Navigation

Question 1. Word Bank Matching (1 point each, 16 points total)

For each statement below, input the letter of the term that is **best and most specifically** described. Note that you can click each cell to mark it off. Each word can only be used at most once.

Sometimes multiple words may appear to apply. Pick the best and most specific match. As one example, a hypothetical prompt about "running a program on an input and using a comparator to check the output against an oracle" logically matches both "testing" and "quality assurance". In such a case you would want to pick "testing", because "testing" is more specifically described.

A. — Code Review	B. — Delta Debugging	C. — Design Patterns	D. — Elicitation
E. — Fault Localization	F. — Informal Goal	G. — Maintainability	H. — Priority
I. — Productivity	J. — Profiling	K. — Readability	L. — Requirements
M. — Risk	N. — Severity	O. — Stakeholder	P. — Static Analysis
Q. — Triage	R. — Validation	S. — Watchpoint	T. — Weak Conflict

Q1.1: L		Detailed descriptions of what the software/program should do.
Q1.2: A		Before changed code is accepted, other developers must inspect the proposed change and its justification. This iterative process is mandated at most major software engineering companies.
Q1.3: Q		The team meets to discuss and assign a priority to each one of a list of reported defects.
Q1.4: R		You are reading the project specification for Project 2 in EECS 370. You think something seems wrong with the specification itself, so ask the course member about a possible mistake in the specification.
Q1.5: S		When debugging, you are interested in stopping the program's execution every time the global variable `global_request_hash` changes. Since that variable is assigned to in many potential places, you employ ...
Q1.6: N		The potential negative effect of a bug on the coding or running of a system, encompassing what happens if the bug is not fixed.
Q1.7: O		Product managers at Spotify meet with artists, record companies and lawyers to find out what new feature they want the most. What role do artists, record companies and lawyers play in this scenario?
Q1.8: K		This static quality property of software relates to its ability to be comprehended and thus maintained.
Q1.9: E		You are working on Euchre for EECS 280. You know there is a bug in your `player.cpp` file because you are failing a relevant test case. Now you are trying to figure out which specific lines are likely implicated in this bug.
Q1.10: J		You analyze your code for projects in 281, measuring the space and time complexity. You want to know the values of these quality properties so that you can optimize them.
Q1.11: F		A high-level notion in requirements elicitation that is communicated but is not precise enough to be measured.
Q1.12: T		You are working at a database company. In conversations, the client requests that "temporary tables must be deleted within five weeks" and then also requests that "temporary tables must be retained long enough for regulatory compliance".
		You are the product manager at TikTok. You are given a new task to work on a new feature on

Q1.13: D		the backend of the application. You begin by trying to better understand what it should do by communicating with the stakeholders.
Q1.14: C		These are best practices that are employed to solve commonly-recurring problems across multiple pieces of software. Common solutions often relate to object creation, structure or behavior.
Q1.15: P		An examination of the potential behaviors of a program without actually running it. May be manual or automatic and is often conservative.
Q1.16: M		You are working at Accenture at the start of the pandemic. It is possible that many of your developers may fall ill and may thus not be able to complete their tasks on time. You set up a meeting to discuss this possibility, its consequences, and its potential mitigations.

Question 2. Delta Debugging (21 points)

(a) (2 points each; 6 points total) Consider the three problems below. For each problem, specify if delta debugging can be used to solve the problem. If it can, provide a brief description of an "Interesting" function that will help solve the problem. If it cannot, specify which properties of delta debugging make it not suitable to solve the problem.

(i) (2 points) We have a list of distinct strings, and every letter of the alphabet is present at least once somewhere in at least one of the strings in the list. We want to find a one-minimal subset of this list such that every letter of the alphabet is still present at least once somewhere in at least one of the strings in the subset.

Your answer here.

ANSWER: Delta debugging is not suitable for this use case. Delta debugging requires an unambiguous Interesting() function, and checking for the presence of certain letters is an ambiguous problem. Consider running delta debugging on "the quick brown fox jumps over the lazy dog". The algorithm will output "quick brown fox jumps over lazy dog".

(ii) (2 points) In an effort to reduce the memory footprint of our C++ program, we have decided to replace all uses of `int` with `short`. However, when we do so, our program fails some of its test cases (we suspect due to integer overflow). We want to identify which `int` fields can be replaced with `short` fields such that the program still passes all tests.

