Final Exam Review

CS 4278/5278: Principles of Software Engineering

Exam Layout

Same as last time

- Tuesday, 4/21
- In-person
- On paper
- 75 minutes
- Open note, open internet, no GenAI, no communicating with classmates
- Timing is intentionally tight to prevent you from not studying and looking things up on the spot during exam time.
    - Pace yourself accordingly
    - Don’t waste time on long-winded answers to the *short* answer questions (bullet points or numbered lists are often good for this).

Delta Debugging

- Delta debugging is an **automated debugging approach** that finds a one-minimal **interesting subset** of a given set.
- Delta debugging is based on **divide and conquer** and relies on critical **assumptions** (monotonicity, unambiguity, and consistency).
- It can be used to find which code changes cause a bug, to minimize failure inducing inputs, and even to find harmful thread schedules.

Delta Debugging

Remember the three main assumptions around Delta Debugging…

- **Monotonicity** - if X is interesting, set of X & anything is interesting
- **Unambiguity** - if X & Y are interesting, intersection of X & Y is interesting
- **Consistency** - X is either interesting or not interesting

And the problems that delta debugging seeks to solve are simplifying, isolating, and identifying failure-inducing components

DD+

- An enhanced version of delta debugging that minimizes failure-inducing inputs more efficiently by reducing redundant tests and improving search strategy.

1. Start with failing input and initial granularity (n = 2)
2. Partition input into n subsets
3. Test:
   a. Each subset
   b. Each complement
4. If failure found:
   a. Reduce input
   b. Keep or refine granularity (don’t reset blindly)
5. If no failure:
   a. Increase granularity ($n \leftarrow \min(|input|, 2n)$)
6. Repeat until minimal failure-inducing set is reached

Fault Localization Overview

- Debugger: **single-stepping** through the program and inspecting variable values.
- Fault Localization: identifying lines implicated in a bug. Humans are better at localizing some types of bugs than others.
- Automatic tools can help with the dynamic analyses of fault localization and profiling

Debugger

- What is a debugger?
  - Can operate on source code or assembly code
  - Inspect the values of registers, memory
  - Key Features
    - Attach to process
    - Single-stepping
    - Breakpoints
    - Conditional Breakpoints
    - Watchpoints

Signals

- Asynchronous notification sent to a process about an event
- Signal handler: a procedure that will be executed when the signal occurs.
    - vulnerable to race conditions

Fault Localization Tools

- Coverage-Based Fault Localization

Statement	3,3,5	1,2,3	3,2,1	3,2,1	5,5,5	2,1,3
int m;						
m = z;						
if (y < z)						
if (x < y)						
m = y;						
else if (x<z)						
m = y; // bug						
else						
if (x > y)						
m = y;						
else if (x>z)						
m = x;						
return m;						
	Pass	Pass	Pass	Pass	Pass	Fail 39

Fault Localization Tools

- Spectrum-Based Fault Localization
  - Ranks lines based on suspiciousness score
  - Informally: The more a line is covered by failing tests but not covered by passing tests, the more suspicious

$$S(s) = \frac{\text{failed}(s) \ / \ \text{totalfailed}}{(\text{failed}(s) \ / \ \text{totalfailed}) \ + \ (\text{passed}(s) \ / \ \text{totalpassed})}$$

Profiling

- A profiler is a performance analysis tool that measures the frequency and duration of function calls as a program runs.
- A flat profile computes the average call times for functions but does not break times down based on context.
- A call-graph profile computes call times for functions and also the call-chains involved
- E.x., event-based profiling, statistical profiling

Event-Based Profiling

- Interpreted languages provide special hooks for profiling
  - Java: JVM-Profile Interface, JVM API
  - Python: sys.set_profile() module
  - Ruby: profile.rb, etc.
- You register a function that will get called whenever the target program calls a method, loads a class, allocates an object, etc.
  - cf. “signal handler”

Statistical Profiling

- You can arrange for the operating system to send you a signal every X seconds
- In the signal handler you determine the value of the target program counter
  - And append it to a growing list file
  - This is sampling
- Later, you use debug information from the compiler to map the PC values to procedure names
  - Sum up to get amount of time in each procedure

Sampling Analysis

- Advantages
  - Simple and cheap – the instrumentation is unlikely to disturb the program
  - No big slowdown
- Disadvantages
  - Can completely miss periodic behavior (e.g., you sample every k seconds but do a network send at times 0.5 + nk seconds)
  - High error rate

Requirements Engineering

- **Requirements** articulate the relationship and interface between a desired system and its environment.
- **Requirement Engineering** is the process of identifying, eliciting, analyzing, specifying, validating and managing the needs and expectations of stakeholders for a software system.
- An **informal goal** is a general intention (e.g., “ease of use” or “high security”)
- A **verifiable non-functional requirement** is a statement using some measure that can be objectively **tested**

Goal → Requirement Example

- **Informal goal**: “the system should be easy to use by experienced controllers, and should be organized such that user errors are minimized.”
- **Verifiable non-functional requirement**: “Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day, on average.”

