EECS 481 — Software Engineering
Winter 2020 — Exam #1

- **Write your UM uniqname and UMID and your name on the exam.**
- There are nine (9) pages in this exam (including this one) and seven (7) questions, each with multiple parts. Some questions span multiple pages. If you get stuck on a question, move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two page-sides of notes.
- Even vaguely looking at a cellphone or similar device (e.g., tablet computer) during this exam **is cheating**.
- Use a **dark pen or pencil** to write your answers in the space provided on the exam. If our scan does not pick up your pen or pencil you will not receive credit. You may use exam margins for scratch work. Do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. We may deduct points if your solution is far more complicated than necessary.
  - *Good Writing Example:* Testing is an expensive activity associated with software maintenance.
  - *Bad Writing Example:* Im in ur class, @cing ur t3stz!1!
- If you leave a non-extra-credit portion of the exam blank or drawn an X through it, **you will receive one-third of the points (e.g.,** $4/3 = 1.33$**), for that portion for not wasting time.**

UM uniqname: ANSWER KEY

UM ID: ANSWER KEY

Name (print): ANSWER KEY

1 Software Process Narrative (13 points)

(1 pt. each) Read the following narrative. Fill in each ____ blank with the single *most specific or appropriate* corresponding concept from the answer bank. (Each ____ blank does have a corresponding answer.) Each option from the answer bank will be used *at most once*.

A. Alpha Testing	B. Beta Testing	C. Call-Graph Profile	D. Comparator
E. Conditional Breakpoint	F. Dataflow Analysis	G. Development Process	H. Dynamic Analysis
I. Flat Profile	J. Formal Code Inspection	K. Integration Testing	L. Mocking
M. Oracle	N. Passaround Code Review	P. Perverse Incentive	Q. Priority
R. Quality Property	S. Severity	T. Software Metric	U. Spiral Development
V. Threat to Validity	W. Triage	Y. Watchpoint	Z. Waterfall Model

A software company is working on a new poetry puzzle game based on the works of golden age Tang Dynasty poet Li Bai and 20th-century activist poet Maya Angelou.

__G__ The company decides to organize its software engineering efforts into distinct phases.

__L__ To save resources, calls to a prose-fetching library always return the exact same text from *I Know Why The Caged Bird Sings*, rather than the actual prose requested.

__N__ Developers perform an analysis that they expect will find defects and improve code. While many other activities, such as code inspection and static analysis can also find defects, the key here is the *expectation* of finding defects and improving code — see the readings.

__K__ After separately constructing an Eastern Poetry module and a Western Poetry module, management instructs engineers to determine if they work together.

__B__ Before the game is released, some users outside the company are asked to comment on a puzzle related to *A Quiet Night Thought*.

__D__ Because of context-specific variations in translation, developers arrange for test answers such as "Thoughts in the Silent Night" or "Contemplating Moonlight" to be deemed acceptably close to the reference answer of "A Quiet Night Thought".

__R__ Developers are concerned about the software's performance on mobile devices.

__C__ To resolve performance problems, engineers desire a nuanced report that explains the time taken by each procedure and its children.

__W__ A report related to "moonlight" is received, but it is judged to be too vague to pursue.

__S__ A defect related to the handling of the *Still I Rise* puzzle is perceived to be of strong relevance to users.

__Y__ To track down a bug related to corruption of the global `stanza` variable, developers arrange to be alerted whenever its value changes.

__H__ Developers determine the `stanza` bug to be caused by a race condition, so they run the program and track the set of locks held at each access to it.

__P__ Management awards a monetary bonus to developers for each document written in rhyming couplets, hoping to improve creativity. Developers focus on writing rhymes.

2 Testing and Coverage (20 points)

Note: The printing of the exam used live in Winter 2020 featured typos for Question 2(a). The exam key presents a corrected version of the problem first and gives an explanation for how to argue the unsolvability of presented formulation later.