Your answer here.

ANSWER: Support: Delta debugging is suitable for this use case. We can define script `is_interesting.sh` such that it takes a list of occurrences of `int` and replaces each with a `short`. The script exits 1 if the code compiles and runs the tests successfully, and it exits 0 if the code doesn't compile or fails any tests.

Reject: Because some operations (such as binary operators) only function correctly if all of the arguments are of the same type, delta debugging may find a one-minimal set but not a minimal set of fields that can be converted from `int` to `short`. As a concrete example, a loop like `int x; for (x=0;x<65537;x++) ...` will terminate if `x` is a 32-bit `int` but will loop forever based on integer overflow if `x` is a 16-bit `short` (since the largest 16-bit number is 65535, it will always satisfy the loop guard). This would result in a test case that neither passes nor fails, but instead loops forever, violating the consistency requirement.

(iii) (2 points) We have a list of distinct non-negative integers. We want to find a one-minimal subset of this list of integers such that the sum of the subset is greater than twenty.

Your answer here.

ANSWER: Delta debugging is not suitable for this use case. Delta debugging requires an unambiguous Interesting() function, and number summation is an ambiguous problem.

(b) We decide to design a new version of the delta debugging algorithm, shuffle debugging, which doesn't require any set intersections or unions. Instead, if neither the first half nor the second half of the set is interesting, shuffle debugging finds an irrelevant element (if one exists), performs a riffle shuffle of the two halves (in which the two halves are interleaved element by element, also called an "in shuffle"), and tries over again. If no irrelevant element exists, we conclude that we have a one-minimal set. We redesign the algorithm as described in the following Python code (note - make sure to read the full code if some is cut off):

Navigation


```
1
2  # Interleave lists A and B, return a new list containing
3  # alternating elements of A and B every other.
4  # Ex:
5  #   interleave([1, 2, 3], [4, 5, 6]) = [1, 4, 2, 5, 3, 6]
6  #   interleave([], [4, 5, 6]) = [4, 5, 6]
7  #   interleave([1, 2], [3, 4, 5, 6]) = [1, 3, 2, 4, 5, 6]
8  def interleave(A, B):
9      result = []
10     for i in range(max(len(A), len(B))):
11         if i < len(A):
12             result.append(A[i])
13         if i < len(B):
14             result.append(B[i])
15     return result
16
17 # Split a list in half and return (first_half, second_half).
18 # If the list isn't evenly divisible, the length of the first
19 # half will be smaller than that of the second half
20 def split(l):
21     first_half = l[:len(l)//2]
22     second_half = l[len(l)//2:]
23     return first_half, second_half
24
25 def SD(C):
26     if len(C) == 1:
27         return C
28     # P1 is the first half, and P2 is the second
29     P1, P2 = split(C)
30     if interesting(P1):
31         return SD(P1)
32     if interesting(P2):
33         return SD(P2)
34     for i in range(len(C)):
35         # C_prime is every element of C except the one at index i
36         C_prime = [x for j, x in enumerate(C) if j != i]
37         if interesting(C_prime):
38             # C_i isn't necessary, so remove it
39             P1_prime, P2_prime = split(C_prime)
40             shuffled = interleave(P1_prime, P2_prime)
41             return SD(shuffled)
42     # The set is already one-minimal
43     return C
44
45
```

We test out this new algorithm on the following example:
`Interesting(X) = [1, 9] is a subset of X`

(2 points each; 6 points total) For each of the following values of `C`, list how many calls to `Interesting` are made when `SD(C)` is called. If the algorithm never terminates, write `INCONCLUSIVE`.

(i) (2 points) `C = [1, 2, 5, 0, 4, 3]`

Your answer here.

ANSWER: 8

(ii) (2 points) `C = [1, 4, 3, 2, 5, 0, 9]`

Your answer here.

ANSWER: 24

(iii) (2 points) `C = [1, 9, 5, 3, 4, 2, 0]`

Your answer here.

