

Beta

- Beta is due next Tuesday
- Beta includes presentations
 - 15 minutes per group
 - at least 2 students per group
 - practice practice practice

1

Team Assessment

- Due **Thursday, Nov 3, by midnight**

https://docs.google.com/forms/d/e/1FAIpQLSebRgKpUhnliD9LWkv3ifgTGqLDou34pkWixnbkVv3kh8girw/viewform?usp=sf_link

- will take less than 5 minutes

2

First test

- Mean and median: 79
- Standard deviation: 9.8
- Max: 99
- Solution posted on moodle

3

What happened to multiple choice?

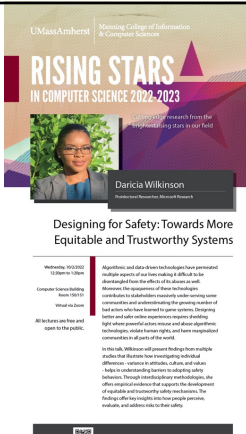
- Questions 2 and 3 were multiple choice.
- All questions were multiple choice, with just a single choice. Single is multiple!
- Yuriy had the same question choice!
- Yuriy had the same question and didn't let him sleep and he screwed up and promised you multiple choice, but then thought he promised short-answer, and wrote short-answer questions on the exam...

4

- Wednesday, 12:20-1:20
- Pizza at noon
- CS150/151

<http://umass-amherst.com/en/992259150/2022/11/01/150-151-1101-2022>

- No class Thursday



RISING STARS
IN COMPUTER SCIENCE 2022-2023

Darcia Wilkinson
Professor Emerita, Research Research

Designing for Safety: Towards More Equitable and Trustworthy Systems

Abstracts and data-driven technologies have permeated multiple aspects of our lives, making it difficult to be disconnected from the effects of the devices we use. However, the development of these technologies contributes to inequalities, especially under varying circumstances and underestimating the varying needs of all those who have benefited from systems. Designing for safety and user-centric experience requires creating high-impact powerful value chains and those algorithms, technologies, and data-driven systems will have significant consequences in all parts of the world.

In this talk, Darcia will present findings from multiple studies that illustrate how investigating individual differences, context in artificial culture, and values helps in understanding the barriers to adopting safety behaviors. Through interdisciplinary methods, she offers implications for the design of the development of safe and trustworthy utility technologies. The findings offer insights on how people perceive, evaluate, and address risks to their safety.

ccs.umass.edu/rising-stars

5

Debugging

6

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

7

A bug – September 9, 1947

US Navy Admiral Grace Murray Hopper, working on Mark I at Harvard

9/9

0800 Action started
 1000 stopped - action ✓
 1300 1500 MP - MC 1.35000000 9.037 896 995 result
 2000 2800 2.13000000 9.415 925049(1.0)
 2000 2800 2.13000000
 Relays 6-2 on 032 failed special speed test
 in relay
 (Relays changed in relay)
 1100 Started Cosine Tape (Sine check)
 1525 Started Multi Adder Test.
 1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.
 1600 1700 automatic started.
 1700 closed down.

Relay #70
 2195
 2197

8

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

9

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

10

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
 - BigInteger 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

11

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

12

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

13

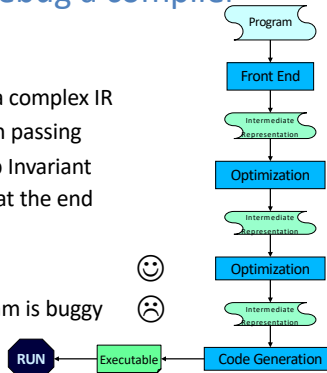
Benefits of immediate visibility

- 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

14

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 ☹️



15

Don't hide bugs

```

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

- This code fragment searches an array *a* for a value *k*.
 - Value is guarantee 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

16

Don't hide bugs

```

// 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 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.

17

Don't hide bugs

```

// 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

18

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

19

Checking For 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!

20

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

21

checkRep: Data Structure Consistency Checks

```
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

22

Bug-Specific Checks

```
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

23

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...

24