EECS 481 — Software Engineering
Fall 2019 — Exam #2

- **Write your UM uniqname and UMID and your name on the exam.**
- There are ten (10) pages in this exam (including this one) and eight (7+1) 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 the backs of the exam pages as scratch paper. 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 Delta Debugging (22 points)

The Delta Debugging (DD) algorithm makes certain assumptions about its input. It requires that the interesting function be *monotonic*, *unambiguous* and *consistent*. Consider a set of of items $C = \{1, \ldots, 8\}$. In each of the following scenarios, we will provide the output of interesting on certain subsets of C and you must fill in the blanks for other subsets. Mark each other subset with Y for "must be yes", N for "must be no" and ? for "could be yes or no, it's unconstrained" such that the entire scenario meets DD's assumptions.

(3 pts.) Fill in the blanks for interesting:

interesting($\{3,4\}$)	=	Y	interesting($\{4,8\}$)	=	Y
interesting($\{8\}$)	=	N	interesting($\{4\}$)	=	____$Y(unambiguous)$____
interesting($\{3,8\}$)	=	____$N(unambiguous)$____	interesting($\{1,3,4\}$)	=	____$Y(monotonic)$____

(3 pts.) Fill in the blanks for interesting:

interesting($\{3,4\}$)	=	____$N(unambiguous)$____	interesting($\{4,8\}$)	=	Y
interesting($\{8\}$)	=	____$Y(unambiguous)$____	interesting($\{4\}$)	=	N
interesting($\{3,8\}$)	=	Y	interesting($\{1,3,4\}$)	=	____$N(unambiguous)$____

Your friend from Ypsilanti proposes an automated debugging algorithm, Ypsilon Debugging (after the Greek υ ypsilon instead of δ delta). Optimizing for the common case in which the final minimized set is small, Ypsilon Debugging (YD) starts by checking each singleton element in turn to see if it is interesting, then checks all pairs, then all triples, etc. It returns the first subset $c \subseteq C$ found with interesting(c) = Y. Fill in the blank or short response:

(1 pt.) ___ Y ___ Will YD find a minimal interesting subset ("Y" or "N")?

(2 pts.) ___ Y ___ Will YD find a one-minimal interesting subset ("Y" or "N")?

(1 pt.) $\mathcal{O}($___ N ___$)$ What is the Big-Oh running time of YD if a singleton is interesting ($|C| = n$)?

(2 pts.) $\mathcal{O}($___ 2^N ___$)$ What is the worst-case Big-Oh running time of YD ($|C| = n$)?

(2 pts.) ___ N ___ Does YD require monotonicity ("Y" or "N")?

(2 pts.) ___ N ___ Does YD require unambiguity ("Y" or "N")?

(2 pts.) ___ N ___ Does YD require consistency ("Y" or "N")?

(2 pts.) With respect to a quality property, describe a situation in which YD outperforms DD.

With respect to the quality property of run-time performance, if interesting can return Unknown but a singleton set is interesting then YD will take $\mathcal{O}(N)$ time while DD will take $\mathcal{O}(N^2)$ time.

(2 pts.) Describe a situation in which YD would be correct but *binary search* would not be.

If the smallest interesting set contains two elements, then binary search will not be correct (it can only return singletons) but YD will be.

2 Design Patterns (14 points)

Consider the following *incorrect* code for implementing a *Singleton* design pattern.

```python
 1 class Singleton:
 2   @staticmethod
 3   def get():
 4     return Singleton._instance
 5
 6   _instance = None
 7
 8   def __init__(self):
 9     self._state = 42
10
11   def current_state(self):
12     if Singleton._instance is None:
13       Singleton._instance = Singleton()
14     return self._state
15
16 def main():
17   print(Singleton.get().current_state())
```

(6 pts.) In at most four sentences, indicate the defect in this code and how you would fix it. Be specific.

The defect is that lines 12–13 should be moved to come just after line 3. Without this change, `get()` can return a `None` value, so line 17 (and similar legitimate uses) will fail.

(8 pts.) Consider a *Singleton* design pattern with a public "get" method and a private constructor. When using *inheritance*, which of those two can (or should) be overwritten by a subclass? Why or why not for each?

The get method should probably not be overwritten: its behavior is always to return (a reference to) the single "global" state, variable or object that is being represented by the Singleton pattern. If it were changed to return different objects with different state (rather than a single "global" one), then code that worked using the superclass would not work using the subclass.

However, the private constructor should probably be overwritten: if new fields were added to the Singletone-guarded "global" object, then the private constructor should be overridden and updated to also set those fields.

In general, however, it would be rare that you would combine the Singleton pattern with inheritance.

3 Requirements Elicitation (24 points)

(4 pts.) You are considering requirements for a bug-finding static analysis tool (like Infer or CodeSonar). Give an example of a useful *informal goal* for a quality requirement. Give an example of a *verifiable* quality requirement (be specific and detailed).

