EECS 481 — Software Engineering
Fall 2019 — Exam #1 — **Answer Key**

- **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**.
- Please write your answers in the space provided on the exam. Clearly mark your solutions. 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, **you will receive one-third of the points for that small portion (rounded down) 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	O. Perverse Incentive	P. Priority
Q. Quality Property	R. Severity	S. Software Metric	T. Spiral Development
U. Threat to Validity	V. Triage	W. Watchpoint	X. Waterfall Model

A collaborative educational software company is working on a new on-line combined coursework and cuisine forum: *Pizza*.

U To assess the total market for a potential feature of this cuisine-and-coursework software, management only surveys chefs and forgets to survey students.

Q The customers require the software to be efficient and also to preserve user privacy.

T Software development is organized around the repeated generation of increasingly-complex prototypes, based on assessments of risk.

C Developers want to know how much time is spent in a particular function alone, as well as when all of its children are considered.

A During initial development, other groups within the company are asked to evaluate the software and look for bugs.

L Before the back-end database is fully implemented, code that depends on it is tested with respect to prototype functions that always return the same answers to every query.

D When evaluating the HTML output of an interaction, the "current time" field of the produced webpage is not considered.

H To help find race conditions, developers run the program and track the set of locks held during every memory access.

E To debug an issue inside a tight loop, developers arrange for execution to be interrupted only on the 10,000th iteration of that loop.

V A new defect report comes in. QA finds it to be a valid report and not a duplicate. Developers are instructed to address it.

P The new defect is a memory corruption issue. Management decides it must be fixed immediately.

N The patch for the bug is shown to two other developers who assess its quality.

O To favor high-quality code, management institutes a policy in which developer bonuses are awarded for fixing reported bugs. Developers intentionally create new bugs, report them, and fix them.

2 Testing and Coverage (17 points)

(4 pts.) Fill in the blanks in the code below with boolean expressions so that (1) all three parameters (`a b c`) appear at least once in the method body, and (2) any test suite that obtains 100% path coverage of feasible paths requires at least four test inputs.

```
1 void frances_allen(int a, int b, int c) {
2   if (a < b) STMT_1;
3   else       STMT_2;
4   if (a < c) STMT_3;
5   else       STMT_4;
6 }
```

Many similar answers are possible. There are four paths, so as long as the conditions are independent and feasible, you need four inputs for 100% path coverage. Example: 1 2 3, 2 1 3, 1 2 0, 2 1 0.

(4 pts.) Write a small method that accepts one parameter x and write a single test input for it such that your single test input maximizes statement coverage but not path coverage.

```
1 void foo(int x) {       // test input: x=6
2   if (x > 5) STMT_1;
3   STMT_2;
4 }
```

The test input x=6 covers both statements but only covers one of the two paths. Similar answers are possible.

(5 pts.) Write a small method that accepts one parameter x such that the test $\{ x = 3 \}$ has 50% branch coverage but the test $\{ x = 7 \}$ has 100% branch coverage.

```
1 void bar(int x) {
2   int i = 3;            // could be a "for" loop
3   while (i < x) {       // but written as "while" for clarity
4     i++;
5   }
6 }
```

There is only one branch: the condition on $i < x$. The test input x=3 takes the false branch, skipping the loop (50% coverage). The test input x=7 takes the true branch multiple times, then eventually takes the false branch *as well* as it leaves the loop (100% coverage).

(4 pts.) Consider a modified version of coverage-based fault localization that uses branch coverage counts instead of statement coverage counts as its mathematical basis. Support or refute the claim that it would be a better localizer.

Likely "refute". Coverage-based fault localization uses visit *counts* to find statements that are run on the failure inducing input but not on the good input. Consider a simple branching program like the before:

```
1 void foo(int x) {
```

```
2    if (x > 5) STMT_1;
3    STMT_2;
4  }
```

On the test inputs 1–8, the statement coverage is 3 and 8 while the branch coverage is 3 and 5. In many cases, you branch and statement coverage *counts* give very similar mathematical results or can be inter-derived from each other (even though branch coverage criteria, overall, are harder to satisfy), so we expect formulas based on them to be similar. Even though branch coverage is "better", branch coverage *counts* (which are used by fault localization) are usually not.

