CSE306 Software Quality in Practice

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

www.cse.buffalo.edu/faculty/alphonce/SP20/CSE306
piazza.com/buffalo/spring2020/cse306
The campus community is reminded that this is also cold and flu season, so it is best to practice healthy habits including frequent hand washing, covering your mouth and nose when coughing or sneezing, avoiding contact with sick people and getting a flu vaccination.

http://www.buffalo.edu/coronavirus
Fix bad code

Write good code
Activities
Assessment
Grading
Learning activities

- **(LEX)** About twenty lab-based exercises, two per week, due in the lab session, done throughout the semester.

- **(PRE)/(PST)** A “process” team project, done twice, once as a pre-assessment in weeks 1 and 2 of the semester, and a second time as a post-assessment in weeks 12 and 13. Students are required to document their development/debugging process.

- **(EXP)** Two three-week exploratory projects. These projects ask student teams to apply the tools and techniques they have been taught up to that point in the course to existing projects provided to them. Students are required to document their use of the tools and the results they obtained.

- **(LPR)** A two-part in-lab practical exam, in week 13.
# Learning outcomes

<table>
<thead>
<tr>
<th>Learning outcome</th>
<th>Instructional methods</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employ static and dynamic analysis tools to detect faults in a given piece of software.</td>
<td>Lecture-based instruction</td>
<td>LEX LPR</td>
</tr>
<tr>
<td>Employ profiling tools to identify performance issues (both time and memory) in a given piece of software.</td>
<td>Lab-based hands-on exercises.</td>
<td>PRE/PST EXP</td>
</tr>
<tr>
<td>Employ testing frameworks to write tests that fail in the presence of software faults, and pass otherwise</td>
<td></td>
<td>LPR</td>
</tr>
<tr>
<td>Employ a structured, methodical approach to detecting, testing, identifying and correcting software faults.</td>
<td></td>
<td>PRE/PST EXP</td>
</tr>
<tr>
<td>Work productively as a member of a software development team.</td>
<td></td>
<td>PRE/PST EXP</td>
</tr>
</tbody>
</table>
## Grading

<table>
<thead>
<tr>
<th>INDIVIDUAL (68%)</th>
<th>TEAM (32%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEX 34%</td>
<td>PRE 4%</td>
</tr>
<tr>
<td>LPR 34%</td>
<td>PST 16%</td>
</tr>
<tr>
<td></td>
<td>EXP 12%</td>
</tr>
</tbody>
</table>
Assessment: Performance Indicators (PIs)

- Each piece of student work will be assessed using performance indicators, each of which has a rubric with four performance levels:
  - “insufficient evidence”
  - “developing”
  - “secure”
  - “exemplary”

- Each performance indicator is assessed independently of the others.
The overall grade for a piece of work is determined by comparing actual performance relative to performance expectations, which increase throughout the course.

Early in a topic the expectation is that performance is close to the "insufficient evidence" level.

Towards the middle of the topic we expect students to be at or above "developing".

Towards the end of the course we expect students to be at or above the "secure" level.

Specific rubrics and performance expectations will be published for each assignment.
ACADEMIC INTEGRITY

- INDIVIDUAL WORK: You must write the code as an individual, and may not seek direct assistance from classmates.
- TEAM WORK: You and your teammates must write the code by together - everyone must contribute.
- You may look up (but must cite!) resources for the necessary algorithms and data structures.
This week...
Teams & Labs

- Form teams of size 3 or 4 this week, by tomorrow's first lab if possible.
- I recommend all team members attend same lab, but not required.
- Labs start this week.
- One member of each team must make a private post in Piazza with the UBIT of each person on their team.
SENS account needed!

- Labs are in Bell 340, a SENS-administered lab.
- If you do not have a SENS account, request one:

  http://www.sens.buffalo.edu/accounts/

Please note: It can take up to one business day for your account to be activated.
(PRE) PROCESS PROJECT

Posted on website

Activity log:
Keep track of how/when you work

Warm-up on C programming
Document your programming process at this point in the course
Compiler

- use gcc compiler with C11 standard
- You can work on any machine, but test on timberlake.cse.buffalo.edu (that's our reference system) before submitting: that's where we'll grade!
Setting the stage...
Is this code buggy?

```c
#include <stdio.h>

int main() {
    printf("Hello, world.");
    return 0;
}
```
It depends - what was the specification for the program?
Should the program check the return value of printf?

