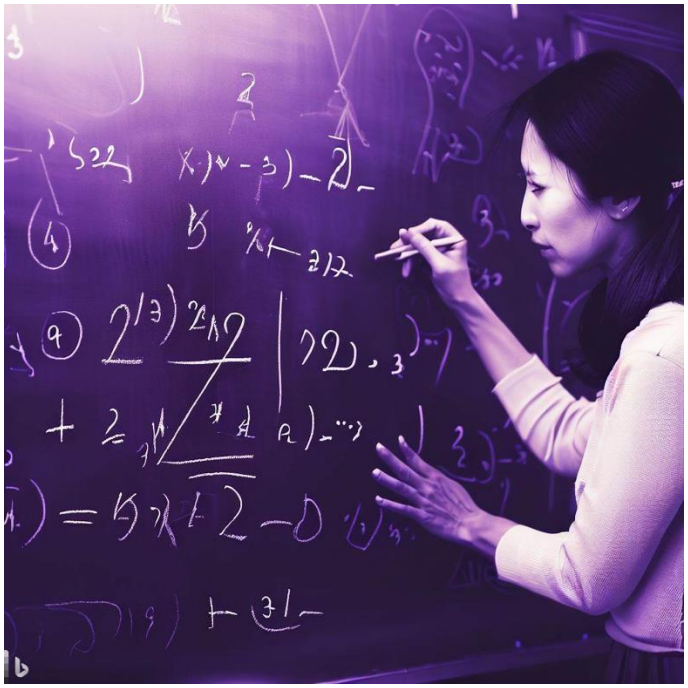


CSE 331

Software Development Process

Jaela Field
(Kevin Zatloukal & James Wilcox & Matt Wang)

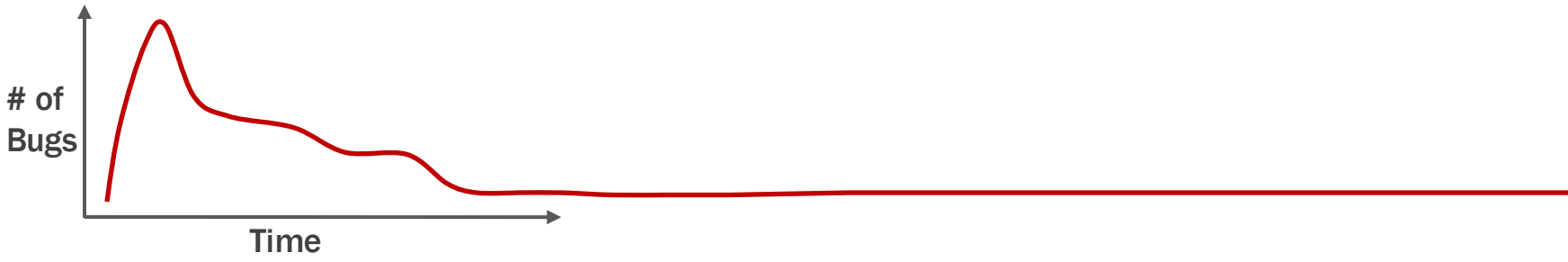
HW3 Summary: Bugs & Time per bug

- **More bugs in more complex applications!**
  - **Bugs often require searching through more than one function**
  - **(client app bugs) + (server app bugs) < (client-server app bugs)**

- **Debugging time grows fast with app complexity!**
  - **Historically:**

    each extra function you must search through adds ~10-15 minutes

    35-40% of bugs taking > 1 hour



HW3 Summary: Search Space of Bugs

- **Shrinking the search space helps a lot**
  - **unit tests!**
  - **defensive programming!**

