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

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

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
  - ByteBuddy 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

Second Defense: Correctness

- get things right the first time
  - do code before you think! Think before you code.
  - do your thinking in design use a pattern to map that design to code
  - think about corner cases
  - when in doubt, throw an exception
    - 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
  - 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)
    - in Java: assert boolean-expression
  - 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
Example: Assertions

```java
/*
* 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);
}
```

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

Let’s Review Some Code

```java
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";
        }
    }
}
```

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 case
   - narrow down location and proximate cause
   - study the data / hypothesize / experiment / repeat
   - may change code to get more information
   - Version control is essential for this!
   - don’t move on to next step until you understand the case

3) fix the bug
   - is it a simple typo, or is it a design flaw? does it occur elsewhere?
   - add test case to regression tests

4) add test case to regression tests

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
- Version control is essential for this!
- don’t move on to next step until you understand the case

Delta Debugging

Localize the bug by focusing on small changes
- Small changes to the input
   - find two inputs that are very similar, but where one input succeeds and the other one doesn’t
   - the difference between the two executions will point you to the bug
- Small changes to the code
   - if small change to the code caused the bug to appear, you can focus on the code that changed
     - This doesn’t mean the bug will be in the recently changed code!
     - the change could have uncovered a latent bug somewhere else
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) {
    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;
```

Finding the Cause

- **explore 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**

Example: Finding a Web Browser Bug

Suppose a web browser displays the wrong output

![Web browser diagram](image)

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
- a test is good if it elicits a bug — and every regression test did in one version of your code
- provides against regressions that reintroduce bug

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

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?

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
Example of a heisenbug

```java
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;
    }
}
```

---

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);
        }
    });
    t.start(); // don't forget to start the thread!
}
// wait for all the threads to finish
for (Thread t : threads) t.join();
// display the final account balance
System.out.println(account.getBalance());
```