossible By Design

in the language
- automatic array bounds checking make buffer overflow bugs impossible
- static typing eliminates many runtime type errors

in the protocols/libraries/modules
- TCP/IP guarantees that data is not reordered
- ArrayList can grow arbitrarily, while an ordinary array has a fixed length

in self-imposed conventions
- immutable objects like Strings and URLs can be passed around and shared without fear that they will be modified
- caution: you have to keep the discipline
  - get the language to help you as much as possible
Second Defense: Correctness

get things right the first time
- don’t code before you think! Think before you code.
  - do your thinking in design; use a pattern to map that design to code
especially true when debugging is going to be hard
- concurrency

simplicity is key
- modularity
  - divide program into chunks that are easy to understand and independent
- specification
  - write specs for all methods, so that an explicit, well-defined contract exists between each method and its clients

© Robert Miller 2009
Third Defense: Immediate Visibility

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

- when localized to a single method or small module, bugs may be found simply by studying the program text
- 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
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: reasoning, code reviews, testing

© Robert Miller 2009
TESTING
Why Test?

Programmers are Human
➢ They make mistakes
➢ They create bugs in their code

Find Them!
➢ Earlier the better

© Robert Miller 2009
Testing Strategies That Don’t Work

exhaustive testing is infeasible
- space is generally too big to cover exhaustively
- imagine exhaustively testing a 32-bit floating-point multiply operation, $a*b$
  - 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
Aims of Testing

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)
**Practical Strategies**

**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
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?)
Basic Notions

how good is the test suite?

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

when is testing done?

- test-first development: 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 TESTS
Example: Thermostat

**specification**

- user sets the desired temperature $T_d$
- thermostat measures the ambient temperature $T_a$
- want heating if desired temp is higher than ambient temp
- want cooling if desired temp is lower than ambient temp

\[
\begin{align*}
\text{if } T_d &> T_a, \text{ turn on heating} \\
\text{if } T_d &< T_a, \text{ turn on air-conditioning} \\
\text{if } T_d &= T_a, \text{ turn everything off}
\end{align*}
\]
How Do We Test the Thermostat?

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

exhaustive testing is not feasible
- would require millions of runs to test all possible (Td,Ta) pairs

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

if $T_d > T_a$, turn on heating
if $T_d < T_a$, turn on air-conditioning
if $T_d = T_a$, turn everything off

© Robert Miller 2009
java.math.BigInteger.add(BigInteger val)

- BigIntegers can hold values bigger than 64-bit
- add method has two arguments, this and val, drawn from BigInteger
More Examples

```java
java.math.BigInteger.add(BigInteger val)
```

- BigIntegers can hold values bigger than 64-bit
- add method has two arguments, this and val, drawn from BigInteger

**Approach 1: partition inputs separately, then form all combinations**

- partition BigInteger into:
  - BigNeg, SmallNeg,, 0, SmallPos, BigPos
- pick a value from each class
  - \(-2^{65}, -1024, 0, 1020, 2^{66}\)
- test the \(5 \times 5 = 25\) combinations
More Examples

static int java.Math.max(int a, int b)

approach 2: partition the whole input space
(useful when relationship between inputs matters)

- partition into:
  - $a < b$, $a = b$, $a > b$

- pick value from each class
  - $(1, 2)$, $(1, 1)$, $(2, 1)$
More Examples

java.math.BigInteger.add(BigInteger val)

- BigIntegers can hold values bigger than 64-bit
- add method has two arguments, this and val, drawn from BigInteger

approach 2: partition the whole input space
(useful when relationship between inputs matters)

- Adding two small numbers (less than 64bit)
  - Result in a small number
  - Or result in a big number

© Robert Miller 2009
More Examples

Set.intersect(Set that)

- partition Set into:
  - $\emptyset$, singleton, many

- partition whole input space into:
  - this $=$ that, this $\subseteq$ that, this $\cap$ that $\neq \emptyset$, this $\cap$ that $= \emptyset$

<table>
<thead>
<tr>
<th></th>
<th>$\emptyset$</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Robert Miller 2009
More Examples

Set.intersect(Set that)

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# More Examples

Set.intersect(Set that)

<table>
<thead>
<tr>
<th></th>
<th>Ø</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>{},</td>
<td>{},</td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>{2},</td>
<td>{2}</td>
<td></td>
</tr>
<tr>
<td>many</td>
<td>{{2},</td>
<td>{2,5},</td>
<td>{2,5}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ø</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>{},</td>
<td>{2}</td>
<td>{2,5}</td>
</tr>
<tr>
<td>one</td>
<td>{2},</td>
<td>{2,5}</td>
<td></td>
</tr>
<tr>
<td>many</td>
<td>{2,5}</td>
<td>{2,5,6}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ø</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td>{2},</td>
<td>{2,5}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ø</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>{2},</td>
<td>{5}</td>
<td>{2,5,6}</td>
</tr>
<tr>
<td>many</td>
<td>{2,3}</td>
<td>{3,4}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ø</th>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>{2},</td>
<td>{5}</td>
<td>{2,5,6}</td>
</tr>
<tr>
<td>many</td>
<td>{2,5}</td>
<td>{6}</td>
<td>{2,3}</td>
</tr>
</tbody>
</table>