(4 pts. each) Consider the following program with blanks. We are concerned with statement coverage, but only for the statements labeled `S_1` through `S_5`.

```
1  void shamir(int a, int b, int c) {
2    S_1;
3    if (a == _5_)                  { S_2; }
4
5    if (_b_ == 3)                  { S_3; }
6
7    if (a _<_ b)                   { S_4; }
8
9    if (a _+_ c == b)              { S_5; }
10 }
```

Fill in each blank in the program with a **single** letter, integer or operator so that it matches the following coverage results. Here are four test inputs:

`T_1 = shamir(1,2,3)`	`T_2 = shamir(5,3,2)`	`T_3 = shamir(5,10,5)`	`T_4 = shamir(3,3,4)`

And here are some corresponding coverage measurements for statements `S_1` through `S_5`:

$\{$`T_1`$\} \mapsto 2/5$	$\{$`T_2`$\} \mapsto 3/5$	$\{$`T_3`$\} \mapsto 4/5$	$\{$`T_4`$\} \mapsto 2/5$
$\{$`T_1, T_2`$\} \mapsto 4/5$	$\{$`T_2, T_3`$\} \mapsto 5/5$	$\{$`T_3, T_4`$\} \mapsto 5/5$	$\{$`T_2, T_4`$\} \mapsto 3/5$

With the above answer, the actual statements visited are:

```
T1 = shamir(1,2,3)   // S1,         S4
T2 = shamir(5,3,2)   // S1, S2, S3
T3 = shamir(5,10,5)  // S1, S2,     S4, S5
T4 = shamir(3,3,4)   // S1,     S3
```

You can see this using the following Python code:

```
def shamir(a,b,c):
    print("S1")
    if (a == 5):    print("S2")
    if (b == 3):    print("S3")
    if (a < b):     print("S4")
    if (a+c == b):  print("S5")
```

```
print(" T1")
shamir(1,2,3)
print(" T2")
shamir(5,3,2)
print(" T3")
shamir(5,10,5)
print(" T4")
shamir(3,3,4)
```

(4 pts.) Suppose we define full *arithmetic coverage* to require that every arithmetic expression evaluate to both a positive number and a negative number. This is analogous to how branch coverage requires each branch to evaluate to both true and false. Give a smallest test suite for `allen()` below that maximizes *arithmetic coverage*. (Boolean expressions are not arithmetic expressions. Zero is neither positive nor negative.)

```
1 void allen(int a, int b) {
2   int p, q, r;
3   p = a + b;
4   q = a * b;
5   if (a < b)    { r = p * q; }
6 }
```

Answer: Test1 = {1, 2}, Test2 = {-2, 1}. Test3 = {-1, 2}.
There are three arithmetic expressions. Test1 makes them 3 2 6 = Pos, Pos, Pos. Test2 makes them -1 -2 2 = Neg, Neg, Pos. Test3 makes them 1 -2 -2 = Pos, Neg, Neg.

(Question 2(a) alternate.) The live printing of Question 2(a) in Winter 2020 featured "else if" on line 9 and a coverage of {T_4} = 3/5. That version is unsolvable. While this course features the occasional unsolvable problem (e.g., Winter 2018 Question 2(b)'s final true branch, or one of the homework assignment subproblems, testing that a program does all and only good things in the "Test Suite Quality Metrics" slideset, etc.), and thus such considerations are in scope, they are generally rare. We gave full credit for either "close" solutions or arguments of impossibility.

We give one example of an impossibility argument. Consider a version of the code with an "else if" on line 9. We focus on that the final guard: `else if (a ___ c == b)`. Broadly, the blank must be filled by an arithmetic operator like +, -, * or /. (Attempts to use a relational operator like <= and compare the boolean result against an integer end up not working because b is never 0 or 1.) We reason by cases.

(a) Case -. Only T_2 makes that guard true, so only test suites involving T_2 can have full 5/5 coverage, which contradicts $\{T\_3, T\_4\} \mapsto 5/5$.

(b) Case * or / or +. No test makes any those guards true (to see this, copy the Python code above, change the line, and rerun it), which contradicts $\{T\_3, T\_4\} \mapsto 5/5$.

Since there is no possible value that can be filled in for that last guard, the problem formulation with the `else` on line 9 is not solvable. Note that this level of detail was *not* expected of students.

3 Short Answer (17 points)

(a) *(4 pts.)* In Dr. Leach's guest lecture, one claim considered was that certain development processes can make creative activities, such as research, inefficient. Support or refute that claim using two examples.