ANSWER: 10

(c) (3 points each; 9 points total) In order for shuffle debugging (SD) to find a one-minimal subset, some of the requirements of delta debugging may or may not be necessary for shuffle debugging. For each of the following requirements, indicate whether violating the requirement could result in SD failing to return a one-minimal subset ("necessary") or whether SD can always

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

operate correctly on inputs that do not meet that requirement ("optional"). If a requirement is necessary for shuffle debugging, give an example where not having it results in an incorrect answer. If a requirement is optional for shuffle debugging, give an example where delta debugging would give the wrong answer but shuffle debugging would give the right answer.

(i) (3 points) Monotonicity?

Your answer here.

ANSWER: For each of these three problems, we gave 0 points for the wrong answer (regardless of associated wrong explanation), 1 point for leaving it blank (as per the test rules), 1.5 points for giving half the answer (such as just saying it was necessary without giving the example), 2 points for the correct answer but an incomplete or murky explanation, and 3 points for the correct answer with a solid explanation.
Shuffle debugging does require monotonicity. An example of a non-monotonic problem that SD would get wrong is as follows. Suppose the only interesting set is exactly [1,9] --- bigger sets, like [1,2,9], are not interesting. Running SD on an input like [1,9,0,2,5,3,4] should return [1,9], but does not.

(ii) (3 points) Unambiguity?

Your answer here.

ANSWER: Shuffle debugging does **not** require unambiguity. Informally, the reason SD can get away with not requiring unambiguity compared to DD is the for loop at the end of SD: it does expensive work to try all one-removal combinations. That means that SD is slower than DD, but it also means that SD works in ambiguous situations.
An example of an ambiguous situation where SD works but DD fails is that a set is interesting if it contains both X and -X. So [3,1,-3] is interesting, and so is [1,5,-5]. However, their intersection, [1], is not interesting. SD correctly finds one-minimal interesting subsets in this case, while DD does not.
A potential common mistake is considering the situation where a set is interesting if its elements sum to zero: [1,2,-3] is interesting, for example. That definition is interesting is ambiguous, but SD fails to work correctly for it. The reason is that that definition of interesting is also non-monotonic, and monotonicity is required for SD. So such an example is not the right one to use when reasoning about SD and ambiguity, because it also brings in monotonicity.

(iii) (3 points) Consistency?

Your answer here.

ANSWER: Shuffle debugging does require consistency. While the types are not always written out explicitly in Python, SD is treating interesting as boolean that is used to drive "if" statements in Python. An inconsistent definition of interesting might return True, False or Unknown — or might not return at all. SD does not have handle for any such case. (To handle inconsistency, you might imagine writing an algorithm where a switch/case construct is used to consider three return values from interesting, but nothing like that is given here.)

Question 3. Short Answer (18 points)

Provide an answer to each of the questions below.

(a) (3 points) In 3 sentences or less, describe and explain three things that we can do to reinforce the readability of our code.

Your answer here.

ANSWER: A strong answer would address points such as: 1) Include comments, 2) Avoid long lines, 3) Avoid having many different identifiers, 4) Fully blank lines may matter more than indentation. A strong answer could also use evidence from the medical imaging brain studies presented, such as mentioning 5) beacons or 6) semantic cues. Students could also bring in notions such as "requiring readability badges for code review", but more finesse would be required since that notion is more directly about improving code review and the question asks about reinforcing readability.

(b) (3 points) In 4 sentences or less, describe and explain the tradeoffs/differences between working in academia and working in industry according to Dr. Ciera Jaspan and/or Dr. Kevin Leach.

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

Your answer here.

ANSWER: A correct answer might address: 1) Academia: freedom to do research in a topic of your choice, but you must ask agencies for funding, 2) Industry: can't always pursue topic that interest you, but you know who's providing the funding have ability to communicate directly. In industry it is often possible to work with real developers on important problems directly, but at the same time any problem you pursue must help the company's bottom line.

(c) (3 points) In 2 sentences or less, describe and explain two major benefits of using Medical Imaging to gather information about software engineering processes.

Your answer here.

ANSWER: The correct answer should address points such as: 1) Unreliable self-reporting, 2) Inform Pedagogy, 3) Understand Expertise, 4) Retrain Aging Engineers, 5) Guide Technology Transfer. We also accept answers that mention 6) Fundamental Understanding, even though it is arguably not directly related to SE. An example of an inadequate answer would be suggesting that medical imaging is 'more precise' or 'allows you to look at humans' or the like. Those alone do not fully complete the logical link to software engineering.

