CS 520
Theory and Practice of Software Engineering
Fall 2020
Debugging
October 13, 2020

Tuesday (October 15)
• Third in-class exercise
• On debugging
• Form 3-, 4-, or 5-person teams
  – Use Moodle to self-select a team; open from today until Thursday at 9 pm EDT
  – Aler closing, the remaining students will be randomly assigned to groups
• Due: Tuesday October 20, 9 am EDT

Ways to get your code right
• Validation (e.g., code reviews, testing, model checking)
  – Purpose is to uncover problems and increase confidence
• Debugging
  – Finding out why a program is not functioning as intended
• Defensive programming
  – Programming with validation and debugging in mind
• Validation ≠ debugging
  – Validation: Reveals existence of problem
  – Debugging: Pinpoints location + cause of problem

A bug – September 9, 1947
US Navy Admiral Grace Murray Hopper, working on Mark I at Harvard
Bug Reporting:
Bug tracking systems

- Commonly provide support for:
  - Logging in and out
  - Writing a new bug report
  - Searching through existing bug reports
  - Reading existing bug reports and updating their status

- Examples: Bugzilla, mantis, trac

Example: Bugzilla UI

Example: Bugzilla bug report
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, e.g.,
  - Manual code review
  - Testing: unit, integration, system
  - Model checking
  - In the field

Defense in depth

1. **Make errors impossible**
   - e.g., 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
     - e.g., assertions to check representation invariants

Defense in depth (cont.)

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 design, e.g., modularity, representation hiding, specs, unit tests, etc.
     - Much harder and more painstaking with a poor design, e.g., no decomposition, representation exposure, no unit tests, etc.

First defense: Impossible by design

- **In the language**
  - e.g., Java makes memory overwrite bugs impossible
- **In the protocols/libraries/modules**
  - e.g., BigInteger will guarantee that there will be no overflow
- **In self-imposed conventions**
  - e.g., unmodifiable collections will guarantee behavioral equality
  - Cautions: 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, e.g.,**
  - Concurrency, non-determinism
  - Difficult test and instrument environments
  - Program must meet timing deadlines

Second defense: correctness (cont.)

- **Simplicity is key, e.g.,**
  - 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 users

Example:
Common compiler architecture

- **Multiple passes**
  - Each operate on a complex IR (Internal Representation)
  - Lot of information passing
  - Very complex Rep(resentation) Invariant
  - Code generation at the end

Third defense: immediate visibility

- If we can’t prevent bugs, we can try to localize them to a small part of the program, e.g.,
  - Assertions
  - Unit testing
  - Regression testing
- 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
  - e.g., 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
  - e.g., 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

Example: Immediate visibility

- Bug types:
  - Compiler crashes
  - Generated program is buggy

Don’t hide bugs (v1)

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

- If that guarantee is broken (by a bug), the code will throw an exception and die.
- Temptation: make code more “robust” by not failing

Don’t hide bugs (v2)

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

- Now at least the loop will always terminate
  - But no longer guarantees 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.
Don't hide bugs (v3)

// 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";

• Assertions let us document and check invariants
• Abort program as soon as problem is detected

Inserting Checks

• Insert checks galore with an intelligent checking strategy, e.g.,
  – 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 Preconditions

// 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

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
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

Debugger: Eclipse IDE

https://www.baeldung.com/eclipse-debugging

Checks In Production Code

- Should you include assertions and checks in production code?
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!

Example: Ariane 5 rocket (1996)
Program halted because of overflow in unused value, exception thrown but not handled until top level, rocket crashes…

Bug Fixing

- Manual
- Automated program repair (APR) techniques commonly consist of 3 main components:
  - Fault Localization
  - Patch (or Repair) Generation
  - Patch Validation

Homework 2

- Re-design, re-implement, and test the Row game app
- MVC architecture pattern, Observer design pattern, Strategy design pattern, Code review proposed fixes
- Due: Friday October 23, 2020, 9 PM EDT

https://people.cs.umass.edu/~hconboy/class/2020Fall/CS520/hw2.pdf