    double check that preconditions are satisfied

    run-time type checking of request/responses

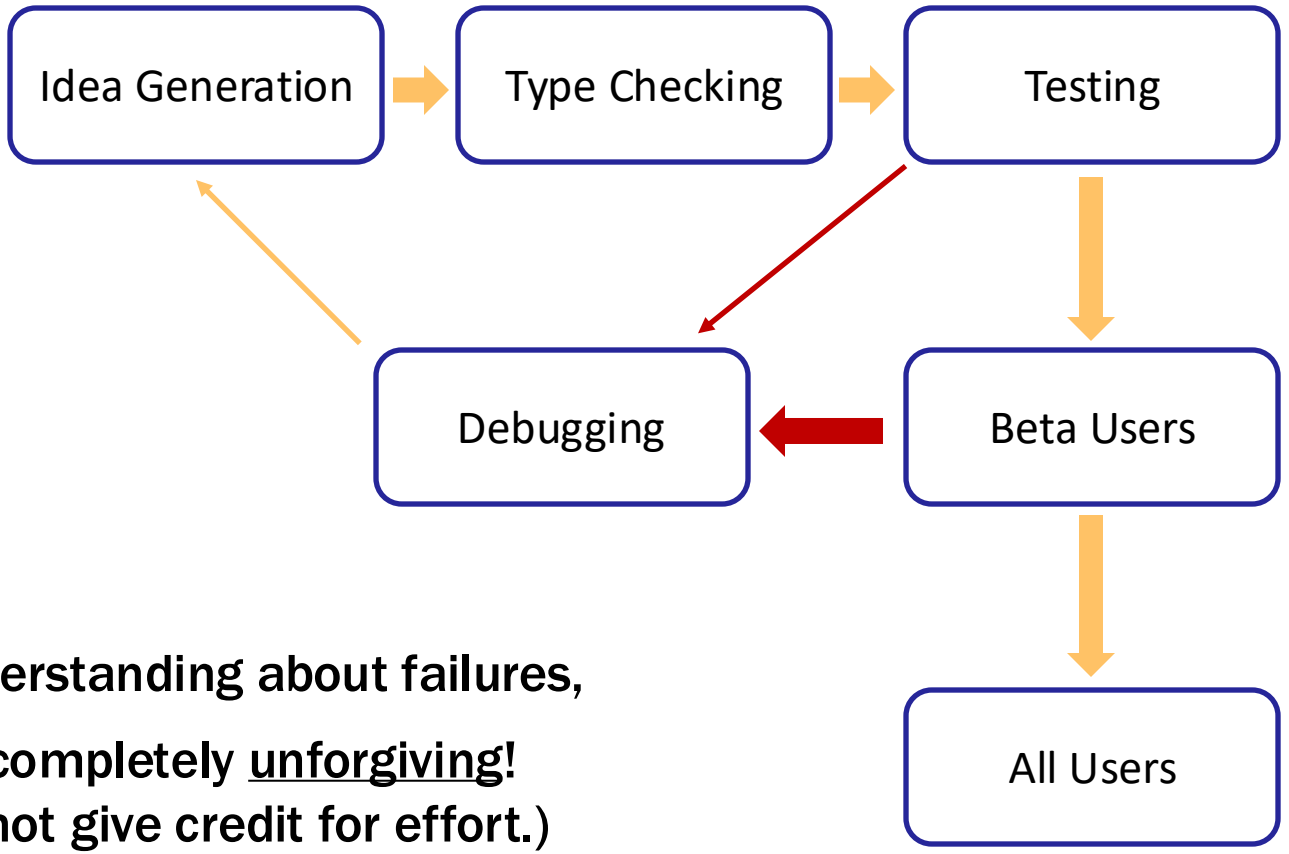
Summary of HW1–3

- HW1: type checking is important
  - found almost 50% of the bugs
- HW2: mutation is dangerous
  - cause of the most horrible kinds of debugging
- HW3: unit testing is important
  - debugging a small space for ~2/3rd of bugs
- Debugging will still happen…
  - need to get better at quickly narrowing in on the bug

Software Development Process

Software Development Process (right now)

Given: a problem description (in English)



Beta users are understanding about failures,

Regular users are completely <u>unforgiving</u>!
(Regular users do not give credit for effort.)

How Much Debugging?