(d) (3 points) In 3 sentences or less, describe two key differences between experts and novices with respect to problem solving and productivity.

Your answer here.

ANSWER: An answer might mention that experts and novices cluster problems differently, that novices make more mistakes, that novices take longer, that the structure of the brain changes over time as you learn more about a topic, or that experts are more efficient (in a physical sense regarding the brain) than novices.

(e) (3 points) In 1 sentence, give an example of a quality requirement. In 2 sentences or less, describe and explain the connections between the environment and the machine.

Your answer here.

ANSWER: A correct "quality requirement" (or non-functional requirement) example might be within one of the following categories: 1) Confidentiality requirement, 2) Privacy requirement, 3) Integrity requirement, 4) Availability requirement, 4) Reliability requirement, 4) Accuracy requirement, 5) Performance requirement.
For the second part, the Requirements and Specifications slideset begins covering the environment and the machine around slide #12. The environment domain includes the real world (e.g., how fast is the car going?) while the machine domain includes software and variable (e.g., this variable should track how fast the car is going). Input and output devices form the interface between them, and system and software requirements often state relationships between them (see Slide #15).

(f) (3 points) Consider a game of Nim with four piles: {A,A,A,A,A}, {B,B}, {C,C,C}, {D,D,D,D,D,D,D} (i.e., 5 2 3 7). It is your turn. What move would you take to win?

Your answer here.

ANSWER: The current board has Nim-sum (XOR) value 3: (4+1) (2) (2+1) (4+2+1). Your goal is to remove multiple pieces from a single pile such that your opponent is left facing a board with value 0. Answers such as "take all of C" reduce the value of the board to 0. Multiple solutions are possible: another possible answer would be "take 3 from D".

Question 4. Fault Localization (14 points total)

Below is a buggy snippet from a Python program that determines which of the 3 values passed in is the largest (and returns 0 if they are all equivalent):

```
          ...
1   class differentValues(object):
2     def greatestValue(self, a, b, c):
3       if (a > b || a > c):
4         return a
5       else if (b > a || b > c):
```

```
6         return b
7       else if (c > a && c > b):
8         return c
9       return 0
           ...
```

Use the standard Tarantula Fault-Localization Suspiciousness Ranking for necessary calculations:

```
Sus(s) = (failed(s) / totalfailed) / (failed(s) / totalfailed + passed(s) / totalpassed)
```

... where totalpassed is the number of passing test cases and passed(s) is the number of those that executed line s (similarly for totalfailed and failed(s)). Note that this is the same formula presented in the Lecture: there is no "trick" in the formula.

(a) (6 points) Give two sets of inputs that cause line 5 to have the highest suspiciousness ranking overall. The first set should pass (i.e., the program obtains the correct answer) and the second set should fail (i.e., the current program obtains the incorrect answer). Use only the values 1, 2 and 3 (in any combination, with repeats if you like) in your answer. Format your answer with the values as triplets by commas and place one answer on each line, as in:

```
(4,5,6)
(7,8,9)
```

Your answer here.

ANSWER: One potential answer is: (3, 2, 1) passing and (1, 2, 3) failing.
Note that in some cases there was some confusion about whether the first set should pass and the second should fail, etc., and thus in some cases partial credit was given.

(b) (4 points) Choose either **library-oriented architectures** (one approach to designing for maintainability) or **multi-language projects** (focusing on the glue code that interfaces between two languages). Identify your choice, and then either **support** or **refute** the claim that Tarantula-style fault localization would be effective at localizing defects in that context. Use at most four sentences.

Your answer here.

ANSWER: For multi-language project glue code, the answer is likely "refute". In glue code, as in the examples shown in class, bugs typically relate to particular data values and how they are converted. By contrast, the same lines are visited each time, so a fault localization approach like Tarantula would assign all of the glue code lines the same suspiciousness. As a concrete example, consider glue code that forgets to register memory with the garbage collector or forgets to free it: the crash or memory leak will not happen immediately on that line (indeed, the problem may be a missing function call that has no associated line), and thus standard fault localization will not be revealing.
For library-oriented architectures (introduced on Slide #41 of the Design for Maintainability slideset), the answer is likely "support". Library-oriented architectures make it easier to test your code; they shorten the distance between the start of the program and your code and allow elements of your code to be tested independently (e.g., your data structure can be tested as a library or service without having to run the whole GUI). Because Tarantula-style fault localization is a dynamic analysis and depends critically on the number and quality of the test cases (e.g., for its mathematics), having more tests that reach your code mean that it will be more likely to help pinpoint suspicious aspects, all other things being equal.