Example informal goals (quality). The tool should run quickly. The tool should have few false alarms. The tool should be easy to use.

Example verifiable requirements (quality). The tool should be able to analyze at least one thousand lines of code per second. The tool should have a false positive rate of at most 2 false alarms per 100 alarms raised. An employee with only two hours of standard training should be able to run the tool on a "Hello World" program.

(8 pts.) You are the customer asking a software company to build you a program that sorts a list of numbers. You both agree that the output of sort must be in ascending order. The company delivers a program that always returns the empty list. You meet to correct this. The company then delivers a program that deterministically tries all permutations of the input list until one is found that is sorted. (This is sometimes called "bogosort" or "permutation sort".) Explain elements that may have gone wrong during requirements elicitation (at any stage) and describe best practices to avoid such mistakes.

For the "empty list" issue, the functional requirements were not *complete* (see Slide 30 of the "Requirements and Specifications" lecture): they did not also specify that the output must be a permutation of the input. This might also be phrased as an *omission* (see Slide 44), which a feature was not stated in any requirements documentation item.

For the "bogosort" answer, the issue is that that algorithm takes far too long (e.g. $\mathcal{O}(N!)$ time). This suggests that quality requirements were inadequately established. One possibility is that they were described as an *informal goal* rather than as a verifiable requirement (e.g., "must not be too slow"). Another possibility is that performance requirements were *ambiguous*, allowing them to be interpreted in different ways — for example, performance might have been phrased in terms of a small, three-element example. On such an $N = 3$ example, $\mathcal{O}(N!)$ and $\mathcal{O}(N^2)$ may show similar performance. If the company thought that meant "must be fast on this small example" and the client thought that meant "must be fast on all examples including this one", it could explain the disparity.

Best practices for avoiding such issues include *good interview/validation skills* (the company should keep asking the client about functional requirements), *domain understanding* (the company should make sure that everyone agrees on domain concepts like "sorting"), *building a glossary* (to make sure that words like "sorted" and "fast" are understood uniformly), and *making prototypes to explore tradeoffs* (to show fast and slow versions of sorting, rarther than just delivering slow bogosort).

You could also argue that *identifying stakeholders* would help (maybe the wrong people were talking and didn't communicate correctness or speed) and *traceability* would help (maybe if we had a test case for how fast it has to be we wouldn't deliver bogosort), but those are a bit less relevant in this particular question.

(4 pts.) Support or refute the claim that *traceability* from requirements to tests is an important consideration when *designing for maintainability*.

Usually "support". Traceability from requirements to tests involves annotating tests with the particular requirements they help assess. For example, if the customer wants the software to be small and fast, Test 1 might be annotated to trace to the "small" requirement and Test 2 might be annotated to trace to the "fast" requirement. Design for maintainability involves recognizing that reading and changing code are dominant software engineering activities: if we can do more work at the start when something is written but save time later, that is usually a net win. Traceability does not necessarily help with readability. However, customers often change their minds and requirements shift. If the customer decides that "small" is no longer a requirement, we can simply remove Test 1 (based on its traceability annotation) rather than having to read through all of the tests to see which are now less relevant. So traceability took more time at the start (to add the annotation) but saved us time in mainteance later: so it is good for designing for maintainability.

(4 pts.) Using a situation from class or your own experience, give an example of a reasonable *strong conflict* in requirements. Then give an example of a reasonable *weak conflict* in the same situation.

In a strong conflict, statements are not satisfiable together. For example, we may desire that a bug-finding analysis (like Infer or CodeSonar) have no false positives and also have no false negatives (but that is impossible by the Halting Problem). We may also desire that a bug-finding analysis "never disclose security vulnerabilities to anyone" but also "reveal all vulnerabilities to the system administrator".

In a weak conflict, statements are not satisfiable together under some boundary condition. For example, "all bugs with high priority must be hidden" and "all bugs with priority 3 or less must be visible" is a conflict only depending on the value of "high".

(4 pts.) A customer cares about "analysis time" and "test suite quality". Your company understands "statement coverage", "branch coverage" and "mutation analysis". Explain *why and how* you would *explore alternatives* with the customer.

Why — we explore alternatives because customers will typically just say "Yes, I want that" to any quality question. In this example, the customer will want low analysis time and a high assessment of test sutie quality.

How — we explore alternatives by presenting mockups or prototypes of various tradeoffs. For example, statement coverage is the fastest but the least accurate. Branch coverage takes a little more time but is slightly more accurate. Mutation analysis takes forever but is more accurate.

4 Design and Maintainability (12 points)

(4 pts.) Compare and contrast *what* and *why* documentation content with respect to both usefulness for *maintenance* and also *tool support*.