An argument could be made for "support", but would need to do more than simply asserting that branch coverage is more precise to get full credit. A short example or a concrete argument (e.g., about how the math works) would seem necessary.

3 Short Answer (18 points)

(a) *(4 pts.)* Choose an example of software development (from your own experience or from class) where "something went wrong" and use that example to illustrate the relationship between *uncertainty*, *risk* and *measurement*.

Many are possible. Consider the Equifax data breach detailed in the lectures. In it, management was likely *uncertain* about the potential cost of the vulnerability. Since *risk* includes both the likelihood of the bad thing happening and the cost if it does, management likely assumed that the likelihood was low, even if the cost was very high. *Measurement* can be used to reduce uncertainty and manage risk, but there is no evidence that they measured anything (from the patch quality to the properties of the vulnerability), and thus their decision not to deploy the path easlier was poor: "something went wrong".

(b) *(3 pts.)* Support or refute the claim that the number of *duplicate* defect reports for the same issue should influence its assessed *severity*. Describe a situation in which a defect report might be resolved without an associated code patch.

Likely "refute". *Severity* relates to the cost of not fixing the bug: the risk or damage in the real world. Risk is the frequency of incidents times the cost per incident. If there are many duplicate bug reports, it means many people are seeing the same bug, so the frequency is high. However, the cost could still be very low (e.g., everyone could be reporting the same unimportant spelling error).

However, "support" is possible. You could argue that user-facing bugs are the ones that matter most. The challenge here would be to add one more link: for example, to suggest that if the bug is important enough that many people are taking the time to report it, it must be worth time or money to them.

(c) *(3 pts.)* Support or refute the claim that newly-written xUnit-style (PyTest, unittest, etc.) *unit tests* should be required to meet a minimum software *readability metric* value to be checked in to a project.

Both are possible. The "support" argument would likely focus on the fact that a failed test must be inspected by developers (e.g., for triage or fault localization). Many companies already use style guidelines; if readability metrics are effective and we spend most of our time reading and doing testing, then requiring highly-readable tests may save effort. This is especially true for certain sectors, like Compiler Companies, where there are many tests available already.

However, "refute" would be my personal choice. First, available readability are descriptive rather than normative: they describe existing legacy code, but they can't be used to guide the construction of new code (cf. "just add 100 newlines to make it more readable"). Second, this may lead to perverse incentives where developers make tests that up the metric but don't actually assess correctness as well. Finally, you typically don't want to place any barrier on test creation. This is especially true in certain sectors, like new Startups, where there are not many tests available.

(d) *(4 pts.)* Identify a developer expectation of modern passaround *code review* that is commonly met. Identify a developer expectation of code review that is rarely met. Describe a buggy patch that code review is unlikely to correctly reject.

Following *Expectations, Outcomes, and Challenges Of Modern Code Review*, we see that developers expect "finding defects", "code improvement" and "alternative solutions". However, the big outcomes are "code improvement", "understanding", "social communication", and "defects".

So "finding defects" and "code improvement" are expectations of passaround code review that *are* met. And "alternate solutions" is an example of an expectation that is *rarely* met.

There are many types of buggy patches that human inspection may fail to warn about. Classic examples involve "invisible" control flow or variable changes, such as those caused by threads or exceptions. So a bug involving a race condition may be hard to spot manually. A bug involving failing to release a resource on an exceptional situation may be hard to spot manually. Finally, bugs like the "goto fail" example that "look like reasonable code" may be hard for humans to catch (but easy for automated analyses to catch).

(e) *(4 pts.)* Explain how automated whitebox *test input generation* uses *path predicates* and *constraint solving*. Describe two situations where test input generation is unlikely to succeed at producing high-coverage test inputs.

Automated whitebox test input generation enumerates each acyclic path through a method. It then gathers up all of the conditions (e.g., the guards to "if" statements) along that path: the conjunction of those conditions is the *path predicate*. When the path predicate is true, execution follows that particular path. *Constraint solving* is used to find values to variables that make the path predicate true. Those values then become the test input that covers that path.