(c) (4 points) Give an example of a security defect, attack or exploit for which we would expect Tarantula-style fault localization to do a good job of pinpointing the right line to change. Then given an example of a security defect, attack or exploit for which we would expect it to do a poor job. Use at most four sentences total (e.g., example, explanation, example, explanation).

Your answer here.

ANSWER: Consider a null pointer dereference. It is a security vulnerability if an attacker can crash your server on deman (sometimes called a "denial of service attack"). Tarantula is likely to be very good at pinpointing the right line to change: the program crashes directly at the relevant line when given the buggy (attack) input, but does not crash on normal test inputs. Lines after the crash are not visited on the buggy input, so Tarantula will be able to make distinctions, etc.
By contrast, consider cross-site scripting, SQL code injection, or even spam detection. All of those security issues relate to the values of strings. Tarantula does not look inside strings or other data values at all. Instead, it only looks at lines visited. However, for those security issues, the lines visited are all the same. In SQL code injection, you always execute the line that places the user input in the database. The only difference is whether the user input string contains "DROP TABLE" or

the like or not (see https://xkcd.com/327/ , etc.). So the fault localization math will give the same suspiciousness value to all of the lines, because they are all visited on both good and bad runs, and thus the fault localization will not be helpful.

Navigation

Question 5. Design Patterns (17 points total)

Consider the following C++ code that will serialize the value of one particular node in a Binary Search Tree (BST). This code snippet is only relevant for parts 5a, 5b and 5c. You may assume the following:

- `parseArgs` will correctly parse any given arguments. Arguments may include a pointer to the root node of the tree as well as the node to find
- All nodes may have a unique integer value (i.e., `node->val`)
- Aside from `serializeNode`, `parseArgs`, and `serializeIntHelper`, you may assume there are no other functions and interfaces
- All functions not defined here are implemented correctly

```
1
2  string serializeNode(void* args) {
3      auto argsPar = parseArgs(args);
4      Node* root = argsPar["root"];
5      Node* nodeToFind = argsPar["input"];
6
7      if (root == nullptr) {
8          throw runtime_error("Error");
9      }
10
11     // Perform Breadth-First Search
12     queue<Node*> bfs;
13     bool nodeFound = false;
14     int nodeVal = -1;
15     bfs.push_back(root);
16
17     while (!bfs.empty()) {
18         Node* curr ;
19
20         // Loop invariant is true here
21
22         curr = bfs.front();
23         bfs.pop_front();
24
25         if (curr == nodeToFind) {
26             nodeFound = true;
27             nodeVal = curr->val;
28             goto nodeFound;
29         }
30
31         if (curr->left != nullptr) {
32             bfs.push_back(curr->left);
33         }
34
35         if (curr->right != nullptr) {
36             bfs.push_back(curr->right);
37         }
38     }
39
40 nodeFound:
41
42     if (!nodeFound) {
43         throw runtime_error("Error");
44     } else {
45         // Serialize the node
46         string serializedVal = serializeIntHelper(nodeVal);
47         return serializedVal;
48     }
49 }
50
51
```

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

5a. (2 points) List one invariant of the function. The invariant should be true inside the loop at the line indicated by the comment. Do not restate any of the assumptions listed above. Do not list a predicate that is trivial or true for all programs (e.g., 1+1=2).

Your answer here.

ANSWER: An invariant is a predicate or formula that is always true on every execution of a particular line. When the code reaches the line specified inside the loop, we know !bfs.empty(), otherwise we would not enter the loop. However, we also know that nodeFound == false, because after it is set to true the code immediately jumps out of the loop (with a goto, sigh). For similar reasons, we know nodeVal == -1. Finally, we also know that root is not null because of the check at the start of the function (although this is not necessary to mention since it, arguably, relates to the input assumptions, since it comes from parsing the arguments). An example of an invariant is thus `root != nullptr && !bfs.empty() && !nodeFound && nodeVal == -1`.