Likley "support". Dr. Leach talked about processes including issue backlogs, sprints, research project descriptions, industrial code management and merge requests (with MargeBot), the pipeline process from precheck through testing through coverage to building to deployment, and the hiring process.

Responses might use any of the above examples from the guest lecture. For example, respones might argue that there is a mismatch between the use of Jira, a particular issue and project tracking system, and the description of an early-phase research project exploring certain aspects of conversational AI. Since whether or not we use a process is a "software process" decision, strong answers will reference concepts related to software processes (e.g., risk, uncertainty, planning, measurement, etc.). In general, the more risk or uncertainty there is associated with even what steps an activity (e.g., research) will take, the more a one-size-fits-all process is likely to chafe and feel inefficient.

Because Dr. Leach asked that his talk and slides not made public, no other company-specific details are mentioned in this answer key.

(b) *(6 pts.)* Your company wants to build high-quality software quickly and inexpensively. The company currently does not use static analysis and is considering re-allocating 10% of its testing effort/budget to static analysis. Identify two *risks* with this proposal. For each risk, identify one associated *uncertainty* and one associated *measurement* that might be taken to reduce that uncertainty.

In general, risks and uncertainties often feel very similar. Examples:

SA may be less effective at finding bugs than testing — will SA be effective on our software? — average number of bugs found per line of code

SA may find unimportant bugs compared to testing — do SA-found bugs matter to management? — average assigned priority of SA-reported defects

SA may not fit with our continuous integration framework — will SA provide useful information rapidly? — average seconds for SA to analyze our software after a code check-in

SA may not always be available — what if SA requires one particular expert developer to get it to run and that person gets sick? — fraction of developers who can run SA on hello-world after 30 minutes of training

SA's reports may be ignored — will developers trust the SA? — fraction of SA reports that are false positives (since developers hate those in bug-finding tools)

(c) *(4 pts.)* Support or refute the claim that a "passaround *triage* review" process would be have a net benefit compared to a current practice in which initial triage involves a single decision maker. You can define passaround triage review as you like, but it should feature an initial triage proposal being evaluated and commented on by other developers, analogous to code review.

Likely "refute". First, this plan makes the most sense if it either saves time (i.e., would be faster than current triage) or is more accurate (i.e., if current triage is often wrong). However, this necessary takes at least as much time as standard triage (two people separately looking at the report take at least as much time as the slower one of them). In addition, we did not encounter any evidence that triage is commonly wrong, or that wrong triage assignments are expensive to fix. (By contrat, we did see evidence that patches written by humans are often reverted, but that's a code review issue, not a triage review issue.) A third angle would be to imagine that triage is often noisy or uncertain, and thus asking multiple people would reduce variance or risk. In large software deployments, however, a bug is often triaged to whoever "owns" the correpsonding module, and two annotators are likely to both agree that that Developer X "owns" module Y (e.g., based on the organizational chart or the commit logs). A fourth angle would be to note that many of the properties of patches that make code review attractive for them (e.g., fixing bugs, sharing library knowledge, etc., from the "Expectations" reading) do not really apply to triage.

(d) *(3 pts.)* Support or refute the claim that it should be simple and quick for any software developer at your company to build and test any software in your company's repository. Use two examples as evidence.

Likely "support". This claim is taken directly from the Henderson *Software Engineering at Google* reading, Section 2.2. "These standard commands the highly optimized implementation mean that it is typically very simple and quick for any Google engineering to build and test any software in the repository. This consistency is a key enabler which helps make it practical for engineers to make changes across project boundaries." A response could use consistency and cross-project development as examples. This is a key factor in continuous integration testing for many GitHub projects, for example: those favor allowing anyone around the world to pick up the project and contribute. Directly mentioning the relevant reading in your response would be strong.

However, other angles are possible. In a project that is making heavy use of *mocking*, for example, it may be difficult for developers not familiar with a given module to understand tests or results based on mocked values. In a project that interacts with the real world (like the ATM or music player or autopilot examples from the lectures) it may be that tests involve a physical component (e.g., actually producing sound through a speaker or moving a rotor blade on a quadcopter) that are necessarily more complicated than "pure software" tests. As a third example, in a project that involves regulatory compliance (like many of the medical device examples from class), it may be that only certain developers can work on certain datasets (e.g., for statutory or privacy reasons).