What is the return value of printf?
### Table 1. Single Taxable Income Brackets and Rates, 2017

<table>
<thead>
<tr>
<th>Rate</th>
<th>Taxable Income Bracket</th>
<th>Tax Owed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>$0 to $9,325</td>
<td>10% of Taxable Income</td>
</tr>
<tr>
<td>15%</td>
<td>$9,325 to $37,950</td>
<td>$932.50 plus 15% of the excess over $9325</td>
</tr>
<tr>
<td>25%</td>
<td>$37,950 to $91,900</td>
<td>$5,226.25 plus 25% of the excess over $37,950</td>
</tr>
<tr>
<td>28%</td>
<td>$91,900 to $191,650</td>
<td>$18,713.75 plus 28% of the excess over $91,900</td>
</tr>
<tr>
<td>33%</td>
<td>$191,650 to $416,700</td>
<td>$46,643.75 plus 33% of the excess over $191,650</td>
</tr>
<tr>
<td>35%</td>
<td>$416,700 to $418,400</td>
<td>$120,910.25 plus 35% of the excess over $416,700</td>
</tr>
<tr>
<td>39.60%</td>
<td>$418,400+</td>
<td>$121,505.25 plus 39.6% of the excess over $418,400</td>
</tr>
</tbody>
</table>
Is this code buggy?

double f(double x) {
    if (x < 9325) { return 0.1 * x; }
    if (x < 37950) { return 932.50 + 0.15 * (x - 9325); }
    if (x < 91900) { return 5225.25 + 0.25 * (x - 37950); }
    if (x < 191650) { return 18713.75 + 0.28 * (x - 91900); }
    if (x < 416700) { return 46643.75 + 0.33 * (x - 191650); }
    if (x < 418400) { return 120910.25 + 0.35 * (x - 416700); }
    return 131505.25 + 36.9 * (x - 418400);
}

Is this code buggy?

double f(double x) {
    if (x < 9325) { return 0.1 * x; }
    if (x < 37950) { return 932.50 + 0.15 * (x - 9325); }
    if (x < 91900) { return 5225.25 + 0.25 * (x - 37950); }
    if (x < 191650) { return 18713.75 + 0.28 * (x - 91900); }
    if (x < 416700) { return 46643.75 + 0.33 * (x - 196150); }
    if (x < 418400) { return 120910.25 + 0.35 * (x - 416700); }
    return 131505.25 + 36.9 * (x - 418400);
}

The code doesn't protest if x < 0.
What we’ll be doing in this course

- Learn tools to explore program structure and behavior.
- Consider correctness relative to a specification and performance relative to a requirement.
- Employ a methodical approach to tracking down, identifying, documenting and fixing problems with code.
- Employ a methodical approach to developing code.
static vs dynamic program analysis

static analysis - done on program without executing it

dynamic analysis - done on program by executing it
Compiler
a static analysis tool

In recitation this week we will explore what a compiler can and can't tell us about our code.
Fig. 2.1  Simplified build and test flow

2  A Systematic Approach to Debugging
Fig. 2.1 Simplified build and test flow
The 13 Golden Rules of Debugging

1. Understand the requirements
2. Make it fail
3. Simplify the test case
4. Read the right error message
5. Check the plug
6. Separate facts from interpretation
7. Divide and conquer
8. Match the tool to the bug
9. One change at a time
10. Keep an audit trail
11. Get a fresh view
12. If you didn’t fix it, it ain’t fixed
13. Cover your bugfix with a regression test
1. Understand the requirements
1. Understand the requirements

- Is it a bug or a misunderstanding of expected behavior?
- Requirements will tell you.
2. Make it fail
2. Make it fail

- Write test cases to isolate bug and make it reproducible.
- This will increase confidence that bug is fixed later.
- These tests will be added to the suite of regression tests (“does today’s code pass yesterday’s tests?”)
3. Simplify the test case
3. Simplify the test case

- Ensure there is nothing extraneous in the test case.
- Keep it simple! Whittle it down until you get at the essence of the failure.
4. Read the right error message
4. Read the right error message

“Everything that happened after the first thing went wrong should be eyed with suspicion. The first problem may have left the program in a corrupt state.” [p. 9]