5b. (2 points) List one post-condition of the function. The post-condition should be true as the function terminates (either via a `return` or an uncaught exception). Do not restate any of the assumptions listed above. Do not list a predicate that is trivial or true for all programs (e.g., 1+1=2).

Your answer here.

ANSWER: A post-condition is a predicate that is true after a function returns. The complicating factor here is that this function can return multiple ways. Sometimes a variable like "retval" is used to refer to the return value, as on Slide #44 of the Test Inputs, Oracles and Generation lecture. This procedure can return one of two: either it throws a runtime error or it returns serializedVal, which is a string holding the serialized representation of the requested node's value. First, a full credit answer must indicate that both outcomes are possible, either textually or by using a logical **or**. Second, a full credit answer should mention or link the success-case return value to nodeToFind (since that is the key correctness condition of the method). For example: `(retval == runtime_error("Error")) || (retval == serializeIntHelper(nodeToFind->val))`

5c. (3 points) Describe why duplicate code can have a negative impact on software maintenance. Describe how you might refactor the original code to improve maintainability. Use at most four sentences total.

Your answer here.

ANSWER: Duplicate code (sometimes called "code clones") can have multiple negative impacts on software maintenance. First, duplicate code takes up more lines. Students might mention that Code Inspection and similar maintenance activities can only cover so many lines of code per hour. Alternately, students might mention that Readability (or Complexity metrics) would be negatively impacted, since readability metrics (and Complexity ones) tend to correlate with code size. Alternately, students might mention that fault localization would be complicated, since there are now multiple lines that are really "the same" but might get different suspiciousness values or even simply add to the list of suspiciousness values (and the "Are Automated Debugging Techniques Actually Helping Programmers?" reading notes that human developers stop reading those lists if they are too long). Alternately, students might mention that a bug found in one place now also has to be fixed in all of the duplicate places.
We would refactor the code to improve maintainability by turning the duplicate code into a procedure that is defined once and called multiple times. One example of this is duplicated code related to decisions about object creation. This scenario is explicitly described starting on Slide #19 of the Patterns and Anti-Patterns lecture. A full credit answer could either describe this in words (e.g., abstracting the duplicated code into a procedure) or could mention the use of a relevant design pattern.

5d. (6 points) Consider **Singleton**, **Template Method**, and **Strategy** design patterns. For each design pattern, provide two specific examples of programs that may utilize the design pattern and explain why such a program benefits from this design pattern. Use at most four sentences per design pattern (context, benefit, context, benefit).

Your answer here.

ANSWER: Student received different random combinations of prompts for this question. The answer key gives some high level points about each one.
Observer: The observer (or publish-subscribe) pattern helps in any situations where multiple objects might care about changes to another object. For example, many components of system might want to observe the "battery" module for "suspend" or "shutdown" events to save critical data. In a word processor like Google Docs, the spell-checker, scroll-bar size calculator, page-number calculator and the like may all want to observe when new characters are entered. The benefit is that all of the listeners get updated when the state of the subject changes.
Singleton: The singleton pattern helps guard access when there is conceptually only one logical instance of a resource. For example, in a cellphone there may only be one GPS module, or in a three-tier web application, there may only be one database connection. The pattern is helpful when you want more control than a "naked" global variable. For example, you may want many places in the code to be able to read from the database connection, but not shut it down.

Template Method: The template method helps when you want to follow the same high-level plan but select lower-level implementations for sub-steps at run-time. A sales app that always receives money, keeps an internal tally, and delivers an item might use the template method pattern if there are multiple ways to "receive money" (e.g., via credit card, via direct deposit, etc.). Part of the advantage is the inversion of control: the subclasses redefine certain steps of the algorithm. (This can also make it tricky to understand!)
Factory: The factory method pattern provides a way to create objects while hiding the exact subclass created. This is a good fit almost anywhere you have subclasses of a superclass and you might add more subclasses as time goes by. For example, an image viewer might want to use an abstract Image class and add support for JPG, PNG, TIFF, etc., over time, without requiring client code to know the names of those subclasses. Factory patterns can also encapsulate dynamic logic around object creation and can help control access to the state of created objects (but not revealing types or fields).
Strategy: The strategy pattern allows an algorithm to be selected at run-time without code duplication. For example, a sales application might have an algorithm for state sales tax, an algorithm for states with no sales tax, an algorithm for international taxes, and so on. The purchase location is not known until runtime, at which point the sub-algorithm is selected. Strategy patterns also all algorithms to be separated from data, admitting reuse (as in the sorting algorithm example, where you pass a comparison function in to "sort").
Creational: Creational design patterns include the named constructor pattern, the factory pattern, the singleton pattern, and so on. Without duplicating an answer above, a student receiving this prompt had more free choice. Suppose the named constructor pattern is chosen. It allows object creation to be guarded by program logic. A normal constructor can be invokved at any point, but a named constructor may apply other logic (e.g., checking available resources, permissions, etc.) before invoking the "real" constructor.