4 Mutation Testing (16 points)

Consider this method to compute the factorial of a number. The original program is shown on the left; three first-order mutants are each indicated by a comment on the right.

```
1  def factorial(n):
2    i = 1
3    fact = 1
4    if (n < 0):              # Mutant 1 has n <= 0
5      return "error"
6    else:
7      while (i <= n):        # Mutant 2 has i < n
8        fact = fact * i      # Mutant 3 has fact = fact + i
9        i = i + 1
10     return fact
```

(a) *(12 pts.)* Complete the table below by indicating whether or not each test kills each Mutant. Write "K" for Killed and "N" for not killed.

	Input (n)	Oracle	Mutant 1	Mutant 2	Mutant 3
Test 0	0	1	**K**	N	N
Test 1	1	1	N	N	**K**
Test 2	2	2	N	**K**	**K**
Test 3	3	6	N	**K**	**K**

(b) *(1 pt.)* What is the mutation score for Tests 0–3 using Mutants 1–3?

The mutation score is the number of mutants killed divided by the total number of mutants. 3/3 = 100%.

(c) *(1 pt.)* What is the mutation score for only Tests 0–**1** using Mutants 1–3?

2/3 = 66%. Note that this question explicitly only considers the first two tests, so Mutant 2 is not killed.

(d) *(2 pt.)* Support or refute the claim that a *higher-order* mutant that combines Mutation 1 and Mutation 3 is useful.

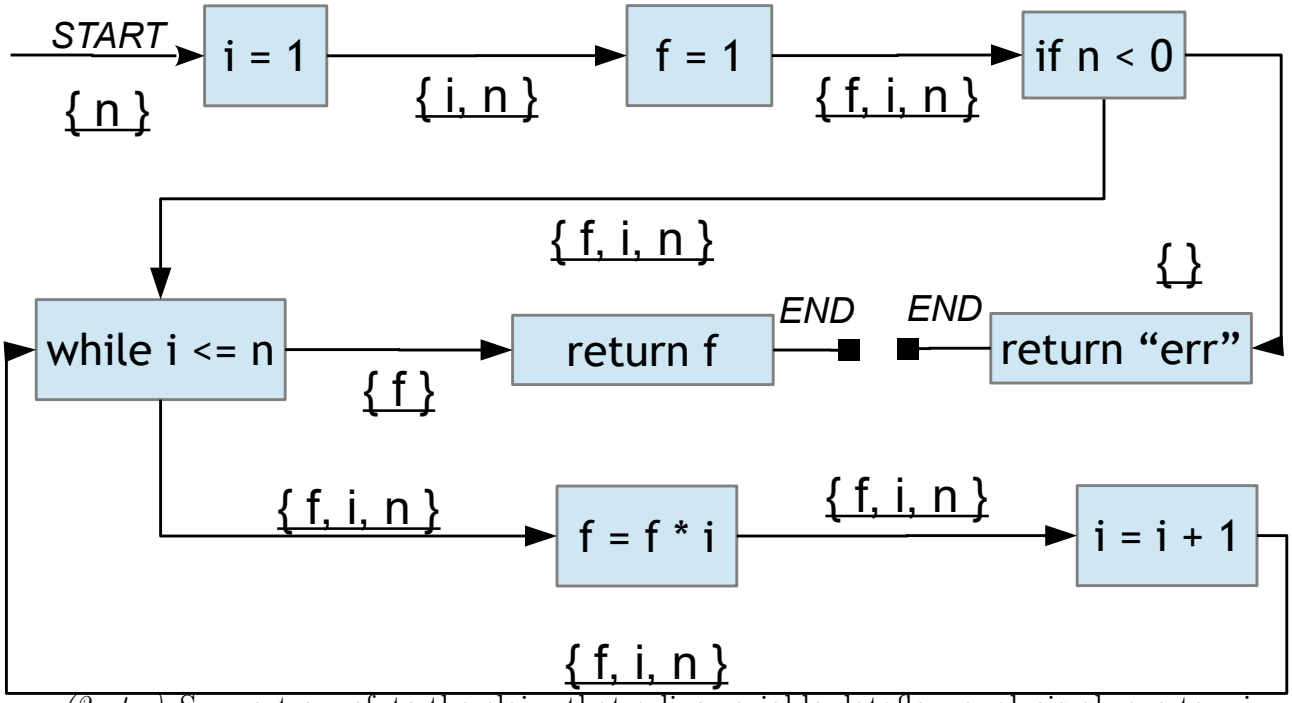
Almost certainly "refute". Mutations 1 and 3 would result in a single mutant that fails *all* tests. As in Homework 3, such mutants, sometimes called "trivial", do not have any differentiating or explanatory power: they require resources to run but do not tell you about the relative quality of your tests.