Requirements Engineering Steps

1. Identifying stakeholders
2. Domain understanding
3. Requirements elicitation (interviews, …)
4. Evaluation and agreement (conflicts, prioritization, risks, …)
5. Documentation and specification
6. Consolidation and quality assurance

Requirements Engineering Steps

- **Functional requirements** describe what the machine should do ("get the right answer")
  - Inputs/Outputs, Interface, Response to events

- **Quality requirements** specify not the functionality of the system, but the manner in which it delivers that functionality
  - Maintainability, Efficiency, Security, Useability, Reliability/Availability, Scalability, Portability/Compatibility, Cost

UI/UX

- **User Experience (UX)** is how a user interacts with and experiences a product, system, or service. It includes a person's perceptions of utility, ease of use, and efficiency.
- **A User Interface (UI)** is an aspect of UX focused on the aesthetics of how a user interacts with the system.
  - Can be graphical (GUI) or textual.

UI Design Priciples

- **Consistency**: Similar things should look and behave the same
- **Feedback**: The system should tell the user what is happening
- **Reversibility**: Users should be able to interrupt or reverse actions
- **Affordance**: Controls should look like how they are used buttons look clickable, scrollable areas have a scroll bar, etc
- **Error Handling**: Handle errors gracefully instead of showing error messages
- **Simplicity**: Don’t make users think too much

UX Design Priciples

- Know your users
- Know their goals
- Minimize steps
- Reduce cognitive load
- Design for errors
- Make the common case fast and easy
- Make the system predictable

Usability Testing

- Give your UI to a user and watch them try to use it
- Don’t explain the interface
- Watch where they struggle
  - Gain insights into how to make the UI easier
- Measure **interaction costs**, the number of interactions it takes to perform common tasks.
- A/B testing is a common way to measure preference and usability

Continuous Integration / Continuous Deployment (CI/CD)