What documentation describes the effect of a pieces of code (e.g., "these two loops sort the array"). Why documentation describes why that code was changed or why a design decision was made (e.g., "the customer wanted this" or "we want to avoid buffer overruns"). Both can be useful for general maintenance. However, tool support is significantly stronger for producing What documentation. This suggests that humans should focus on adding Why documentation; you can often recover What information by reading the code, but Why information has to be recorded.

(4 pts.) Consider an autograder used for class assignments. (For example, consider `autograder.io` evaluating HW3 submissions in this class.) How would you design it to support testing its non-interface (i.e., non-web, non-GUI) functionality?

One key approach to design for testing is to structure the code like a *library* or via a clean API (see Slide 40 of the lecture). With a focus on separation of concerns between the user-facing GUI and the grading logic, the grading logic can be subjected to unit tests more easily. This is facilitated by including *entry points* into the grading logic. Certain *design patterns*, such as Model View Controller (where the View is the GUI) can also help here.

Other answers, such as using design by contract or including Why documentation, are generally helpful but are unlikely to be full credit alone for this question.

(4 pts.) Consider an automated fuzz-testing input generation tool (like AFL) and an automated unit test generation tool (like EvoSuite). For each, support or refute the claim that it would be useful for quality assurance for a web-based autograder (including the web interface this time).

AFL — likely "refute". One of the challenges with test input generation is constraint solving (or coming up with random inputs). In class, we used reads from the network as a specific example of something that would be hard for tools to reason about (e.g., Slide 24 of the Inputs, Oracles and Generation lecture). AFL would generate end-to-end test inputs, which would have to include the HTTP request for the submission and the HW3 code itself – it is very unlikely to generate well-formed HTTP or Python. The optional reading "Automated Whitebox Fuzz Testing" mentions this on the first page, and any respons mentioning that sort of argument should be full credit here. In addition, AFL only makes inputs, not oracles. Basically, AFL will spend all of its time making the "invalid PNG" equivalent for HW submissions, which will not do a good job of testing grading logic beyond "you get zero points".

EvoSuite — likely "support". By contrast, EvoSuite does not make end-to-end tests. Instead, it generates tests for individual classes. Since many of those classes would be "inside" the autograder program (past the part where HTTP requests are required, for example), we expect EvoSuite-generated inputs to obtain higher coverage. In addition, AFL generates oracles in addition to inputs.

5 Interviews (10 points)

You are responsible for *giving* a non-behavioral technical interview to job candidates; you are the interviewer. Your company views technical interviews as an assessment of software engineering skills. The programming problem you ask of candidates is:

> Write CharCount(), a function that counts the number of occurrences of a character in a string.

The candidate's complete response is below. The first two commented lines indicate questions the candidate asked you.

```
/* Q: Can the string be empty? A: Yes. */
/* Q: Should I optimize for multiple repeated calls? A: No. */

int CharCount(String str) {
  // counting occurrence of character with loop
  int charCount = 0;
  for (int i=0; i<str.length(); i++) {
    if (str.charAt(i) == 'a') {
      charCount++;
    }
  }
  return charCount;
}

// test: "barbara liskov" -> 3
```

(2 pts.) Identify two thing that the candidate did well.

Does ask about a functional property (empty). Does ask about a non-functional property (caching performance). Does include an internal comment. Does use reasonable variable names. Does include a test.

(8 pts.) Identify and justify four significant things that the candidate did poorly.

No Why documentation is given. No indication is made of the running time. Only one test case is present, and it is very poor: no negative or corner case test are present. No discussion is made of design, maintainability or similar concerns. No evidence is given that the code is actually correct if the lists are empty (although inspection would reveal it).

A particularly egregious failure of "Design for Maintainability" and "Requirements Elicitation" here is that the candidate has hard-coded `'a'` as the only character to look for: you'd almost certainly want that to be a parameter!

6 Other Topics (18 points)

(4 pts.) Consider a dynamic memory analysis, such as taint tracking (to determine if data from an untrusted source ever reaches a trusted consumer) or memory leak detection (to determine if all allocated memory is eventually freed). Explain how a multi-language project involving both C++ and Python (with the associated "glue code" or "foreign function interface") complicates such an analysis. Be as specific as you can.

Most tools only support one language, as in Slides 67–68 of that Lecture. If the untrusted source come from Python (GUI code, for example) and the trusted consumer is in C (a database, for example), then a tool that only tracks Python values will not spot the bug.

Solving this issue is tricky. Multi-language project glue code must be careful to handle heap-allocated objects. In this problem, Python uses garbage collection but C++ does not. Glue functions like Py_BuildValue may complicate taint tracking (e.g., if they are passed tainted data along one path but not along another).