5e. (2 points) Suppose that a particular software engineering project spends 39% of its lifetime effort on implementation, 39% of its lifetime effort on testing, and 22% of its lifetime effort on other non-testing maintenance. You are considering a design that would (a) increase the effort required for implementation by 14% (for example, if implementation previously took 10 hours, but that effort is increased by 35%, it would now take 13.5 hours); but that would also (b) reduce the effort required for testing by 14%. Assume the project originally required 100 effort units to complete over its lifetime: calculate the new lifetime effort units required by the project with your new proposed design.

Your answer here.

ANSWER: 100.0

5f. (2 points) In four sentences or fewer, reference the paper *The art of software systems development: Reliability, Availability, Maintainability, Performance (RAMP)* and other knowledge to support or refute the following claim: We **always** want to spend more on design to *reduce* maintenance costs. You should include at least two pieces of evidence or argument, at least one of which should be associated with the paper. Use at most four sentences.

Your answer here.

ANSWER: The answer here will depend on the variant of the question: exams might have used "always", "sometimes", or "never". The typical answer is **support** in the positive direction: a recurring theme in software engineering is that we can spend more up-front effort on design to reduce maintenance costs. This is explicitly covered starting on Slide #3 of the Design for Maintainability lecture. However, students might bring up other evidence, such as the chart, first introduced on Slide #18 of the Process, Risk and Scheduling lecture, that mentions how defects cost more to fix as the time increases between when they are introduced and when they are found.
Although many examples from the RAMP paper can be used, this is an example of a question that focuses on critical reading comprehension and retention: much of the RAMP paper does not directly address this issue per se, and thus students who did not recall the paper or the concepts may have struggled to find something relevant in time while taking the exam. One potential answer would be to take the paper at a very high level: the authors argue strongly in favor of a particular framework ("In order for this art to be more effective and controllable, a peformability framework, which combines the four non-functional requirements (performance, reliability, availability, and maintainability), is proposed") and say that it directly includes maintainability. However, students could also have pointed to particular pieces of evidence from the case studies. For example, near "The example of InterMod60 is a clear example where the restructuring of the intermodulation algorithms", the restructuring (or not) is a design-related decision (i.e., how should we design the software? what structure should it use?). Ditto for "The use of time-synchronized processes would have been the preferred choice for achieving higher throughput", in which the use of a certain process type is a design decision. Those specific examples require a bit more finesse (since they speak more directly about availability or throughput or performability, and so the student would have to link that to maintainability), but can be made full credit.

Question 6. Interviews (14 points total)

A candidate at a technical interview is given the following prompt:

Write `isAnagram()`, a function that returns whether two strings are anagrams or not. String A and string B are anagrams if A's characters can be rearranged to form string B.

Below, the candidate provides an implementation for isAnagram():

```
bool isAnagram(string A, string B) {
    // creates a size 26 array filled with zeroes
    int charCounter[26];

    for (char letter : A) {
        charCounter[letter - 'a']++;
    }

    for (char letter : B) {
        --charCounter[letter - 'a'];
        if (charCounter[letter - 'a'] < 0) {
            return false;
        }
    }

    for (int count : charCounter) {
        if (count != 0) {
            return false;
        }
    }

    return true;

}


```

6a. (1 points) Identify a test input that would return true for the above implementation.

Your answer here.

