CS 520
Theory and Practice of Software Engineering
Fall 2021

Debugging

October 21, 2021
Recap: Some common software development processes

• Traditional development process
• Test-driven development process
• Collaborative development process
Recap:
Traditional development process

1. Requirements
2. Architecture
3. Design
4. Implementation
5. Verification & validation (e.g., testing)
Recap:
Test-driven development process

1. Add new test case(s) for a new feature and run all test cases
   – The new test case(s) should fail
2. Implement that new feature and run all test cases
   – All tests should pass
3. Refactor the implementation as needed to improve its quality and run all test cases
   – All tests should still pass
4. Repeat

https://en.wikipedia.org/wiki/Test-driven_development
Recap:
Collaborative development process

• A very popular flavor of Agile to rapidly iterate in Sprints
  – Each Sprint develops then releases the product

• Three pillars:
  – Transparency
  – Inspection
  – Adaptation

• Used by large tech companies such as Facebook, Google, Microsoft

https://www.scrum.org
Tuesday (October 26)

• Third in-class exercise on debugging
• Form 3-, 4-, or 5-person teams
  – Use Moodle to self-select a team; open until Tuesday 26, 12:00 pm (noon)

• Due: Tuesday November 2, 11:59 pm (a little before midnight)
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’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
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

Welcome to Bugzilla. To see what's new in this version of Bugzilla, see the release notes! You may also want to read the Bugzilla User's Guide to find out more about Bugzilla and how to use it.

Most common actions:
Search existing bug reports
Enter a new bug report
Summary reports and charts

Change password or user preferences
Log out hconboy@cs.umass.edu

Add to Sidebar (requires a Mozilla browser like Mozilla Firefox)
Install the Quick Search plugin (requires Firefox 2 or Internet Explorer 7)

Enter a bug # or some search terms:
Find [Help]
Bug Reporting: Common bug report format

- Brief description
- Reporter and reporting date
- Environment: Operating system, application’s version
- Severity and priority (e.g., a small integer range, enum type for LOW/MEDIUM/HIGH)
- Steps to reproduce along with expected and actual result descriptions (e.g., text, screenshot)
- Comments
- Status (e.g., new, team responsible, fixed)
Example: Bugzilla bug report

Average bug report consists of following sections:

- **Summary**: 
  - A brief description of the issue.
- **Product**, **Component**, **Version**, **Status**, **Priority**, **Severity**
- **Description**: 
  - Detailed explanation of the problem.
- **Resolution**: 
  - Status after resolution.
- **Attachments**: 
  - Proposed patches, test cases, etc.

For example:

**Summary**: Names can't handle special characters

**Product**: PROPEL
**Component**: UI
**Version**: unspecified
**Status**: NEW
**Priority**: Normal
**Severity**: normal
**Resolution**: 

**Description**: When creating subprojects whose names contain a period, PROPEL seems to ignore everything that comes afterwards. So, when trying to create two subprojects A.1 and A.2 for property A, only one appears. The second one is not created and there's no error, it just mysteriously disappears. Alpha-numerics only seem to work just fine.

Reportedly, same error occurs when using a colon (:) not tested.
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 rep(resentation) 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 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
  – 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,** 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)

// 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)

// 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.,
  – Pre- and post-condition 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

// k is guaranteed to be present in a

```java
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
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?
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...
Other common debugging techniques

• Debugging (or logging) statements

• Debugger

• Delta debugging
Debugging (or logging) statements

• Add println debugging statements

• Use a Logger
  – e.g., log4j available from here: https://logging.apache.org/log4j/2.x/
Debugger

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

Delta Debugging is a methodology to automate the debugging of programs using a scientific approach of hypothesis-trial-result loop. This methodology was first developed by Andreas Zeller of the Saarland University in 1999.[1]

In practice, the Delta Debugging algorithm builds on unit testing to isolate failure causes automatically - by systematically narrowing down failure-inducing circumstances until a minimal set remains. For example, if you can supply a test case that will produce the bug you are looking for, then you can feed that to the Delta Debugging algorithm, which will then simply try to trim useless functions and lines of code that are not needed to reproduce the bug, until a 1-minimal program is found.

Delta Debugging has been applied to isolate failure-inducing program input (e.g. an HTML page that makes a Web browser fail), failure-inducing user interaction (e.g. the keystrokes that make a program crash), or failure-inducing changes to the program code (e.g. after a failing regression test).

Later, some software development tools have been inspired by Delta Debugging, such as the bisect commands of revision control systems (e.g., git-bisect, svn-bisect, hg-bisect, etc.), which, instead of working on the program's code, apply the delta debugging methodology on the code history by comparing various versions until the faulty change is found.

Delta debugging

Delta Debugging is a methodology to automate the debugging of programs using a scientific approach of hypothesis-trial-result loop. This methodology was first developed by Andreas Zeller of the Saarland University in 1999.[1]

In practice, the Delta Debugging algorithm builds on unit testing to isolate failure causes automatically - by systematically narrowing down failure-inducing circumstances until a minimal set remains. For example, if you can supply a test case that will produce the bug you are looking for, then you can feed that to the Delta Debugging algorithm, which will then simply try to trim useless functions and lines of code that are not needed to reproduce the bug, until a 1-minimal program is found.

Delta Debugging has been applied to isolate failure-inducing program input (e.g. an HTML page that makes a Web browser fail), failure-inducing user interaction (e.g. the keystrokes that make a program crash), or failure-inducing changes to the program code (e.g. after a failing regression test).

Later, some software development tools have been inspired by Delta Debugging, such as the bisect commands of revision control systems (e.g., git-bisect, svn-bisect, hg-bisect, etc.), which, instead of working on the program's code, apply the delta debugging methodology on the code history by comparing various versions until the faulty change is found.

Tuesday (October 26)

- Third in-class exercise on delta debugging
- Form 3-, 4-, or 5-person teams
  - Use Moodle to self-select a team; open until Tuesday 26, 12:00 pm (noon)

- Due: Tuesday November 2, 11:59 pm (a little before midnight)