The BUGS Lecture
Avoiding, Finding and Eliminating Bugs
Saman Amarasinghe
Spring 2014
Origin of the term “debugging”

On September 9, 1947 Grace Hopper, the very first programmer, was writing a program on the Mark II Computer. It stopped working, and after examination, she found the problem.
Topics for today and Friday

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

Reasons why real programmers don’t worry about bugs

5) Bugs happen 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) This is super easy and 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

   Convincing argument, except for one problem….

   ....the program had a bug!
BUGS ARE EVERYWHERE!
AND THEY CAN DO BAD THINGS

© Robert Miller 2009
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

- Control software ported from Ariane 4
- 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

© Robert Miller 2009
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 disciplined

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

**No such luck in programming systems!**

© Robert Miller 2009
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
Means for Quality

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

© Robert Miller 2009
Example courtesy of Katasonov
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
- BigInteger guarantees that there will be no overflow

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
    e.g. with private and final

© Robert Miller 2009
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
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

© Robert Miller 2009
Example: Assertions

```java
/**
 * @param n
 * n must be nonnegative.
 * @return n!, the number of permutations of n objects.
 */
public static int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}

/**
 * @param n
 * @param k
 * Requires 0 <= k <= n.
 * @return \( \binom{n}{k} \), the number of distinct subsets of size k in a set of size n.
 */
public static int combinations(int n, int k) {
    return fact(n) / (fact(k) * fact(n - k));
}
```

where would assertions be usefully added to this code?
Why Test?

Programmers are Human

- They make mistakes
- They create bugs in their code

Find Them!

- Earlier the better
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, \( a \times b \)
  - 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

© Robert Miller 2009
Two Problems

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
Formal reasoning or verification

- Constructs a formal proof that a program is correct, by showing that whenever the preconditions are satisfied, the program always produces a state in which the postconditions are true.
- Verification is tedious to do by hand, and automated tool support for verification is still an active area of research.
- Small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system or the bytecode interpreter in a virtual machine or the anti-lock break system of a car.

Informal reasoning

- Having somebody else carefully read your code to uncover bugs
- In industry, this practice goes by various names: code reviews, code inspection, walkthroughs.
- *Pair programming* is an extreme form of this idea

© Robert Miller, Saman Amarasinghe, Max Goldman 2009-2014
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 is output 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 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 TESTS

© Robert Miller 2009
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

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
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 \((T_d, T_a)\) 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
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
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 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
  - $(1, 2), (1, 1), (2, 1)$

© Robert Miller 2009
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
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>

\(\circ\) 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>∅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>∅</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>∅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>∅</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>∅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>∅</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>∅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>∅</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>

© Robert Miller 2009
### 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}, {2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td></td>
<td>{2,5}, {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>{}, {2}</td>
<td>{}, {2,5}</td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>{2}, {2,5}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td>{2,5}, {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>{2,3}, {3,4}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>many</td>
<td></td>
<td>{2,5}, {6}</td>
<td>{2,3}, {5,6,7}</td>
</tr>
</tbody>
</table>
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

```java
/* @param a *
 * @return |a|
 */

public static int abs(int a) {
    ...
}
```

**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...**

```java
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