MIT OpenCourseWare
http://ocw.mit.edu

6.005 Elements of Software Construction
Fall 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

6.005 elements of software construction

Debugging

Rob Miller
Fall 2008

© Robert Miller 2008

Today's Topics

how to avoid debugging

- assertions
- code reviews

how to do it when you have to

- reducing test cases
- hypothesis-driven debugging
- binary search

very hard bugs

- Heisenbugs

© Robert Miller 2008

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

© Robert Miller 2008

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
- BigInteger guarantees that there will be no overflow

in self-imposed conventions

- immutable objects can be passed around and shared without fear
- 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 2008

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
  - use abstract data types with well-defined interfaces
  - avoid rep exposure
- specification
  - write specs for all modules, so that an explicit, well-defined contract exists between each module and its client

© Robert Miller 2008

Third Defense: Immediate Visibility

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 can be found simply by studying the program text

© Robert Miller 2008

Example: Assertions

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

/*
 * Returns (n choose k), the number of distinct subsets
 * of size k in a set of size n.
 * Requires 0 <= k <= 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?

© Robert Miller 2008

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 2008

Let's Review Some Code

```
public class PigLatin {
    static String[] words;

    public static String toPigLatin(String s) {
        words = s.split(" ");

        String result = "";
        for (int i = 0; i <= words.length; ++i) {
            piggify(i);
            result += words[i];
        }
        return result;
    }

    public static void piggify(int i) {
        if (words[i].startsWith("a") || words[i].startsWith("e") || ...) {
            words[i] += "yay";
        } else {
            words[i] = words[i].substring(1);
            words[i] += words[i].charAt(0) + "ay";
        }
    }
}
```

© Robert Miller 2008

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

© Robert Miller 2008

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

© Robert Miller 2008

Example

```
/**
 * 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

© Robert Miller 2008

Code for contains()

```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) {
    search:
    for (int i = 0; i < s.length(); ++i) {
        for (int j = 0; j < t.length(); ++j, ++i) {
            if (s.charAt(i) != t.charAt(j)) continue search;
        }
        return true;
    }
    return false;
}
```

© Robert Miller 2008

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())
- println is a surprisingly useful and universal tool
  - in large systems, use a logging infrastructure instead of println

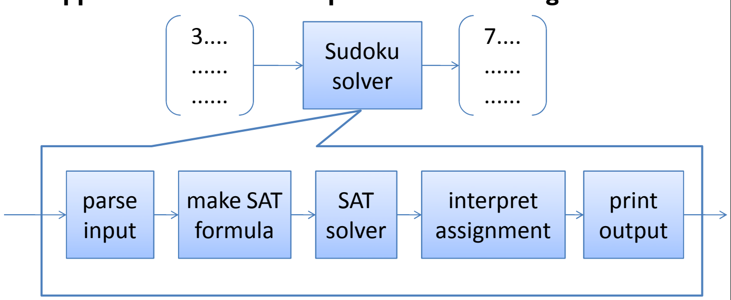
use binary search to speed things up

- bug happens somewhere between first and last statement
- so do binary search on the ordered set of statements

© Robert Miller 2008

Example: Finding a Sudoku Bug

suppose a Sudoku solver produces the wrong answer



Note that this isn't a state machine diagram or a module dependence diagram; it shows **data flow**, which is often useful for thinking about bugs.

© Robert Miller 2008

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

© Robert Miller 2008

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
- concurrency

heisenbugs

- nondeterministic, hard to reproduce
- may even disappear when you try to look at it with println or debugger!

one approach

- build a lightweight event log (circular buffer)
- log events during execution of program as it runs at speed
- when you detect the error, stop program and examine logs

© Robert Miller 2008

Example of a heisenbug

```
public class Bank {
    int balance;

    public Bank(int balance) {
        this.balance = balance;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }
}
```

© Robert Miller 2008

Example of a heisenbug

```
// our bank account starts with $100
final Bank account = new Bank(100);
// start a bunch of threads
List<Thread> threads = new ArrayList<Thread>();
for (int i = 0; i < 10; ++i) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            // each thread does a bunch of bank transactions
            for (int i = 0; i < 10000; ++i) {
                account.deposit(1);  // put a dollar in
                account.withdraw(1); // take it back out
            }}});
    t.start(); // don't forget to start the thread!
    threads.add(t); }
// wait for all the threads to finish
for (Thread t: threads) t.join();
// display the final account balance
System.out.println(account.getBalance());
```

© Robert Miller 2008

Summary

avoid debugging

- it's not fun and not productive
- many of the techniques of this class are designed to save you from bugs

approach it systematically

- simplify test cases
- find cause before trying to fix

© Robert Miller 2008