# CSE306 Software Quality in Practice

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

# PRE

- How did it go?
- What were major challenges for your team?

# PRE

- Document baseline approach to SW development in a team environment

- What are we looking for?
  Documentation of process.

- Some teams did not collaborate/ communicate well.
  Something to work on: how can you (as an individual & as a team) encourage/ ensure collaboration and communication?

# Learning outcomes of course

(I) Employ static and dynamic analysis tools to detect faults in a given piece of software.

(II) Employ profiling tools to identify performance issues (both time and memory) in a given piece of software.

(III) Employ testing frameworks to write tests that fail in the presence of software faults, and pass otherwise

(IV) Employ a structured, methodical approach to detecting, testing, identifying and correcting software faults.

(V) Work productively as a member of a software development team.

# Think broadly

# Think broadly

build to LPR

# Think broadly

build to LPR

apply in other courses

# Think broadly

build to LPR

apply in other courses

showcase to potential employers

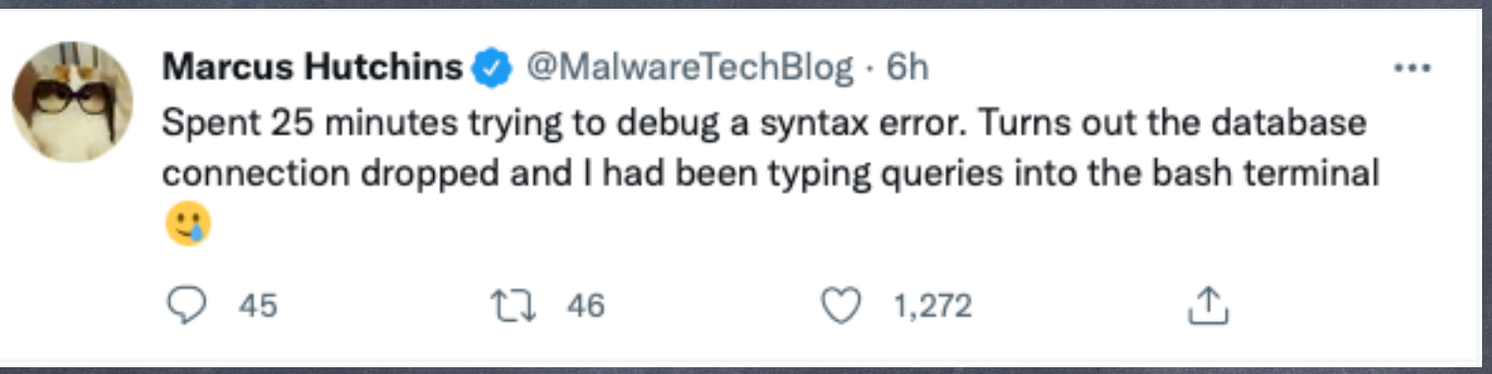
# EXP01

- Released this past Sunday (@123)
- Team-based: same teams as for PRE
- Clone repo via GitHub as usual so course staff can view: 3 (of 16) teams have accepted so far
- Learning goals:
  - show you can apply process
  - show you can use tools effectively
  - show you can engage in teamwork
    - communication and collaboration are key
    - More to come between EXP01 and EXP02

# EXP01

- Demo

# "Check the plug"



# LEX08

.h and .c

# Question for class

When stepping through code with debugger, why are declarations skipped?

```
int foo() {
    int x;
    double y;
    y = f(x) * 3;  // why does debugger skip to here?
    ...
}
```

# Answer

Declarations are handled by compiler at compile time. They have no run-time analogue.

# make

## What is it good for?

"You can use [make] to describe any task where some files must be updated automatically from others whenever the others change."

[https://www.gnu.org/software/make/manual/make.pdf, page 1]

# make and makefiles

- makefile contains rules that describe update dependencies

# rules

```
target : prerequisites
	recipe
```

# rules

target : prerequisites
recipe

Must be a tab!

# target

- A target is usually the name of a file that needs to be generated/ updated during the 'make' process
- The rule will be used by 'make' when the target is out-of-date, and so should say how to update the target

# target

- A target is usually the name of a file that needs to be generated/ updated during the 'make' process
- The rule will be used by 'make' when the target is out-of-date, and so should say how to update the target

"Bear in mind that make does not know anything about how the recipes work. It is up to you to supply recipes that will update the target file properly. All make does is execute the recipe you have specified when the target file needs to be updated." [p. 5]

# target

```
primOpt.o: primOpt.c primOpt.h
        gcc -c -Wall primOpt.c
```

# target

- A target can be "phony" - an arbitrary label for an action given by the rest of the rule

# target

```
clean:
	rm -f prim0pt.o main
```

"...the clean target will not work
properly if a file named clean is ever created
in this directory. Since it has no prerequisites, clean
would always be considered up to date and its recipe would
not be executed. To avoid this problem you can explicitly
declare the target to be phony by making it a
prerequisite of the special target .PHONY" [p. 29]

# target

```
.PHONY: clean
clean:
        rm -f primOpt.o main
```