5 Dataflow Analysis (20 points)

Consider a *live variable* dataflow analysis for *three* variables, `i`, `f` and `n`. We associate with each variable a separate analysis fact: either the variable is possibly read on a later path before it is overwritten (live) or it is not (dead). We track the *set* of live variables at each point: for example, if `i` and `f` are alive but `n` is not, we write { `i, f` }. The special statement `return` reads, but does not write, its argument. (You must determine if this is a forward or backward analysis.)

(18 pts.) Complete this live variable dataflow analysis for `i`, `f` and `n` by filling in each blank *set* of live variables.

Note that this is the *factorial* method from the previous question! Live variable analysis is a *backward* analysis. One trick here is that there are two endpoints, but you can work backwards from either/both in any order and as long as you follow the rules you'll get the right answer. Another trick is that n is live on entry: it's a parameter to the function and we need it for the while loop guard. Note also that there are no dead assignments in this method.



(2 pts.) Support or refute the claim that a live variable dataflow analysis always terminates, even on programs that contain loops.

"Support" only. Following the slides (search for "Termination") and readings, each dataflow fact starts at some low value like # or {} and can only be increased. In this example, you can go from {} to {i} to {f,i} to {f,i,n} (etc.) but after at most three updates per edge (or k updates if there are k in-scope variables), there's nothing else to add to one set of live variables. With k variables this terminates in $\mathcal{O}(kE)$ time where E is the number of edges.

For further reading: formally terminiation requires that the transfer function be *monotonic* and that the lattice be *finite height*. We didn't use that terminology in class, but it's true here.

6 Quality Assurance Analyses (14 points)

(a) *(3 pts.)* Describe a situation under which the *Eraser* lock-set analysis might report a false positive race condition on a shared variable.

This is directly taken from the slide entitled "Eraser Lockset Example" (near slide 56; also Fig. 3 in the optional Eraser reading). If a shared variable is sometimes guarded by only Lock1 and sometimes guarded by only Lock2, the intersection will be empty and eraser will raise the alarm, even though the variable is actually always protected. Alternate answer: if you're not using locks but instead using some other exotic form of concurrency control (monitors? atomic test-and-set?), Eraser won't see/understand that and will alarm.

(b) *(4 pts.)* Support or refute the claim that the task of finding race conditions is more suited to dynamic analyses than to static analyses.

Likely "support". Two of the tools we studied in class, CHESS and Eraser, are both dynamic analyses. CHESS argues that a small number of scheduler interleavings suffices to find most concurrency bugs. Eraser piggybacks on your existing tests to check a simple rule. Dynamic analyses also typically have a very simple "bug report" format: you just show that dynamic run to developers to demonstrate the race condition. Static analyses for concurrency bugs are tricky because of the large number of scheduler interleavings: static analyses that try to consider them all typically have to approximate (cf. "*" in dataflow analysis) and end up giving many false positives or false negatives.

However, you could "refute". There are static type systems and syntactic sturctures meant to help with concurrency issues. Java's "synchronized" keyword is a (weak) example: you cannot forget to release a lock because it is released for you when you leave the curly-brace synchronized scope. You could also argue that dynamic analyses require you to have high-quality test inputs, which is a burden, so a static analysis that does not require those might be better. You'll likely have more thoughts on this after HW4, but HW4 is not in scope for this exam.

(c) *(3 pts.)* Support or refute the claim that information in a bug report is more likely to help guide an automated static analysis (rather than an automated dynamic one).

Likely "refute", but this is quite open-ended. The clearest argument is that bug reports sometimes include steps to reproduce — i.e., test inputs. You could then a dynamic analysis on that input. For example, if the user bug report says "I am getting weird values on input XYZ", you could run Eraser, an automated dynamic tool, on your program on inputs XYZ to see a race condition is involved. The bug report's guidance is the particular input, which both helps highlight the race condition and also saves time (you don't have to run your whole test suite or make up new inputs with AFL/EvoSuite, etc., just use that one input from the bug report).

