CSE306 Software Quality in Practice

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

- Posted on website
- Academic Integrity
The CSE Department has a zero-tolerance policy regarding academic integrity (AI) violations.

When there is a potential violation of academic integrity in a course, the course director shall first notify the concerned students. This notification begins the review and appeals process defined in the University’s Academic Integrity statement:

http://catalog.buffalo.edu/policies/course/integrity.html

Upon conclusion of the review and appeals process, if the department, school, and university have determined that the student has committed a violation, the following sanctions will be imposed upon the student:

§ 1. Documentation. The department, school, and university will record the student's name in departmental, decanal, and university-level academic integrity violations databases. THE UNIVERSITY RECORD IS PERMANENT, AND CAN AFFECT YOUR JOB PROSPECTS (E.G. MEDICAL or LAW SCHOOL).

§ 2. Penalty Assessment. The standing policy of the Department is that all students involved in an academic integrity violation will receive an F grade for the course. The course director may recommend a lesser penalty for the first instance of academic integrity violation, and the adjudication committees that hear the appeal at the department, decanal and provost level may recommend a lesser or greater penalty.
Reminders

- Piazza
- BitBucket private repositories
  - CSE410Solo-<UBIT USERNAME>
  - CSE410Team-<TEAM NAME>
- Add PreProcessProject to team repo
Compiler

- use /util/bin/gcc compiler
- use -std=C11 (you can use other options too)
- test on timberlake.cse.buffalo.edu (that’s our reference system)
- If you modify your .cshrc file you can access the CSE installation of the compiler directly from the Bell 340 machines.
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

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

- 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

"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]
5. Check the plug

- Don’t overlook the obvious - things like permissions, file system status, available memory.

- “Think of ten common mistakes, and ensure nobody made them.” [p. 9]
CONTINUING ON FROM LAST TIME
6. Separate fact from fiction
6. Separate fact from fiction

- “Don’t assume!”
- Can you prove what you believe to be true?
- This is tough - it is hard to set aside your well-founded intuitions!
7. Divide and conquer
7. Divide and conquer

- Beware bugs caused by interactions amongst components.
- Develop a list of suspects (source code, compiler, environment, libraries, machine, etc)
- Each component alone may work correctly, but in combination bad things happen
- Can be especially tricky with multithreaded programs
8. Match the tool to the bug
8. Match the tool to the bug

- If all you have is a hammer ... you’ll end up with a very sore thumb.
- Build a solid toolkit to give you choices.
- Use multiple tools/approaches (e.g. testing and debugging work better together than either along)
9. One change at a time
9. One change at a time

- Be methodical. If you make multiple changes at one time you can't tease apart which change had which effect.

- With your list of suspects, document what you predict the outcome of a change will be.

- Document the changes you make, and the results.

- Did results match predictions?
10. Keep an audit trail
10. Keep an audit trail

- Make sure you can revert your code: use a code repository! This lets you back out changes that were not productive.
11. Get a fresh view
11. Get a fresh view

- Ask for someone else to have a look — but not before having done steps 1 - 10!

- Even just explaining the situation can help you better understand what is happening.
12. If you didn't fix it, it ain't fixed.
12. If you didn't fix it, it ain't fixed

- Intermittent bugs will recur.

- If you make a change to the code and the symptom goes away, did you really fix it? You must convince yourself that the fix you applied really did solve the problem!
13. Cover your bug fix with a regression test
13. Cover your bug fix with a regression test

- Make sure the bug doesn't come back! Just because it worked yesterday doesn't mean it still works today. This is especially important in team environments where you are not the only person touching the code.
Essential tools

- compiler (e.g. gcc)
- debugger (e.g. gbd)
- memory checker (e.g. memcheck)
- runtime profiler (e.g. gprof)
- automated testing framework (e.g. cunit)
- build tool (e.g. make, gradle)
- code repository (e.g. git)
- organization tool (e.g. Trello)
- pad of paper / whiteboard