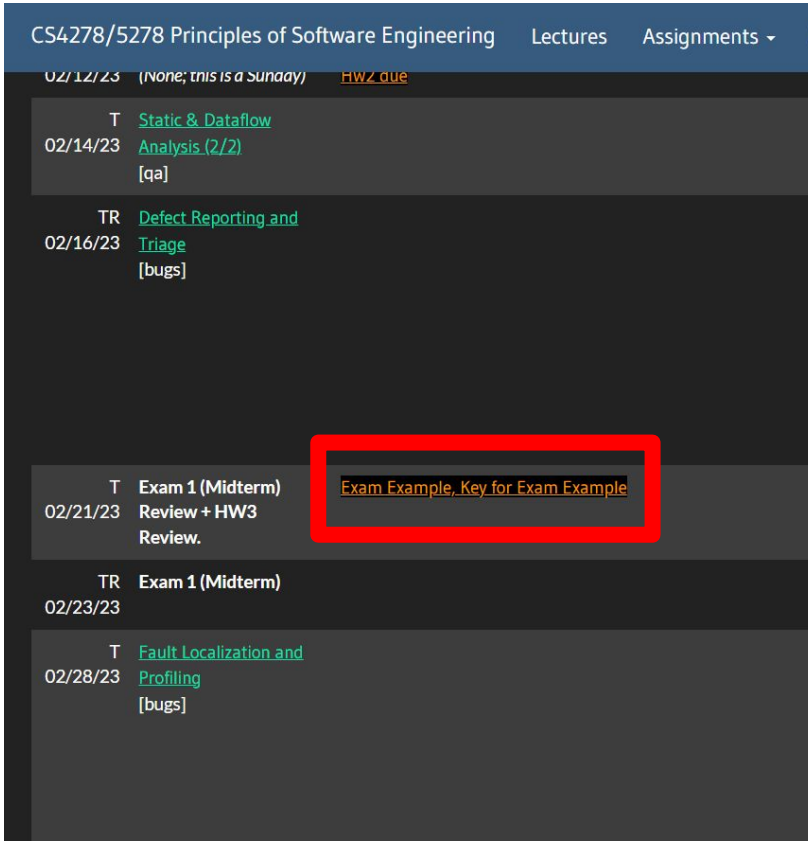
Exam 1 and HW3 Review

CS 4278/5278: Principles of Software Engineering

Skyler Grandel
Graduate Teaching Assistant
skyler.h.grandel@vanderbilt.edu

Exam 1

- Thursday Feb 23
- 12:01 am – 11:59 pm
- 2 hours
  - Latest start time: 9:59 pm
- TAs available 9 to 9
- Practice Link on Piazza



Exam Structure

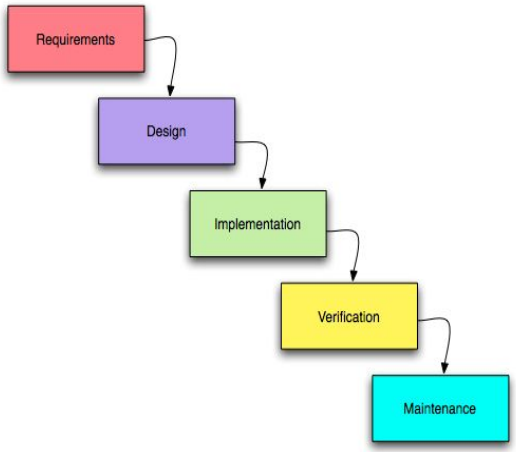
- The exam will be a webpage on Dr. Huang’s website
- 6 multipart questions + 1 bonus
- Short answer, answer bank, fill in the blank