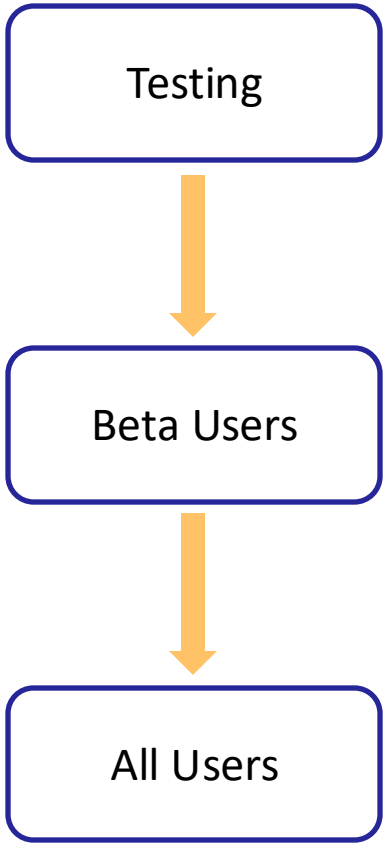
- Bugs typed in… 1 per 20 lines
  - the norm for pretty much everyone
- Bugs after type checking… 1 per 40 lines
  - assume 50% caught by type checker (saw 41% in HW1)
- Bugs after unit testing… 1 per 133 lines
  - assume 70% caught by unit testing

    optimistic: studies find about <70% are caught by unit testing
  - remaining bugs are sent to beta testers

How Much Debugging?

- Bugs after testing… 1 per 133 lines
  - assume 70% caught by testing
  - studies find about 65% are caught by testing
- Are rest are caught by beta users?
  - not enough of them
  - millions of users will find all bugs
- Bugs after beta users… 1 per 2000 lines
  - number from Microsoft
  - anything created by humans has mistakes

    only a small number of users give 0 stars



How Many Bugs Sent to Beta Users?

- **Every 2000 lines of code**

100 bugs typed in	1 per 20 lines
– 50 bugs caught by type checker	(50%)
= 50 bugs	
– 35 bugs caught by unit testing	(70%)
= 15 bugs	

- **Need to debug 14 bugs from beta users**
  - **will still send 1 bug to regular users**

What Kind of Bugs Sent to Beta Users?

- Comes back without steps to reproduce the failure
  - only comes back with a description of the failure

    maybe a vague (possibly incorrect) description of steps

- Only sent to beta users if it…
  - type checks
  - gets past unit tests

- Most such bugs often at the seams between functions
  - multiple functions need to be debugged
  - will take a long time to track down (many hours)

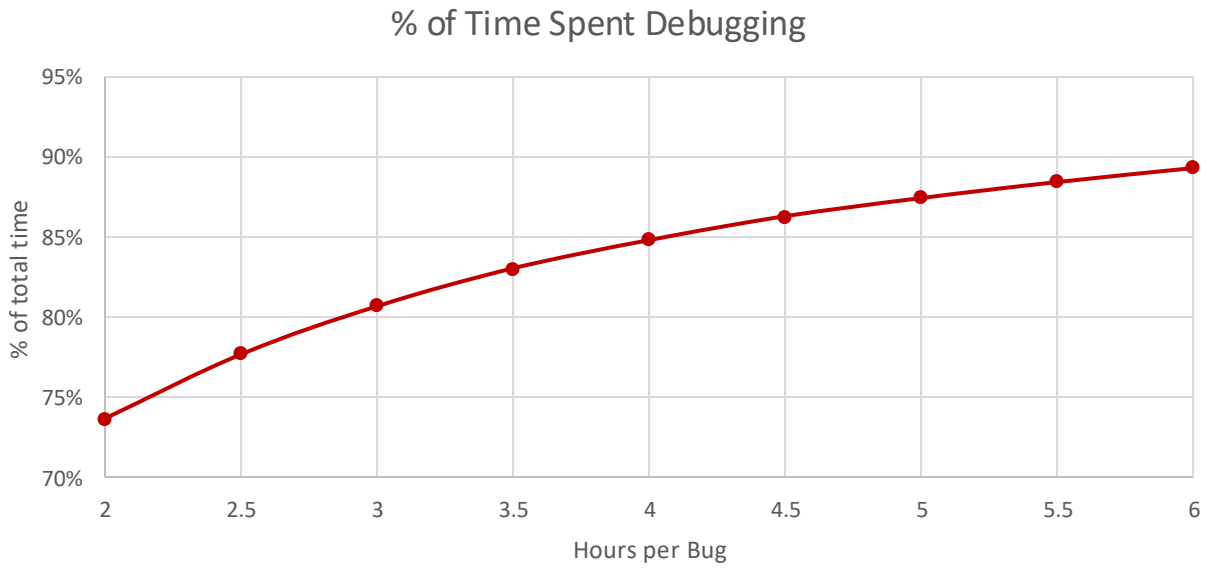
    we saw an extra 10-15 minutes for every additional function in HW3