However, you could also "support". In required readings like *Experiences Using Static Analysisto Find Bugs* and optional ones like *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*, we saw that developers really hate false

positives and reading through voluminous static analysis reports. We also saw that bug reports sometimes include information like stack traces. You could run a static analysis (like FindBugs) and filter out any warnings that are not about methods in the stack trace from the bug report. The guidance provided by the bug report is the set of methods it implicates: that helps reduce the static analysis human-reading burden.

Bug reports usually guide *manual* static analyses, like inspecting code, and *manual* dynamic analyses, like running code under a debugger. Those are both out of scope for this question!

Many well-reasoned answers are possible here, but high-quality answers will include true claims about bug reports linked to true claims about analyses.

(d) *(4 pts.)* Give three advantages of *abstraction* in analysis and one disadvantage of it.

Many answers are possible. Some exmaples:

Advantage: termination. In dataflow analysis, abstract information down to a few dataflow facts means that the analysis terminates even if the program has a loop.

Advantage: understandability. Eraser abstracts concurrency control down to the set of locks held. If you get a warning from Eraser, you know exactly what it means: variable x is accessed without being guarded by a consistent set of locks.

Advantage: reduce the state space / efficiency. Dataflow analyses often consider whether a pointer is null or not, or whether a string is tainted or not, rather than considering all 2^{32} (or whatever) possible values. This is slightly different from termination because "theoretically" 2^{32} would eventually terminate, just not in real life.

Advantage: handle "I don't know". Because most dataflow questions are undecideable, simple abstractions allow for principled behavior when an approximation must be made. The slides talk about *soundness* vs. *compelteness*: no false positives vs. no false negatives. This is related to termination and understandability, but is more about developer trust in the tool (e.g., you can trust it to have no false negatives).

Advantage: avoid human biases. In class we gave an example of a bug that was "hidden" by tabbing. By abstracting the program (e.g., to an abstract syntax tree), an analysis is not fooled by such coincidences of whitespace.

Disadvantage: introduces approximation. Because abstractions throw away information, they often come to incorrect conclusions. Dataflow analyses may miss correlated conditionals, Eraser will miss non-lockset concurrency control, and so on.

Disadvantage: requires creativity. The analysis designer has to think of the right abstraction. The wrong one might make the wrong tradeoff between analysis speed and analysis precision.

Disadvantage: must reason about updating abstract values. That is, you must come up with the "transfer functions" or the like. If you are tracking whether pointers are null or not, someone needs to write down that 'malloc' and 'new' return non-null values, just as someone needed to annotate how 'sanitize' updated values for secure information flow.

7 Extra Credit **(1 pt each; we are tough on reading questions)**

(Feedback) What is one thing you would change about this class for next year? What is one thing you like about this class?

(Feedback) What is one thing you would change about the department or the major?

(My Choice Psych) What's the difference between the *backfire effect* and *confirmation bias*?

From the *Quality Assurance and Testing* lecture. The backfire effect is where you "double down" on a false belief when presented with opposing information — but studies have found very little evidence for the backfire effect (it doesn't really happen). Confirmation bias is where you search for or interpret information to affirm your prior beliefs. It's a huge deal and actually happens in the real world.

Full credit typically required mentioning that one was "real" and one was "fake". Reasonable explanations that missed that most salient feature typically received half credit.

(My Choice Reading) In Petrović and Ivanković's *State of Mutation Testing at Google*, what did they do to make it easier to understand the results of the analysis?

From their abstract: "Furthermore, by reducing the number of mutants and carefully selecting only the most interesting ones we make it easier for humans to understand and evaluate the result of mutation analysis . . . We focus on a code-review based approach and consider the effects of surfacing mutation results on developer attention."

(Your Choice Reading 1) Identify any different **optional** reading. Write a sentence about it that convinces us that you read it critically. (Our subjective judgment applies here!).

(Your Choice Reading 2) Identify any different **optional** reading. Write a sentence about it that convinces us that you read it critically. (Our subjective judgment applies here!).