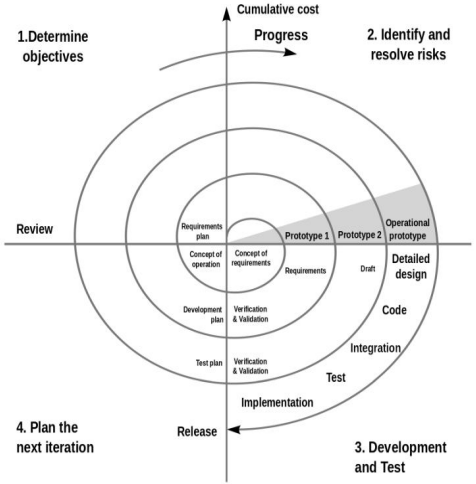
Exam 1 Material

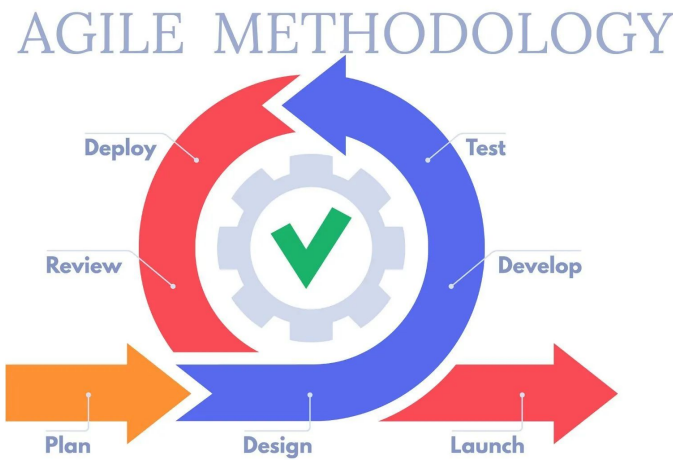
- Process, Risk, Scheduling
- Measurement and Quality Assurance
- Testing and Code Review
- Dynamic, Static, and Dataflow Analysis
- Defect Reporting and Triage

Process, Risk, Scheduling

- Spiral, Agile, Waterfall - advantages and shortcomings







Process, **Risk**, Scheduling

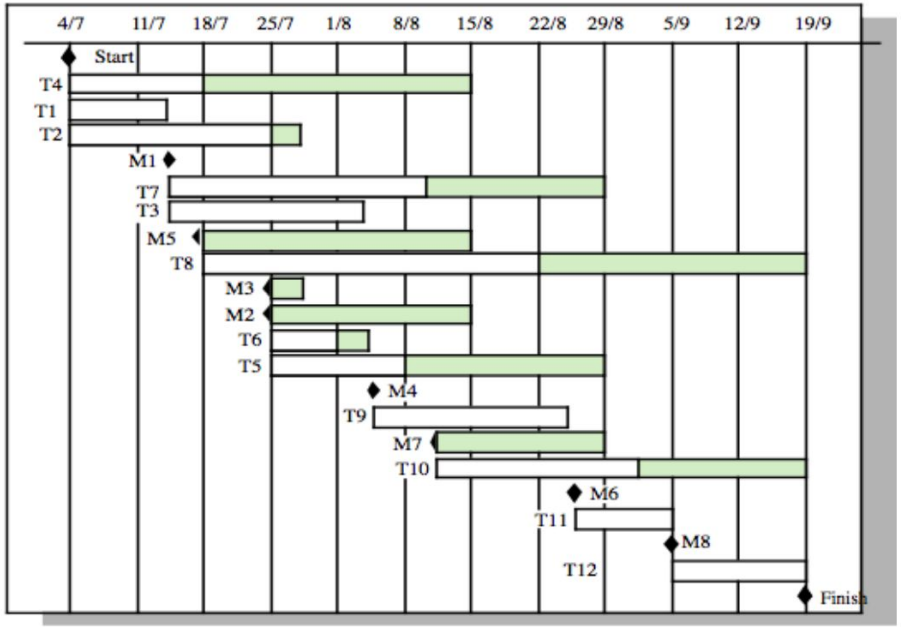
- Zero-risk bias - prefer eliminating risk over larger reduction in risk
- Risk management as a project management tool
- Trade-offs and benefits of proper risk management
- Best practices for managing risk
- Balancing risk and innovation

Process, Risk, **Scheduling**

- Strategies to estimate time for a project (cocomo)
- Relationship between scheduling and risk
- Milestones vs. deliverables
  - Endpoint of a task vs Results for the customer
- “Almost done” problem

Process, Risk, **Scheduling**

- Visualization - Gantt Diagram



Measurement and Quality Assurance

- Measurement - how is it applied to SE?
- What decisions can be made based on metrics?
  - Where should funds/effort be allocated?
- McNamara Fallacy
  - Making decisions based solely on quantitative metrics
- Maintainability Index - general purpose
  - Halstead Volume, Cyclomatic Complexity, LOC

Measurement and Quality Assurance

- Types of validity (construct, predictive, external)
- Streetlight effect
  - Searching for something and looking only where it is easiest
- Statistics: false positive paradox, correlation != causation, confounding variables
- Metric-based incentives

Measurement and **Quality Assurance**

- What is Quality Assurance?
- Halting Problem in QA
  - We can never be sure a program is correct
- Testing can give us an estimate
  - Demonstrates the presence of bugs, not their absence

Testing and Code Review

- XUnit & unit testing frameworks
  - Write tests that look like other code
- Another process in test-driven development
- Mocking and its applications (with APIs)
  - Writing code to approximate unavailable objects
  - Dynamic and static mocking

Testing and Code Review

- Types of testing
  - Regression - running old tests
  - Unit - test individual pieces
  - Integration - end-to-end testing
  - Fuzz - testing lots of random inputs
  - Penetration - testing for security vulnerabilities