    HW3 had 700 lines… industry programs will be 100,000 minimum

Productivity Estimate

- **2000 lines of code**
  - **assume a familiar setting (know how to solve problems)**
  - **let "h" be the number of hours to debug one such bug**

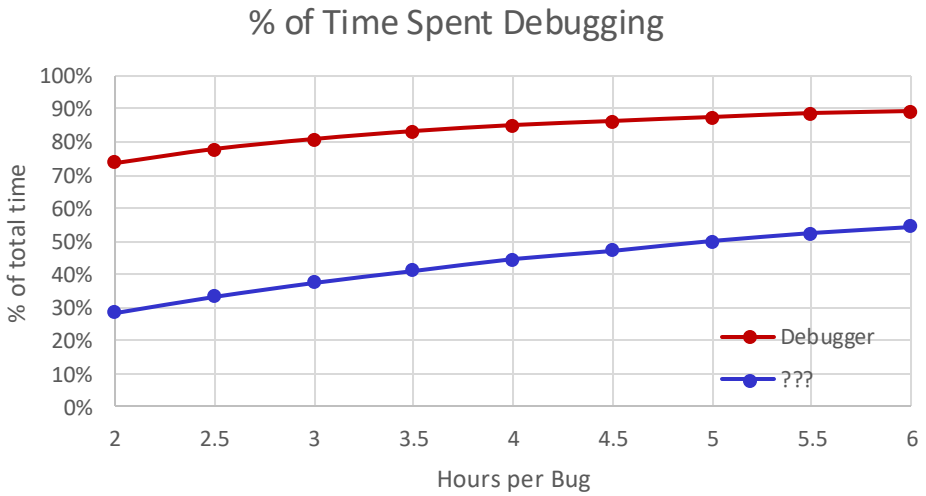
5 hours	typing & fixing type errors
5 hours	testing & fixing *unit* test failures
14 * h hours	debugging & fixing bugs



What Else Can We Do?

- **2000 lines of code**
  - **assume a familiar setting (know how to solve problems)**
  - **let "h" be the number of hours to debug one such bug**

5 hours	typing & fixing type errors
5 hours	?? **removes 11 bugs** ??
5 hours	testing & fixing *unit* test failures
3h hours	debugging & fixing bugs



even at h=5, debugging <u>not</u> the majority of time

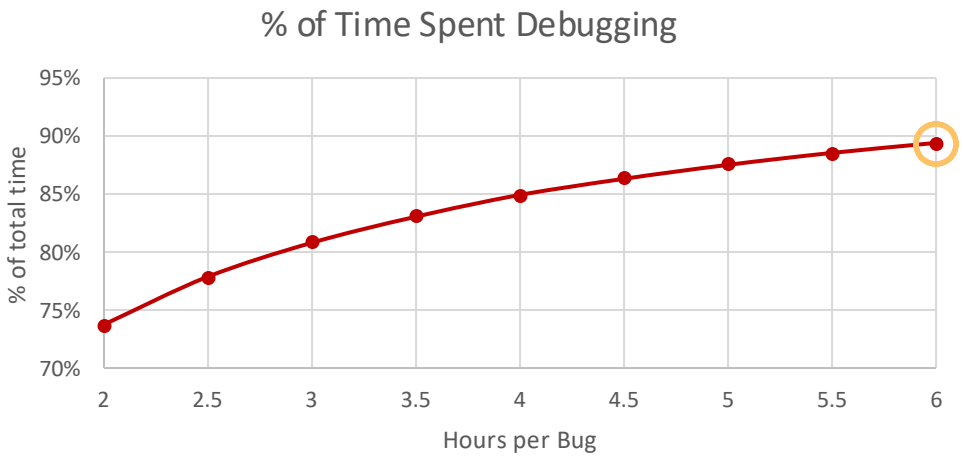
bottom programmer is 2-3 times more productive

How Much Room For Improvement?

