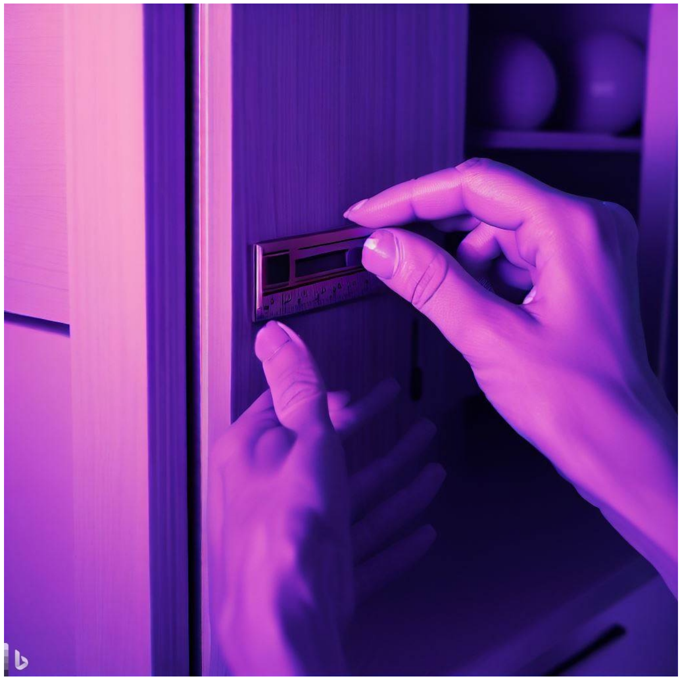


CSE 331

Correctness

Kevin Zatloukal

Scale of Modern Software

Analogy to physical objects

- 100 well-tested LOC = nice cabinet
- 2,500 LOC = room with furniture
- 2,500,000 LOC = 1000 rooms =

North Carolina class WW2 battleship

=

entire British Navy in WW2

Correctness Is Harder in Larger Programs

- Much harder to write large programs correctly
  - bugs in N-line program grow like $\Theta(N \log N)$ [Jones, '12]
  - time to write programs grows like $\Theta(N^{1.05})$ [Boehm, '81]
- Parts are more *interdependent*
  - correctness of any 100 lines depends on 1,000s of others
- Debugging becomes incredibly difficult
  - a mistake in one "ship" causes another to sink
- Small probability cases become high probability
  - even "impossible" cases happen

Correctness Is Harder in Larger Programs

- Must work harder to ensure each piece is correct
  - check every piece more times to find mistakes
- Learn proper technique on smaller programs
  - much easier to learn now, rather than later

Course Goals

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

Specifically, we will focus on writing code that is

- correct ← use this time to develop proper technique
- easy to understand
- easy to change
- modular

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

Quality is Harder in Large Programs

- Natural state of software is "spaghetti code"
  - all the parts are interdependent
  - cannot be disentangled
- Becomes impossible to **change**
  - any change to any part breaks some other part
- Use **modularity** to fight against interdependence
  - requires constant effort

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

Which Ones and How Much

- The first question to ask yourself:

  How much of this is needed for my program?

- Correctness is easier for some programs vs others

- Personally, I break this into 5 cases…
  - "levels" of difficulty
  - (I made this terminology up)

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	?			
1	?			
2	?			
3	?			

Level -1

- Small number of inputs / configurations
- Just check them all!
  - this is the right answer
- This category does not require a programmer
  - anyone can check the answer
  - programming is hard, so skip it when you can

