Regression testing

- Whenever you find a bug
  - Reproduce it (before you fix it!)
  - Store input that elicited that bug
  - Store correct output
  - Put into test suite
  - Then, fix it and verify the fix
- Why is this a good idea?
  - Helps to populate test suite with good tests
  - Protects against regressions that reintroduce bug
    • It happened once, so it might again

Rules of Testing

- First rule of testing: **Do it early and do it often**
  Best to catch bugs soon, before they have a chance to hide.
  Automate the process if you can
  Regression testing will save time.
- Second rule of testing: **Be systematic**
  If you randomly thrash, bugs will hide in the corner until you’re gone
  Writing tests is a good way to understand the spec
  Think about revealing domains and boundary cases
  If the spec is confusing ➔ write more tests
  Spec can be buggy too
  Incorrect, incomplete, ambiguous, and missing corner cases
  When you find a bug ➔ fix it first and then write a test for it

Testing summary

- Testing matters
  - You need to convince others that module works
- Catch problems earlier
  - Bugs become obscure beyond the unit they occur in
- Don’t confuse volume with quality of test data
  - Can lose relevant cases in mass of irrelevant ones
  - Look for revealing subdomains (“characteristic tests”)
- Choose test data to cover
  - Specification (black box testing)
  - Code (glass box testing)
- Testing can’t generally prove absence of bugs
  - But it can increase quality and confidence

Debugging

Ways to get your code right

- Validation
  - Purpose is to uncover problems and increase confidence
  - Combination of reasoning and test
- Debugging
  - Finding out why a program is not functioning as intended
- Defensive programming
  - Programming with validation and debugging in mind
- Testing ≠ debugging
  - test: reveals existence of problem
  - debug: pinpoint location + cause of problem

A bug – September 9, 1947

US Navy Admiral Grace Murray Hopper, working on Mark I at Harvard
A Bug’s Life

• Defect – mistake committed by a human
• Error – incorrect computation
• Failure – visible error: program violates its specification
• Debugging starts when a failure is observed
  – Unit testing
  – Integration testing
  – In the field

Defense in depth

1. Make errors impossible
   – Java makes memory overwrite bugs impossible
2. Don’t introduce defects
   – Correctness: get things right the first time
3. Make errors immediately visible
   – Local visibility of errors: best to fail immediately
   – Example: checkRep() routine to check representation invariants
4. Last resort is debugging
   – Needed when effect of bug is distant from cause
   – Design experiments to gain information about bug
     • Fairly easy in a program with good modularity, representation hiding, specs, unit tests etc.
     • Much harder and more painstaking with a poor design, e.g., with rampant rep exposure

First defense: Impossible by design

• In the language
  – Java makes memory overwrite bugs impossible
• In the protocols/libraries/modules
  – TCP/IP will guarantee that data is not reordered
  – Right usage will guarantee that there will be no overflow
• In self-imposed conventions
  – Hierarchical locking makes deadlock bugs impossible
  – Banning the use of recursion will make infinite recursion/insufficient stack bugs go away
  – Immutable data structures will guarantee behavioral equality
  – Caution: You must maintain the discipline

Second defense: correctness

• Get things right the first time
  – Don’t code before you think! Think before you code.
  – If you’re making lots of easy-to-find bugs, you’re also making hard-to-find bugs – don’t use compiler as crutch
• Especially true, when debugging is going to be hard
  – Concurrency
  – Difficult test and instrument environments
  – Program must meet timing deadlines
• Simplicity is key
  – Modularity
  – Divide program into chunks that are easy to understand
  – Use abstract data types with well-defined interfaces
  – Use defensive programming; avoid rep exposure
  – Specification
    • Write specs for all modules, so that an explicit, well-defined contract exists between each module 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
  – Assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation
  – Unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (unless it’s in the test driver)
  – Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there’s a mistake 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

Benefits of immediate visibility

• Key difficulty of debugging is to find the code fragment responsible for an observed problem
  – A method may return an erroneous result, but be itself error free, if there is prior corruption of representation
• The earlier a problem is observed, the easier it is to fix
  – For example, frequently checking the rep invariant helps the above problem
• General approach: fail-fast
  – Check invariants, don’t just assume them
  – Don’t try to recover from bugs – this just obscures them
How to debug a compiler

- Multiple passes
  - Each operate on a complex IR
  - Lot of information passing
  - Very complex Rep Invariant
  - Code generation at the end
- Bug types:
  - Compiler crashes
  - Generated program is buggy

Don't hide bugs

```c
// k is guaranteed to be present in a
int i = 0;
while (true) {
    if (a[i] == k) break;
    i++;
}

// k is guaranteed to be present in a
int i = 0;
while (i < a.length) {
    if (a[i] == k) break;
    i++;
}
```

- This code fragment searches an array `a` for a value `k`.
  - Value is guaranteed to be in the array.
  - If that guarantee is broken (by a bug), the code throws an exception and dies.
- Temptation: make code more “robust” by not failing

Don't hide bugs

```c
// k is guaranteed to be present in a
int i = 0;
while (i < a.length) {
    if (a[i] == k) break;
    i++;
}
assert (i < a.length) : "key not found";
```

- Now at least the loop will always terminate
  - But no longer guaranteed that `a[i] == k`
  - If rest of code relies on this, then problems arise later
  - All we’ve done is obscure the link between the bug’s origin and the eventual erroneous behavior it causes.

Inserting Checks

- Insert checks galore with an intelligent checking strategy
  - Precondition checks
  - Consistency checks
  - Bug-specific checks
- Goal: stop the program as close to bug as possible
  Use debugger to see where you are, explore program a bit

Checking For Preconditions

```c
// k is guaranteed to be present in a
int i = 0;
while (i < a.length) {
    if (a[i] == k) break;
    i++;
}
assert (i < a.length) : "key not found";
```

Precondition violated? Get an assertion!
Downside of Assertions

```java
static int sum(Integer a[], List<Integer> index) {
    int s = 0;
    for (e:index) {
        assert(e < a.length, "Precondition violated");
        s = s + a[e];
    }
    return s;
}
```

Assertion not checked until we use the data
Fault occurs when bad index inserted into list
May be a long distance between fault activation and error detection

checkRep: Data Structure Consistency Checks

```java
static void checkRep(Integer a[], List<Integer> index) {
    for (e:index) {
        assert(e < a.length, "Inconsistent Data Structure");
    }
}
```

- Perform check after all updates to minimize distance between bug occurrence and bug detection
- Can also write a single procedure to check ALL data structures, then scatter calls to this procedure throughout code

Bug-Specific Checks

```java
static void check(Integer a[], List<Integer> index) {
    for (e:index) {
        assert(e != 1234, "Inconsistent Data Structure");
    }
}
```

Bug shows up as 1234 in list
Check for that specific condition

Checks In Production Code

- Should you include assertions and checks in production code?
  - Yes: stop program if check fails – don’t want to take chance program will do something wrong
  - No: may need program to keep going, maybe bug does not have such bad consequences
  - Correct answer depends on context!
- Ariane 5 – program halted because of overflow in unused value, exception thrown but not handled until top level, rocket crashes…