- **Suppose we could…**
  - **remove all 14 bugs by the end of unit testing**
  - **in the same amount of time**

    plausible since fixing unit test failures involves debugging

5 hours	typing & fixing type errors
3 hours	?? **removes 14 bugs** ??
2 hours	testing & fixing *unit* test failures



would cut 90% of time spent

would be 10x more productive

"10x developer" possible in a setting where debugging is hard but can be avoided with extra effort

“Engineers are paid to think and understand.”

— Class slogan #1

Standard Techniques for Correctness

Standard practice (60+ years) uses three techniques:

- **Tools**: type checker, libraries, etc.
- **Testing**: try it on a well-chosen set of examples
- **Reasoning**: think through your code carefully
  - convince yourself it works correctly on *all inputs*
  - have another person do the same ("code review")

Comparing These Techniques

- Differ along some key dimensions
  - does it consider all allowed inputs
  - does it make sure the answer is fully correct ("=")

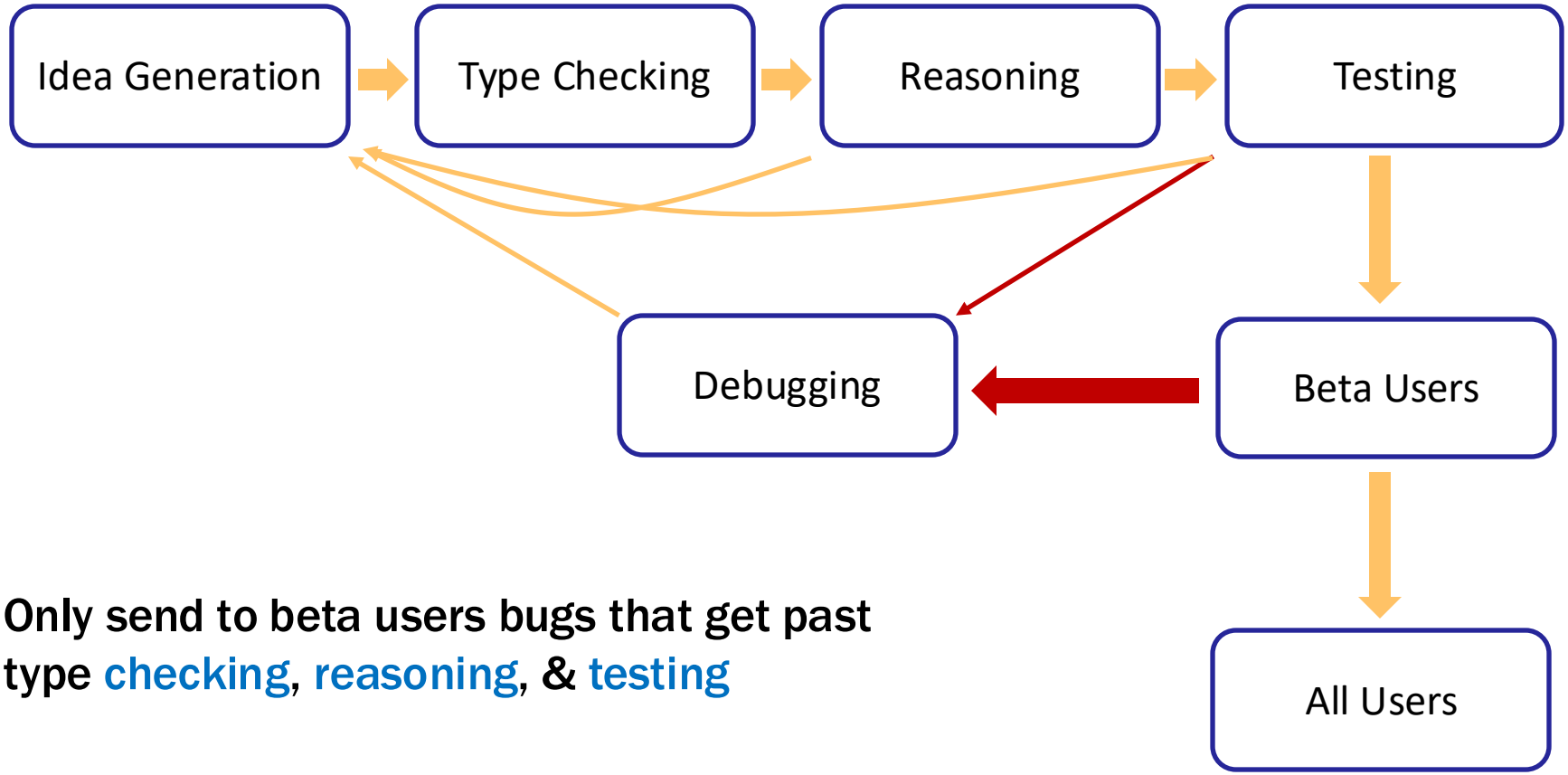
Technique	All Inputs	Fully Correct
Type Checker	✓ Yes	✗ No
Testing	✗ No	✓ Yes
Reasoning	✓ Yes	✓ Yes