Level -1

- **Coding is the wrong tool for this job**

- **Examples with one input / configuration:**

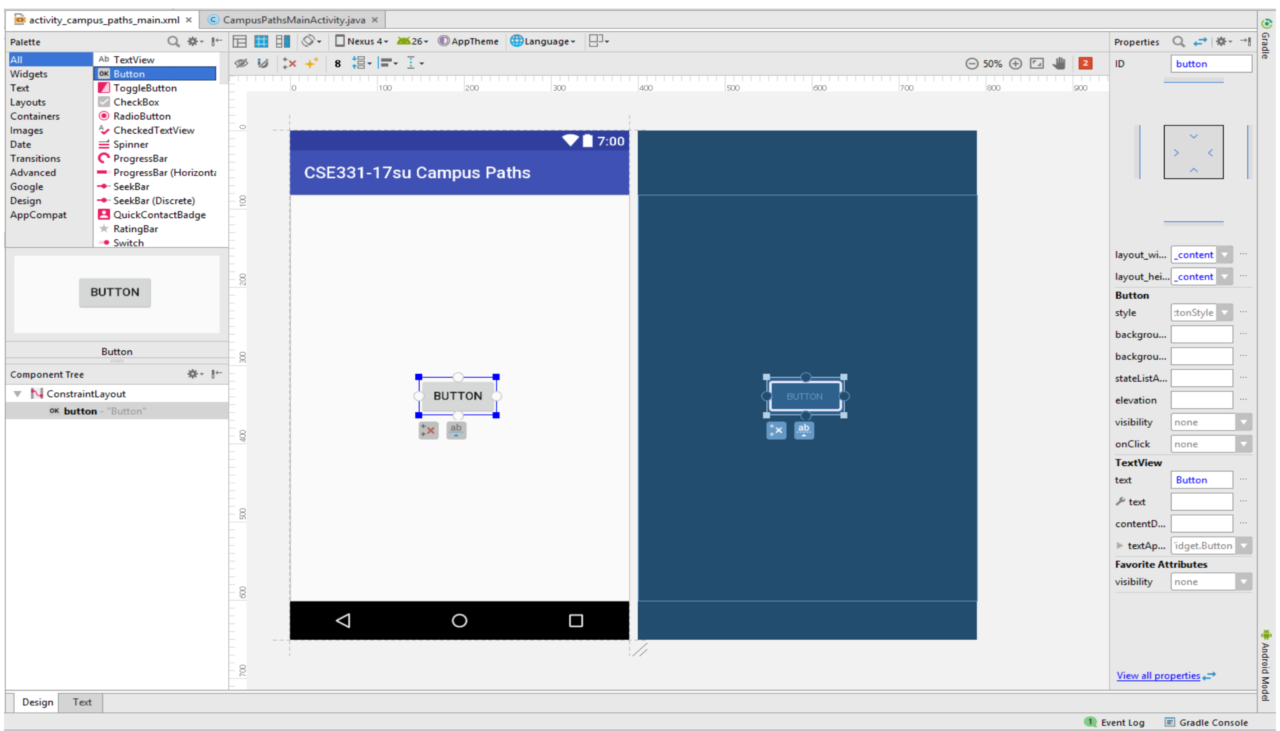
  – **using code to draw a *specific* picture (use Illustrator)**
    c.f. drawing a picture in LaTeX

  – **using code to transform *specific* data (use Excel)**
    e.g., stack three columns of numbers into one column

Level -1

- Can happen as part of a larger application

- iPhone development lets you draw the UI:



Level -1

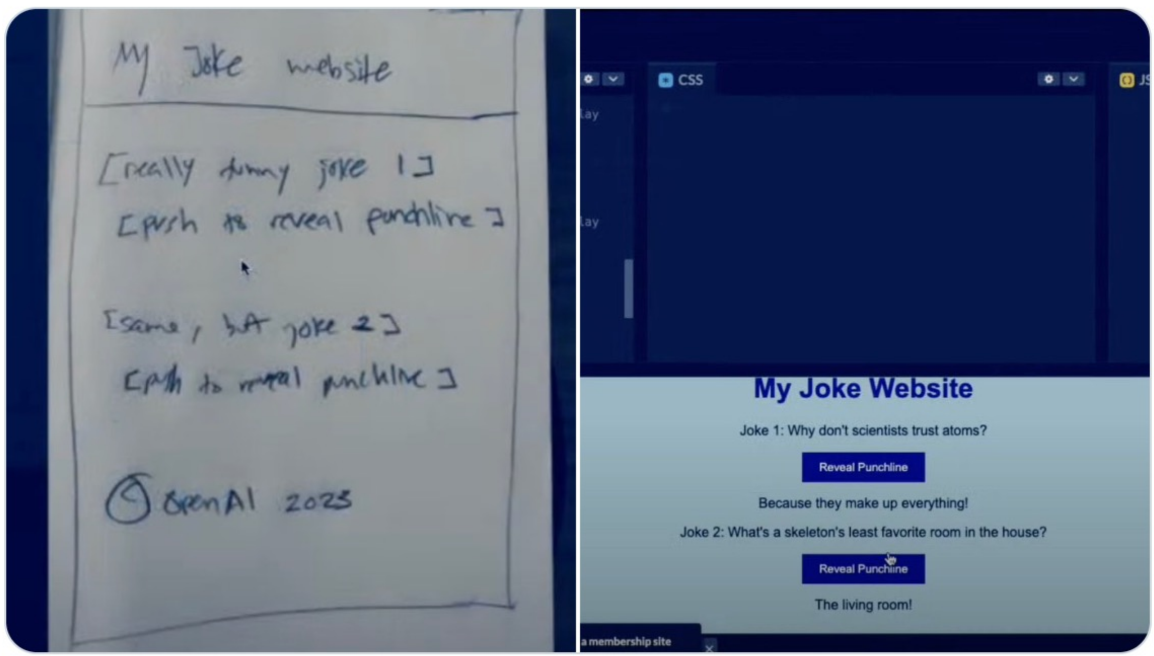
Mckay Wrigley
@mckaywrigley

Greg Brockman (@gdb) of OpenAI just demoed GPT-4 creating a working website from an image of a sketch from his notebook.

It's the coolest thing I've *ever* seen in tech.

If you extrapolate from that demo, the possibilities are endless.

A glimpse into the future of computing.



Level -1

- Can happen as part of a larger application
  - may require code but not deep reasoning
- Happens more often than you think
  - individual function can be level -1

    e.g., two boolean inputs (only 4 configurations)
  - quite common with UI

    e.g., when I click the button, it should say "hi"
- Be on the lookout for these cases
  - save yourself work by spotting them

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	?			
1	?			
2	?			
3	?			

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	?			
2	?			
3	?			

Level 0

- Instructions say exactly how to calculate answer
  - we are just translating math into code
- Still easy to make mistakes
  - too many inputs to test them all
  - need to additional ways of checking for bugs

Non-programming Example

- Important to calculate grades correctly!

```
fx  =0.6*G4+0.15*I4+0.25*J4
```

Homework	Extra Credit	Midterm	Final	Combined
87.5%	1	64.0%	91.6%	0.25*J2
91.4%	1	87.9%	70.8%	85.8%
86.2%	5	93.0%	62.0%	81.8%
96.5%	1	60.9%	69.0%	84.4%
98.2%	0	88.6%	91.3%	95.0%
86.3%	0	91.5%	63.0%	81.3%

- The syllabus says the formula
  - ask someone else to double-check (“code review”)
  - spot check some of them

Programming Example 1

- Implement absolute value

- Specification says |x| = x if x ≥ 0 and –x otherwise
  - definition is an "if" statement

```
function abs(x: number): number {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

Programming Example 2

- Implement factorial

- Specification says 0! = 1 and (n+1)! = (n+1)*n!
  - definition is recursive

```
function factorial(n: number): number {
  if (n === 0) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

Level 0

- Arise more often than you think
  - sometimes the only way to write a specification is to spell out how to calculate the answer

- To make sure its correct, we need:
  - code review: second set of eyes
  - type checker: third set of eyes (so to speak)
  - some tests

    can’t test every case, so we need to pick the right ones

    (more on this next lecture…)