Testing and Code Review

- Bias in testing (test what works!)
- Coverage as a metric for test suite comparison
- Coverage instrumentation and relation to observer effect
  - Instrumenting a program could change its behavior
- Branch & line coverage
  - Branch is more difficult but gives more confidence
  - You should be able to calculate both

Testing and Code Review

- Alpha Testing - by devs
- Beta Testing - by external users
- A/B testing - show impact of a difference in one feature
- Sample *common* and *harmful* functionality with tests

Testing and Code Review

- Mutation testing - defect seeding to test quality of a test suite
  - i.e: intentionally adding bugs
- Mutation operator and mutant orders
- Competent programmer hypothesis and relation to mutation
- Equivalent mutants
- Coupling effect
  - simple faults are coupled with complex ones

Testing and Code Review

- Test case: input data, oracle, comparator
- Test case minimization
- Coverage branch edges vs. paths
- Enumerating paths and loops in a program

Testing and Code Review

- Test generation - DART approach
- Invariants and oracle inference
  - predicate over expressions that is true on all executions.
- Test generation tools
  - Pex
  - EvoSuite

Testing and **Code Review**

- Code review - find defects, improve quality
- Formal code inspection
  - More formal and holistic
- Pull request - proposed changes to merge into a repository

Testing and **Code Review**

- Inspection incentive and root cause analysis
  - Why inspect? To prevent problems from reoccurring
- Metrics on inspection (efficacy, speed, fatigue, etc.)
- Different types
  - Formal inspection, walkthrough, **pair programming**, passaround, ad hoc

Dynamic, Static, and Dataflow Analysis

- Dynamic analysis - analyzing a program by running it
- Assists with hard-to-test bugs
- Race condition - output depends on sequence of “uncontrollable” events
- Steps
  - Run program systematically (controlled input or environment)
  - Monitor internal state at runtime
  - Analyze results

Dynamic, Static, and Dataflow Analysis

- Edge and path coverage
- Taint tracking using sources and sinks
- Execution time profiling
- Focus on one property of output information for dynamic analysis
- Input dependent analysis

Dynamic, Static, and Dataflow Analysis

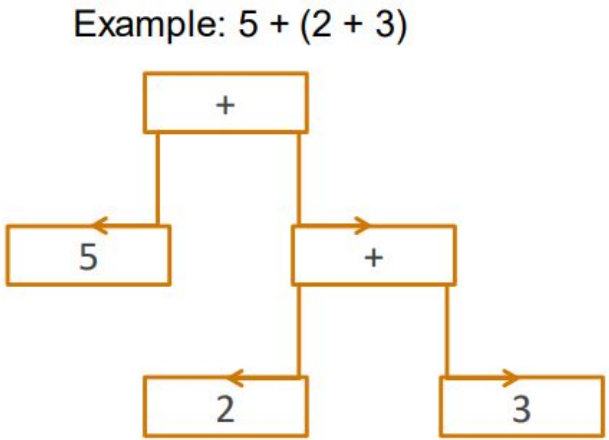
- Examples of dynamic analysis
  - Eraser
  - Chaos Monkey
  - CHESS
  - Driver Verifier
  - Testing!

Dynamic, **Static, and Dataflow Analysis**

- Static analysis - analysis of code *not* at runtime
- Dataflow analysis - approach to static analysis
- Main ideas
  - Abstraction as hiding unnecessary details to simplify program
  - Programs being simplified down to trees, graphs, or strings

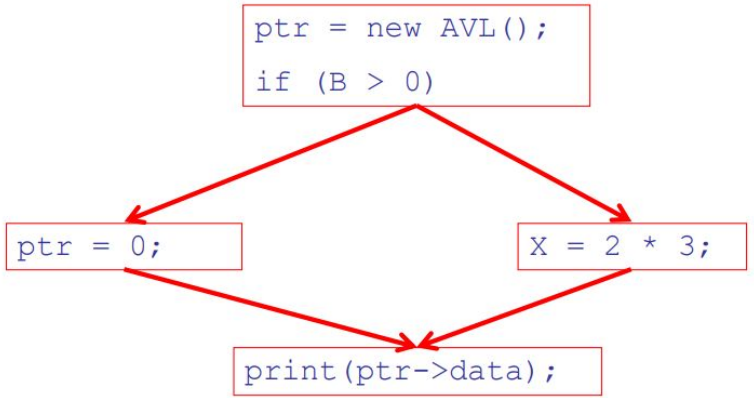
Dynamic, **Static, and Dataflow Analysis**