- Combination removes >97% of bugs
  - each tends to find different kinds of errors
  - e.g., type checker is good at typos & reasoning is not

    humans often skip right over typos when reading

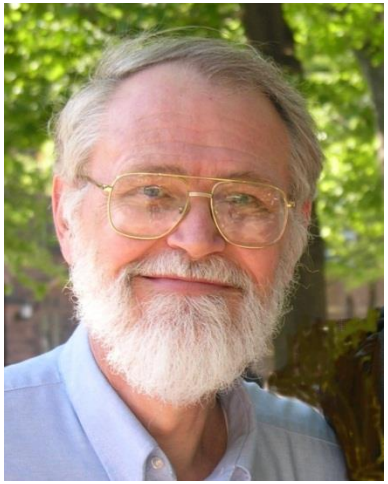
Software Development Process (Improved)

Given: a problem description (in English)



Only send to beta users bugs that get past type checking, reasoning, & testing

"Debugging is twice as hard as writing the code in the first place."



Brian Kernighan

Reasoning is Expected

- **In industry**: you will be expected to think through your code
  - standard practice is to do this *twice* ("code review")
    you think through your code then ask someone else to also
- Professionals spend most of their coding time reasoning
  - reasoning is the core skill of programming
- Interviews are tests of reasoning
  - take the computer away so you only have reasoning
  - typical coding problem has lots of cases that are easy to miss if you don't think through carefully
  - (not about knowing "the answer" to the question
    interviewers will throw out interviews that went too well!)

“Automating” Reasoning & LLMs

- Reasoning & debugging are provably impossible for a computer to solve in all cases

- Current LLM error rates are much higher than humans
  - requires an (expert) human to do a lot of debugging

    starts with reading and **understanding** all the generated code…

    probably easier to rewrite it yourself
  - studies (so far) show little productivity improvement

    if it reads your mind, it saves you typing, but that's not the limiting factor

    if it doesn't read your mind, you must still spend time understanding it

- LLMs are especially bad at reasoning
  - e.g., bad at learning formal properties
  - e.g., bad at catching rare cases

Actually Correct Automated Reasoning

- There are non-LLM (and crucially, deterministic) approaches to automated reasoning
  - "formal methods" & "formal verification"
  - SAT & SMT-based solvers (incl. model checking)
  - program synthesis
  - automated theorem proving & proof assistants
- Very promising area of research, but…
  - many require graduate-level study to use
  - many current open problems (modularity, scalability)
  - thus, not common in most software engineering fields (yet!)