Test input generation is unlikely to succeed if the constraint solving fails. While it is very good for small linear arithmetic problems, it may fail in other situations. For example, a condition that involves obtaining something from a network may not be solvable in terms of variables under your control. A condition involving nonlinear arithmetic (e.g., $x^2 = 25$) or string constraints (e.g., regular expression queries) may be difficult to solve. Even a very long method with 200 if statements would be hard to solve, since the conjunction of the 200 clauses per path might take the solver a long time.

4 Mutation Testing (14 points)

Consider the following implementation of duplicate array element detection and some associated test inputs, oracles and suites:

```
1  def duplicates(arr):
2    n = len(arr)                    # len([5,6]) == 2
3    answer = False
4    for i in range(n):              # range(3)   == [0,1,2]
5      for j in range(i+1,n):        # range(2,4) == [2,3]
6        if (arr[i] == arr[j]):
7          answer = True
8    return answer
9
10 testInput1 = []                   # oracle1 = False
11 testInput2 = [3,3]                # oracle2 = True
12 testSuiteA = [ testInput1, testInput2 ]
13 testInput3 = [4,5,6]              # oracle3 = False
14 testInput4 = [7,8,9,8]            # oracle4 = True
15 testSuiteB = [ testInput3, testInput4 ]
16 testSuiteC = [ testInput1, testInput2, testInput3, testInput4 ]
```

(6 pts.) Suppose the only mutation operator available to you is *Rename*, which renames one use of a variable to another. *Rename* requires 3 parameters, A, B, and C, and renames the first use of variable A on line B to C. For example, you might rename the reference to `answer` on line 3 to `i`. You may only mutate the method body (lines 2–8), not the tests.

Give an example of a single mutation such that *one of* `testSuiteA` or `testSuiteB` kills the mutant *but not both* (with no Python runtime errors, such as references to undefined variables or index out of bounds accesses) — and say which one kills the mutant.

Many answers are possible.

"Change j to i on line 6" is one example. The condition becomes `if (arr[i] == arr[i]):`, which is always true, so if the for-loops execute at all (i.e., if the input is non-empty), the procedure always returns true. This mutant returns False for T1 and True for T2 (both as before), so it is *not* killed by Test Suite A. However, it returns True for T3 and T4, which means it fails T3, so it *is* killed by Test Suite B.

(6 pts.) You are still limited to single variable renaming mutations. Give an example of a single mutation such that both `testSuiteA` and `testSuiteB` kill that mutant (i.e., that mutant fails at least one test in `testSuiteA` and also fails at least one test in `testSuiteB`) without any Python runtime errors.

Many answers are possible.

"Change *answer* to j on line 7" is one example. With this change, it is not possible to set answer to True, so the procedure always returns False. This fails Test 2, and thus Suite A. However, this also fails Test 4, and thus Suite B. So it fails both suites.

(2 pts.) Consider a single mutant that changes the == on line 6 to <. For how many tests

in `testSuiteC` does this mutant obtain the wrong answer? What is the mutation adequacy score of `testSuiteC`, expressed as a fraction, with respect to that single mutant?

With that mutation, if there are two elements in strict ascending order in the array, duplicates() returns True. So the modified version returns False for T1 but False for T2 — unlike the original, the mutant does not actually catch duplicate elements, they have to be less-than. It returns True for T3 ($4 < 5$) and True for T4 ($7 < 8$).