- Abstract Syntax Tree represents syntactic structure of source code



Dynamic, **Static, and Dataflow Analysis**

- Dataflow analysis
  - Gather information on the possible set of values at various points
  - Definite null dereference on CFG

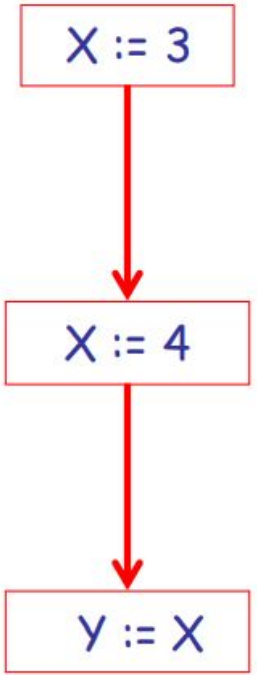


Dynamic, **Static, and Dataflow Analysis**

- Rice’s Theorem and Undecidability of Program’s Properties
  - All of the interesting properties of a program are undecidable
  - Conservative Program Analyses (imprecision)

Dynamic, **Static, and Dataflow Analysis**

- Rules for transfer functions: ⊥, T, a
  - Forward analysis
- Live variables
  - Backward analysis
  - If the current value of a variable is never used, the variable is considered to be dead



Defect Reporting and Triage

- Fault - exceptional situation at run time
- Defect - characteristic of a product which hinders its usability for its intended purpose
- Bug report - Accurately and precisely describe the bug and how to reproduce it
- Triage - measure of urgency

Homework 3 Intro

CS 4278/5278: Principles of Software Engineering

Starting Point

- Grading server uses **Python 3.5.2**
- Read documentation on the **ast** module and the **astor** module.
- You should submit a single file, “mutate.py”
  - The program should generate mutants that are named “0.py”, “1.py”, … (up to 100 files)
  - Other outputs are ignored

- Can someone quickly explain what mutation testing is?
  - hint: make sure to review mutation testing for the exam!

Mutation Operators

- You should implement and support the following three mutation activities:

1. Negate any single comparison operators (>= becomes <, = becomes !=)
2. Swap binary operators +, and -, as well as * and //.
3. Delete an assignment or function call statement.

Held-Out Test Suites

- Test Suites A, B, C, D, and E have 92%, 91%, 90%, 88%, and 79% statement coverage of fuzzywuzzy, respectively. These suites have 80, 57, 47, 32, and 9 tests, respectively.
- **Swap Binary Operators** to distinguish Test Suite A and B from C, D, and E
- **Swap Comparison Operators** to distinguish between Suite B, C, D, and E
- **Delete Assignments and Function Calls** to distinguish C, D, and E. With care, to distinguish between A and B.
- **Higher-Order Mutation** may distinguish Test Suites B, C, and D.
- **Use Creativity** to distinguish between Test Suite A and B
    - hint: try changing assignments!

Starter Code

```
import ast
import astor
with open(“xxx.py”, “r”) as src:
    # convert BinOp “+” to “-”
    tree = ast.parse(src.read())
    new_tree = AddTransformer().visit(tree0)  // how to write a transformer?
    file = astor.to_source(new_tree).strip()
    # then write to an output file
```

Starter Code

- How to write a Transformer?
  - Read the ast.NodeVisitor and ast.NodeTransformer sections in ast documentation
    - NodeTransformer is a subclass of NodeVisitor (recall ISD concepts…)
    - Use inheritance to create different transformers:
      - e.x., **class AddTransformer(ast.NodeTransformer)**
      - Transformer subclasses should have a visitor function (see documentation)
    - https://docs.python.org/3/library/ast.html
  - How do I parse a python file into an AST? How do I turn an AST into a source file?
    - Read documentation on ast.parse, ast.dump, astor.to_source, etc.
    - https://astor.readthedocs.io/en/latest/
- Is this the only approach?
  - No, previous students have tried several other approaches that worked well!
  - The transformer approach above is one that should be straightforward

Common Pitfalls and Advices

1. Don’t start with higher order mutants!
   a. Though carefully designed higher order mutants can be important, most higher order mutants have a high chance of being detected by every test suite.
2. Increasing the odds of one mutation operator also effectively reduces the odds of the others (since you can only produce a fixed number of mutants)
3. Be careful not to mistakenly share the tree data structure between mutants, as you may end up with more edits than you thought
4. Try making a high-quality test suite locally and evaluating against it.
5. Make sure you actually have a chance of mutating every relevant node.
6. You may want to implement additional mutation operators.
   a. See https://huang.isis.vanderbilt.edu/cs4278/readings/mutation-testing.pdf
7. After creating your mutants, you should run pylint to minimize the number of linting/syntax errors reported

Interested in AST?

Take Compilers (CS 3276/5276)!