© Robert Miller 2009
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
Boundary Testing

```
public int abs(int x)
   // returns: |x|
```

Tests for abs

- what are some values or ranges of x that might be worth probing?
  - \( x < 0 \) (flips sign) or \( x \geq 0 \) (returns unchanged)
  - around \( x = 0 \) (boundary condition)
  - Specific tests: say \( x = -1, 0, 1 \)

How about...

```
int x = -2147483648; // this is Integer.MIN_VALUE
System.out.println(x<0); // true
System.out.println(Math.abs(x)<0); // also true!
```

From Javadoc for Math.abs:

- Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative

© Robert Miller 2009
COVERAGE
Coverage

how good are my tests?

➢ measure extent to which tests ‘cover’ the specification or code

➢ What coverage do you get?
  • All Statements
  • All Branches
  • All Paths
How Far Should You Go?

for code coverage
- all-statements, all-branches: if possible
- all-paths: infeasible

industry practice
- all-statements is common goal, rarely achieved (due to unreachable code)
- safety critical industry has more arduous criteria (eg, “MCDC”, modified decision/condition coverage)
A Typical Statement Coverage Tool

EclEmma Eclipse plugin

coverage statistics for packages and classes
**Black Box vs. Glass Box Testing**

**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 NullPtrException just because that’s what the current implementation does
- good tests should be **modular** -- depending only on the spec, not on the implementation

© Robert Miller 2009
Black Box vs. Glass Box Testing

**best practice**

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

© Robert Miller 2009
Testing Framework

**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 Development

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

© Robert Miller 2009
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
ten test did in one version of your code
  ➢ protects against reversion 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

© Robert Miller 2009
DEBUGGING
How to Debug

1) reproduce the bug with a small test case
   - find a small, repeatable test case that produces the failure (may take effort, but helps clarify the bug, and also gives you something for regression)
   - *don't move on to next step until you have a repeatable test*

2) find the cause
   - narrow down location and proximate cause
   - study the data / hypothesize / experiment / repeat
   - may change code to get more information
   - *don't move on to next step until you understand the cause*

3) fix the bug
   - is it a simple typo, or is it a design flaw? does it occur elsewhere?

4) add test case to regression tests
   - then run regression tests to ensure that the bug appears to be fixed, and no new bugs have been introduced by the fix
Reducing to a Simple Test Case

find simplest input that will provoke bug

- usually not the input that originally revealed existence of the bug
- start with data that revealed bug
- keep paring it down (binary search can help)
- often leads directly to an understanding of the cause

same idea is useful at many levels of a system

- method arguments
- input files
- keystrokes and mouse clicks in a GUI
Example

```java
/**
 * Returns true if and only if s contains t as a substring,
 * e.g. contains("hello world", "world") == true.
 */
public static boolean contains(String s, String t) { ... }
```

- a user discovers that
  contains("Life is wonderful! I am so very very happy all the time ",
  "very happy")
  incorrectly returns false

**wrong approach:**
- try to trace the execution of contains() for this test case

**right approach:**
- first try to reduce the size of the test case
- even better: bracket the bug with a test case that fails and similar test cases that succeed
Finding the Cause

**exploit modularity**
- start with everything, take away pieces until bug goes
- start with nothing, add pieces back in until bug appears

**take advantage of modular reasoning**
- trace through program, viewing intermediate results
- insert assertions targeted at the bug
- design all data structures to be printable (i.e., implement toString())
  - Java arrays are not printable; Java collections (e.g. ArrayList) are
- println is a surprisingly useful and universal tool
  - in large systems, use a logging infrastructure instead of println

**use binary search to speed things up**
- bug happens somewhere between first and last statement
- so do binary search on the ordered set of statements
THE BAD AND THE UGLY
The Ugliest Bugs

we’ve had it easy so far

- sequential, deterministic programs have repeatable bugs

but the real world is not that nice…

- timing dependencies
- unpredictable network delays
- varying processor loads
- concurrent programming with threads

heisenbugs

- nondeterministic, hard to reproduce
- may even disappear when you try to look at it with println or debugger!
TEST FIRST PROGRAMMING
/**
 * 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) {...}
Debugging - the term coined by Grace Hopper working in the Mark II Computer