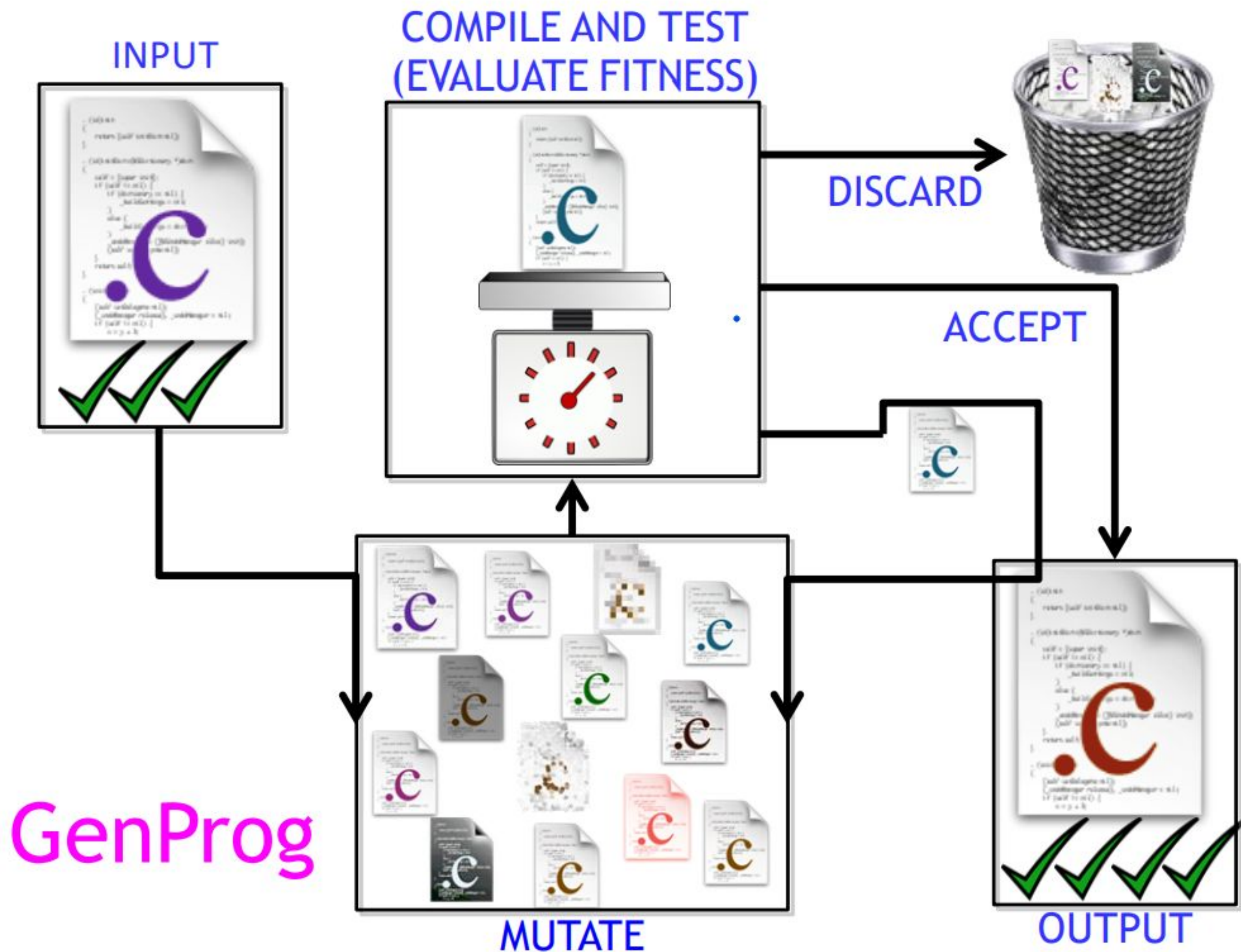
- DevOps practice that automates building, testing, and deploying code changes for faster and more reliable software releases.
- Shorten the time between code changes and production deployment
- Developers frequently merge code → system automatically builds and runs tests.
- Emphasizes communication between team members and automation of testing/deployment

Automatic Program Repair

- Can use strategies and techniques learned in this class to find evidence of and fix existing bugs
- Fault localization, mutation, testing to find/fix bugs
- A patch might contain extraneous edits (use delta debugging to minimize)
- Each repair has to pass the whole test suite
- Can use static analysis to prevent testing “duplicates” aka equivalent patches

Automatic Program Repair

- Ideally…
  - Mutation testing takes a program that passes all tests, and human mistake-based mutants (that aren’t equivalent) must fail at least one test
  - Program repair takes a program that fails test suite, requires that one mutant (based on human repairs from fault localization) only passes all tests



INPUT
COMPILE AND TEST
(EVALUATE FITNESS)
DISCARD
ACCEPT
GenProg
MUTATE
OUTPUT

Automatic Program Repair

- APR is good at fixing lots of bugs
  - Typically require small changes
  - Changes typically have to be AST modifications
- APR isn't so good at other types of bugs (yet)
  - Particular values being off
  - Bugs that require human expertise
- LLM-based APR is better at a wider range of bugs

Productivity

- Experiment with system response time
  - Short term mental memory buffer can be disrupted by increased system response time
  - Faster response time enabled significant performance enhancement
  - Cost of upgrading a processor can be more than justified by savings in user time
- “Programming speed” - higher-order language, less CPU time, faster coding
- “Program economy” - faster running programs, experience, lower-level language

Productivity

- Experts just solve problems in one step - quicker
- Novices focus more on surface features
- Experts focus on underlying principles
- With learning, shift in how knowledge is organized (from surface to principles)
- Improving how one learns would be done by identifying available knowledge and manipulating or working off of that

Productivity

- Main idea: programming speed (associated with a higher-order language, faster coding, less CPU time) is a commonly mistaken belief
- Using abstraction is the real path to success
- Can get abstraction through language, or other avenues - the ideal of abstraction is the insight
- Abstraction can take years, but that is the true limitation to productivity

Next Time

- AI in SWE
- How to use AI tools to improve your coding and productivity
- This will be a demo
- Complete the presurvey before class
- Attendance is required
- **Bring your laptop!!!**

Course and Teaching Evaluations

- Please take the rest of class time to fill out the course and teaching evaluations!