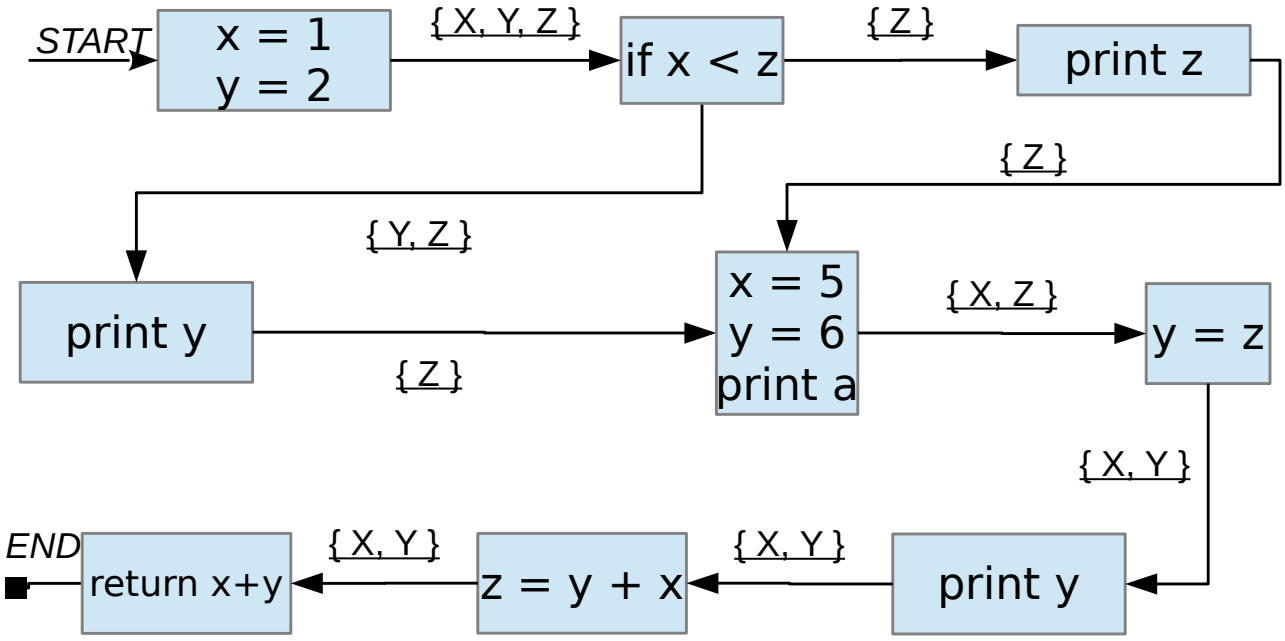
So it obtains the wrong answer on 2 tests (T2 and T3).

The mutation adequacy score is $1/1$. It is the fraction of mutants killed by the test suite. There is only one mutant, and it is killed by the test suite. (This is potentially tricky: $2/4$ is *not* the correct answer or formula here.)

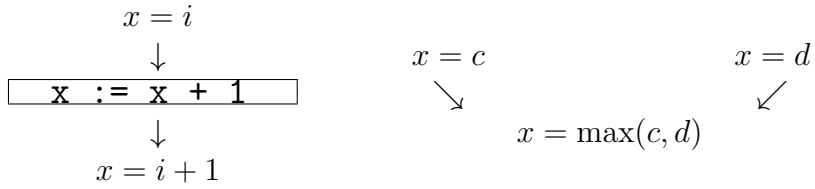
5 Dataflow Analysis (20 points)

Consider a *live variable* dataflow analysis for *three* variables, `x`, `y` and `z`. We associated 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 `y` and `z` are alive but `x` is not, we write { `y, z` }. The special function `print` reads, but does not write, its argument. (You must determine if this is a forward or backward analysis.)

(16 pts.) Complete this live variable dataflow analysis for `x`, `y` and `z` by filling in each blank *set* of live variables.



(4 pts.) Consider the following two *transfer functions* for the "definitely `null`" analysis (sometimes called "constant propagation"). One transfer function handles variable increments, the other handles two joining paths. Name two significant problems that would arise with the dataflow analysis algorithm if we adopted them.



Many bad things happen. First, the rule on the left never "settles down". If you imagine a simple "for (x=0; x < N; x++) ..." loop, the dataflow analysis would go around the loop "forever", updating the abstracted value of "x". That means that dataflow analysis would no longer be an *algorithm* because it does not terminate for all inputs. (That's particularly

bad because dataflow analysis needs to be efficient for use in optimizing compilers and other automated static analyses.)

Answers that get at the above point by suggesting there would be so many abstracted values (e.g., x = 0, 1, 2, 3, 4, ...) that you could update "x" to that you wouldn't actually be saving anything by abstraction are also correct, but must be phrased carefully.

However, the second rule can also get the *wrong answer*. Consider "if (false) x=4; else x=3;". The join rule will take the max of those two values and tell you that "x" is necessarily, always 4 after that "if". In fact, it is not. Other dataflow analyses use "*" (or somesuch) when they don't know the answer, but they are *conservative* and never report false things. This rule would cause us to report false things.