(6 pts.) Consider the following quotation reproduced in Cockburn and Williams *The Costs and Benefits of Pair Programming*:

> A system with multiple actors possesses greater potential for the generation of more diverse plans for at least three reasons: (1) the actors bring different prior experiences to the task; (2) they may have different access to task relevant information; (3) they stand in different relationships to the problem by virtue of their functional roles . . . An important consequence of the attempt to share goals and plans is that when they are in conflict, the programmers must overtly negotiate a shared course of action. In doing so, they explore a larger number of alternatives than a single programmer alone might do. This reduces the chances of selecting a bad plan.

Relate this quotation to *Agile Development* and its result that team diversity has a positive effect on software functionality. Then relate this quotation to *conflict resolution* in requirements elicitation.

In Agile development, studies have found that team diversity has a positive effect on software functionality. (By contrast, team autonomy results in software that is delivered on time, but that may lack certain functionality.) The first half of the quote offers an explanation for this observation: diversity correlates with different priori experiences, different information, and different relationships to the problem — and thus a diverse pair is able to generate more diverse plans, one of which is more likely to succeed.

Conflict resolution involves coming to a better understanding for terminology, designation, structure, weak, and strong conflicts identified in the requirements. To resolve weak and strong conflicts, negotiation and the exploration of quality tradeoffs are typically required (Slide 29).

That same approach — negotiation and the exploration of alternatives — is described in the quote above about Agile. In the pair programming set up, the two participants must agree on the goals and the plan. The goal aspect is directly comparable to RE: the goals are the requirements. Unlike RE, the Agile participants also negotiate over the plan (i.e., the implementation).

(4 pts.) Explain the relationship between coverage-based *fault localization* and *automated program repair* (APR). In that context, give one example of a bug that would be difficult for APR to fix and one example of a bug would be easy for APR to fix.

Automated program repair uses fault localization to reduce the search space. Even in a large program with millions of lines, coverage-based fault localization can often narrow a bug down to 10–100 suspicious lines. Because automated program repair typically runs with a time cutoff (or otherwise can't take too many machine hours if it is to be cost-competitive with humans), bugs that are well-localized are more likely to be repaired by random probes quickly.

As a result, APR is likely to succeed in all of the same situations where fault localization succeeds. A bug in which there is a crash on a certain conditional branch that is rarely executed will be easy to localize and thus easy to fix. By contrast, a bug in which the same lines are visited on the normal and buggy runs — such as cross-site scripting or SQL code injection — would be difficult to localize and thus difficult to fix.

(4 pts.) You are considering a medical imaging brain study experiment aimed at understanding code readability. Participants are placed in an fMRI scanner, shown pieces of code or prose, and given questions to answer about them. Participants are shown programs with different complexities (e.g., bubblesort vs. quicksort) as well as prose snippets with different complexities (e.g., elementary vs. high school reading level). The researchers contrast the code and prose conditions and conclude that a activity in a particular part of the brain is associated with assessing readable code but not with assessing unreadable code. With respect to *construct validity* and/or *controlled experimentation*, argue for or against believing the results of this study.

Likely "against believing". First, the study claims to study code readability. However, the researchers contrast the code and prose conditions, rather than high and low readability conditions. In addition, the study shows different levels of complexity, rather than different levels of readability. This could be viewed as a problem of construct validity: the study claims to be about code readability but it is really about code-vs.-prose or complex-vs.-simple. Another way to say this is that it is not a well-controlled experiment: the same condition should be shown to subjects with just one change (the readability). However, that is not done here as described.

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

(Feedback) In retrospect, what is one thing you enjoyed about the second half of this class? What should be changed about the second half of this class for next year?

Everything. :-)

(Feedback) What would you like to see changed about the discussion sections?

Nothing! :-)

(Your Choice Reading) Identify one of the readings from the second half of the class that we did not cover explicitly during the lecture. Write a sentence about it that convinces us that you read it critically. (Our subjective judgment applies here — sorry!).

(My Choice Reading) In "Experts bodies, experts minds: How physical and mental training shape the brain", what change in the brain "across each domain of expertise, . . . reflecting an increased level of expertise" was found?

We have seen, across each domain of expertise, that the degree of **functional brain plasticity** reflecting an increased level of expertise was task-dependent and followed extensive practice.

Students also mentioned facts like "the brains in London taxi drivers grew . . ." or "your metabolic efficiency increases with expertise . . .". Those are *true*, but they are not related to the quote the question asks about *from that particular paper*.

(Guest Lecture 1) In "Medium-Scale Software Engineering", give one point raised by Dr. Leach about industrial code management at Clinc.

Among others, Dr. Leach gave a MargeBot example, talked about code merges, showed a feature request, and a pipeline ("opt-in, precheck, test, coverage, build, qa_deploy").

(Guest Lecture 2) In "Accelerated Gnome Sequencing for Precision Medicine", what did Dr. Wadden say was the critical issue with hardware debugging at Sequal?

A lack of introspection. The long times required to run tests.