CSE 331

Course Review

Kevin Zatloukal

Administrivia: Course Evals

- Would like to get above 50%
  - more statistical sample
- Looking for feedback on first attempt at this course
  - know about a few stumbles

    HW2: need to explain pattern matching better

    HW8: need to explain POST requests better
- Will make it your time to write an eval

Administrivia: Final Exam

- Both exams are on Tuesday
  - B section at 2:30 in Kane 110
  - A section at 4:30 in Kane 110

- Will be like the midterm but longer (110 minutes)
  - emphasis on the same core reasoning skills
  - ideally more comprehensive

Administrivia: Final Exam

- 6–7 problems
- 4 problems like the midterm
  1. Correctness of a complex loop
  2. Writing a loop correctly given the invariant
  3. Writing code correctly given no invariant
  4. Testing

  core material
- 2–3 more on
  - **things we skipped in midterm:** ADT reasoning, induction
  - **things covered more recently:** subtyping, equality, design patterns
  - **more like the midterm (!) or anything else**

Debugging is No Fun

- Code with mutable state often involves **debugging**
  - saw this in HW7-9
- Gets even worse as the program gets larger
  - lots more parts that can fail
  - lots more code to search through
- Only time spent debugging makes you hate debugging
  - watching a lecture won't do it

Engineers Are Paid to Think

- **For complex code, reasoning is not optional**
  - **going to do the reasoning eventually**

    complex code is not correct by accident
  - **choice is between**
    1. reasoning
    2. debugging and then reasoning

- **Easier to get it right the first time!**

Course Goals

To teach you to the skills necessary to write programs at the level of a professional software engineer

Specifically, we will teach the skills to write code that is

- correct
- easy to understand
- easy to change
- modular

We will set an **extremely high bar** for correctness

Standard Techniques for Correctness

Standard practice uses three techniques:

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

Each removes ~2/3rd bugs but of different kinds

Combination removes >97% of bugs

Tools

- Saw one case with no tool support: client-server
- POST is “passing arguments” to the server
  - but there is no type information!

    we must check all the types ourselves at runtime (or debug!)
- Type checking is very useful!

Reasoning

- Reasoning is the key skill of a programmer

  "the Olympic athletes of forward reasoning" — J. Wilcox

  – either reason now or after debugging
  – why we spent 12+ lectures on it

- Saw how mutation makes everything harder
  – most hard bugs in HW2 were mutation
  – most hard bugs in HW7 were mutation

    some function was mutating one of its arguments when it shouldn't

  – most hard bugs in HW8-9 were mutation

    components work by mutating `this.state`

- Pro Tip: limit mutation to make reasoning easier

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	no mutation	“	libraries	calculation induction
2	local variable mutation	“	“	Floyd logic
3	array / object mutation	“	“	rep invariants
4	state in two programs	“	“	more invariants

Topics

Primary Topics

- Basics of correctness (tools, testing, & reasoning)
- Writing correct programs without mutation
- Writing correct programs with mutation
- Client & Server applications

Other Topics

- Abstraction
- Debugging
- Design Patterns

Tips

1. Know the Correctness Level

- Level –1 is especially important
  - no need for reasoning — just look at it!
  - happens frequently with UI

    look at it to see if it is right

    try it out to see if it transitions pages correctly

- Reasoning is only necessary at Level 1–4
  - spend your reasoning efforts only there

- At level 3–4, expect to debug
  - make sure all the helper functions are right before you start

2. Keep the Level Low

- Do not make the level higher for no reason
  - do not mutate if you don't need to
  - do not introduce state when not necessary

    ex: passing data through a field rather than as an argument

    why do this to yourself?

- Bad programmers make easy problems hard
  - good programmers don't do that
  - plenty of problems to solve that are already hard

3. Use the Type Checker

- **Type checking catches many errors for you**
  - **see previous example about POST requests**

- **Do not purposefully ignore its help**
  - **no type checking means lots more debugging**

    we all make lots of mistakes in these areas
  - **saw plenty of "`val: any`" in HW8**

    why do this to yourself?

4. Use Libraries When Available

- It is hard to get code working correctly
  - design, testing, reasoning, debugging
  - don’t do all that work when you don’t need to
- If someone already did the work, take advantage
  - reduces the work to making sure you use it right

5. Start With Data Design

- Figure out what data you need to make the UI work
  - code follows from the data
- Server stores the permanent data
  - decide what operations are needed by the client
  - make sure you get these right!
- UI stores data necessary to render
  - everything on the UI is somewhere in the data

6. Start with Simple, Concrete Data Types

- Reasoning is easier with concrete data types
  - start there, whenever possible
- Hard for us to predict:
  - what will be slow
  - what users will like (see, e.g., ChatGPT)
- Avoid unnecessary work
  - not everything is an ADT

    I'm talking to you, Java!
  - don't complicate things until you know you need to

    don't prematurely optimize

    don't prematurely abstract

7. Hide Complex Data Structures in ADTs

- Introduce ADT on first change to data structures
  - if you change it once, you'll probably do it again
- Give a simple spec to clients
  - probably the initial concrete state is the abstract state
  - allow the clients to think about it as they did before
- Make sure you got it right before moving on

  debugging is hard enough already
- Rep invariants are the key to <u>complex</u> data structures
  - see, e.g., AVL trees

8. Hide Complex Loops in Helper Functions

- Think of a simple, declarative explanation for clients
  - don't let complexity leak everywhere else
- Make sure you got it right before moving on

  debugging is hard enough already
- Loop invariants are the key to <u>complex</u> loops
  - see, e.g., dynamic programs in CSE 421
  - write it down fully and check it carefully in every branch

    won't get it right by accident
  - human readers don't need invariants for every loop

9. Be Systematic When Debugging

- After 20 minutes, stop trying things randomly
- Check the easy stuff first
  - if the server needs restarting, reasoning is a waste of time
- Debugging happens when your knowledge is wrong
  - don't think "it can't be there"…
- Think through all the places the bug could be
  - start eliminating them one at a time
  - think of an experiment that will reduce the search

10. Expect to Get Things Wrong

- First design is inevitably wrong in some ways
  - can only design perfectly when you've built *that* before
- Can't always guess on paper
  - need to try building it to see the key problems
- Design for changeability in the parts likely to change
  - put abstraction in place the first time you change it
  - (don't introduce abstraction unnecessarily)

Startups

Startups in 2021

	UW	Stanford
Funded Startups	70	465
Dedicated VC Funds	1	3

Engineers Started Many Companies

Some prominent examples…

Microsoft	Bill Gates & Paul Allen
Apple	Steve Jobs
Google	Sergey Brin & Larry Page
Meta	Mark Zuckerberg
NVIDIA	Jensen Huang
tsmc	Morris Chang

Very Little Downside

- Starting a company has almost no downside
  - expected to fail
  - lose other people's money, not yours
- "Founder of X" looks great on a resume
  - demonstrates grit, risk-taking, etc.
  - many other important skills
- Main loss to you is the time spent

Don’t Feel Weird Raising Funds

- Investors work for you
  - they join your team and are expected to help
  - the more they pay, the more you expect from them
- VCs can help you find customers, employees, etc.
- Old saying in finance

  When you owe the bank $100, that’s your problem.

  When you owe the bank $100m, that’s the bank’s problem.

Advice for Starting a Company

- Advice from YC:
  1. Build something people want
  2. Launch fast & iterate
  3. Write code & talk to users
  4. Find 10–100 people who love it
- Think of things you would like to use
  - or that improve the lives of others
- Can become for-profit or non-profit
  - hard to know up front what will work best