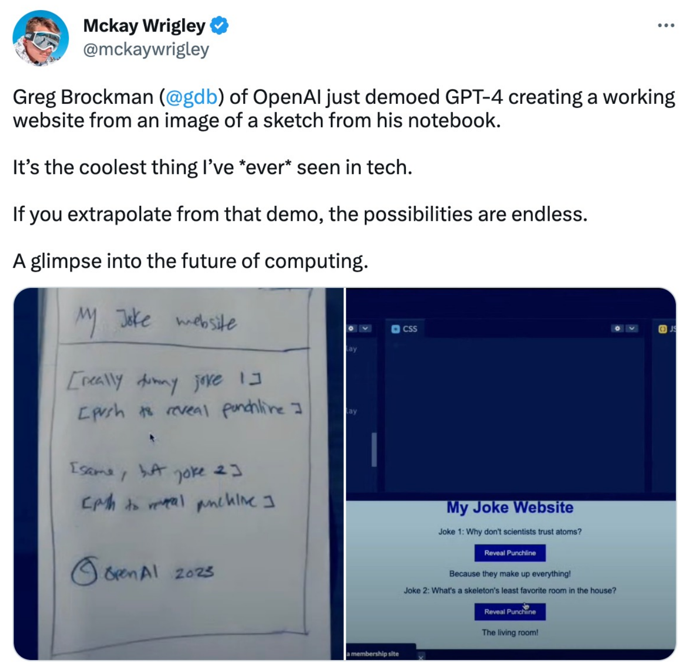
Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	?			
2	?			
3	?			

Another Idea

- **Why not ask the AI if the code is correct?**

- **General case is impossible for any program**
  - see Rice's Theorem (CSE 311)



Programming Example 3

- Implement “maximum of a and b”

- Spec 1: a if a ≥ b and b otherwise
  - definition is an “if” statement
  - says how to compute the answer

- Spec 2: “x such that (x = a or x = b) and x ≥ a and x ≥ b”
  - now level 1
  - some reasoning is required
  - (an example of a “declarative” definition)

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

Reminders

- Now is the time to practice proper technique
  - much harder to learn technique on hard problems
- We will set an **extremely high bar** for correctness
- Temptation to use shortcuts never goes away
  - e.g., skipping reasoning (or tools/testing) on level 0+
- Work skipped now costs 5x as much later
  - much more likely as the code base gets bigger
  - debugging later is harder and more **painful**

Tools

JavaScript and TypeScript

- We will use TypeScript this quarter
  - adds a type system to JavaScript (JS)
  - `tsc` checks types and then removes them (to get JS)

- We will learn the language slowly over the quarter
  - there is no hurry

Type Checkers

- The main part of “Tools” is the type checker

- Type Checkers are very useful for finding bugs
  - another set of “eyes” helping us find them
  - you have probably learned this already

- TypeScript and Java have different type systems…
  - they can catch different bugs for us

Type Checkers

Java and TS differ in what properties they guarantee

- In many areas, TypeScript is more capable:

Condition	Java	TypeScript
x is a string or number	`Object`	`string | number`
x is a string or null	`String`	`string | null`
x is a string	—	`string`
x is a string array	`String[]`	`string[]`
… immutable string array	—	`readonly string[]`

Type Checkers

Java and TS differ in what properties they guarantee

- In many areas, TypeScript is more capable.
- In Java, we are responsible for
  - making sure the argument is really String or Number
  - making sure references are not null
  - making sure arrays are not modified
- In TypeScript, the type checker can do that for us

Type Checkers

Java and TS differ in what properties they guarantee

- In some areas, Java is more capable:

Condition	Java	TypeScript
x is a number	`float`	`number`
x is an integer	`int`	–
x is a non-negative integer	–	–
x is 1, 2, 3, or 4	–	`1 | 2 | 3 | 4`

Type Checkers

Java and TS differ in what properties they guarantee

- In some areas, Java is more capable
- In TypeScript, we are responsible for
  - making sure the argument is really an integer
- JavaScript uses floating point for all numbers
  - can accurately store anything from a Java `int`
  - but we get no help from the type checker