6 Quality Assurance Analyses (18 points)

(3 pts. each) Read the analysis task descriptions. For each task, choose three components of a good analysis solution, one from each column of the table. (For example, "a 1 p" is valid but not "a b 2" or "c 3".) If multiple combinations might fit, choose the *best* answer or the one corresponding to a tool or technique from class or the readings. You may use an option more than once across multiple questions.

a. Automated Dynamic Analysis	1. Enumerate	m. Abstracted Values
b. Automated Static Analysis	2. Replace	n. Common Subroutines
c. Manual Dynamic Analysis	3. Solve	o. Scheduler Interleavings
d. Manual Static Analysis	4. Track	p. Sets of Locks Held

- B 4 M. This is our definitely-null dataflow analysis. You wish to determine if a program variable `ptr` could ever possibly take on the value `null` on any run of the program that reaches line 56.

- A 4 P. This is the Eraser lockset dynamic analysis. You wish to determine if there are any shared variables for which a program's threads do use use a consistent concurrency control policy.

- A 2 N. This is the Chaos Monkey or Simian Army dynamic analysis. You wish to assess the availability of a service even if its dependencies or assumptions might fail.

- D 1 O. This is manual code inspection carrying out a CHESS-like activity. You wish to inspect a small program, for which you have no available inputs, to obtain a human's assurance that it handles critical sections correctly, regardless of the order of its thread operations.

(3 pts.) Support or refute the claim that an effective approach to finding performance defects (e.g., running slowly, using too much energy, etc.) would be to identify frequently-executed regions of code via sampling-based profiling and apply formal code inspection there.

Both are possible. For "running too slowly", this is a very good approach: profilers identify hot spots in the code, and inspecting them may reveal potential optimization or algorithmic changes. Typically human inspection is needed to spot algorithmic or Big-Oh changes, and formal code inspection is better at finding alternate algorithms than is code review (see elsewhere for expectations and outcomes of code review).

However, for "using too much energy", this is much less clear. It is not immediately obvious that that instructions that use the most time (and are thus found by the profiler) are also the ones that use the most energy. In practice, that is often true, but if you said that it might not be, you can argue for "refute". In practice, "support" is a little stronger.

(3 pts.) Support or refute the claim that false positives (i.e., false alarms) are more important than false negatives (i.e., missed bugs) when managers consider deploying bug-finding analyses in software engineering.

Both are possible. This relates to the *soundness* or *completeness* of analyses.

This is typically "support": a number of the required (*Ayewah et al.'s Experiences Using Static Analysis to Find Bugs [Google]* or *Parnin and Orso's Are Automated Debugging Techniques Actually Helping Programmers?*) and optional (*Bessey et al.'s A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*) readings covered exactly this point in great detail.

Loosely, after a certain number of false alarms, humans lose trust in the tool and stop paying attention to reported warnings. At that point, it doesn't matter if the tool is correct or not, because humans are not heeding it. In the usual case, humans prefer a sound analysis where everything the analysis warns you about is really a bug.

However, you could try to argue "refute" if you mention a particular domain that really cares about correctness. A common example from class the Aerospace industry, but other high-assurance domains (e.g., medical devices) also qualify. In those domains, the cost of failure (the cost of missing a bug) is so high that developers prefer a complete analysis, where every possible bug is reported, even if that means we spend some time on false alarms.

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?
(Free/Participation) Make up one "believable" (but possibly outlandish) claim about the class, professor or course staff. If we get enough interesting responses, we'll post them.
(My Choice Psych) Briefly describe the setup and outcome of Sherif's *Robber's Cave* psychology experiment studying *Realistic Conflict Theory*. (Hint: we played a related musical theatre song in class before discussing it.)

This is the experiment where otherwise-similar children were randomly assigned into one of two groups at a summer camp and the groups came into serious conflict on their own. Simplified, it can be used to argue that prejudice and fear of the other have innate components, and that we don't need someone to look different or have a different religion for conflict to arise.

(My Choice Reading) In Terrel et al.'s *Gender differences and bias in open source*, what high-level conclusion was drawn?

"Surprisingly, our results show that women's contributions tend to be accepted more often than men's. However, for contributors who are outsiders to a project and their gender is identifiable, men's acceptance rates are higher. Our results suggest that although women on GitHub may be more competent overall, bias against them exists nonetheless."

(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!).