ANSWER: An input such as "abc", "cba" would return true.
Some students were tempted to suggest that the code does not run at all and thus no tests return true. However, part (c) below implies that the code is meant to run, many students asked the course staff on the forum and received guidance, interviews are interactive, and the exact language used is not specified. Limiting your answer to a statement that the code does not run at all typically merits partial credit but does not give you a chance to demonstrate your mastery of the material for full credit.

6b. (1 points) Identify a test input that would return false for the above implementation.

Your answer here.

ANSWER: An input such as "abc", "def" would return false.

6c. (4 points) The interviewer runs the implementation on a set of test cases. 6 test cases pass, while 4 fail. The interviewer suggests that perhaps the candidate made some assumptions that resulted in an erroneous implementation. List two questions that this candidate could have asked such that each question would have revealed a separate mistaken assumption. (For example, candidates implementing integer division might avoid a divide-by-zero error by asking "can the denominator be zero?".)

Your answer here.

ANSWER: A key issue with the provided implementation is that it only handles lower-case letters. It also does not handle any non-letter characters. Questions such as "can the strings contain spaces?" or "can the strings contain uppercase letters?" or "is internationalization a concern?" would all reveal these issues. Depending on the string type in this language, the provided implementation may also assume that the inputs are not null: it does not check for, or handle null inputs, gracefully. A question such as "could the strings be null?" would address this concern (note that null and empty are not the same in this context).

6d. (4 points) Assume that the company makes hiring decisions solely based on this programming interview task: candidates are evaluated based on the questions they ask during the interview and the performance of their submitted code on test cases. Identify and explain two flaws in this hiring process with respect to the most common company hiring goals (e.g., not hiring people who are likely to do a poor job, hiring people who are likely to do a good job, etc.).

Your answer here.

ANSWER: One issue is that the process does not assess many relevant software engineering job tasks, such as a writing comments, comprehending code, or maintaining code. (For example, no mention is made of assessing the readability of candidate's source code. Similarly, the candidate is not given code written by someone else and asked to fix it.)
Another issues is that the process does not include any behavioral aspects. Candidates who do not fit with the corporate culture or are not pleasant to talk to are unlikely to do a good job at the collaborative aspects of software engineering. (For example, no mention is made of asking the candidate to describe a situation with a communication failure and how it was resolved.)
It may be tempting to answer that the process would miss good candidates, but care must be taken with such an answer: modern companies are much more interested in ruling out poor candidates (by their estimation) than in making sure that all good candidates are hired, and student answers that mention failing to hire good candidates without mentioning this asymmetry would not receive full credit.

6e. (4 points) Support or refute the claim that *automated program repair* would likely repair the defect(s) in the code listed above, assuming a test suite with 100% line coverage that contains at least one test the code currently passes and one test the code currently fails.

Your answer here.

ANSWER: Automated program repair is **unlikely** to repair the defects in the code listed above. There are at least two critical aspects to automated program repair here: fault localization (can it determine where to make the change?) and mutation (what sort of edits will it try to make the change?). Suppose the bug is that the code should handle all 256 character values and a buggy input shows that by using uppercase letters in string A. Fault localization for this problem is arguable. Students could suggest that as soon as the uppercase letter is encountered, the program will crash immediately in the first for loop, implicating the key line. However, charCounter is a stack-allocated array, and in practice walking off the end of a stack-allocated array typically does not immediately crash the problem and instead insidiously corrupts memory (indeed, this is how most buffer overruns work!). So students could argue that either way. However, the mutation operator (or edit operator) aspect is much more significant. Current automated program repair approaches insert, delete or swap statements or expressions. In some cases, like Facebook's SapFix, they may also use particular templates (e.g., inserting null checks to deal with null pointer exceptions). The automated program repair techniques discussed in class or in the readings or commonly deployed do not support changing or making new data structures (or new function etc.). As a result, no mutation considered will fix the bug (since fixing the bug either requires allocating more memory for the charCounter array or using a different data structure or algorithm). Even an APR algorithm that swaps constant numerical expressions would be unlikely to succeed, since 256 (or larger) is not present in the program text to be used to replace the 26. An answer specifically suggesting that fault localization would work and also that APR might have a mutation or edit operator that changes numbers randomly (e.g., picking entirely new numbers) could potentially receive full credit, but otherwise students should argue that APR will not succeed here.

Question 7. Extra Credit (1 point each)

(*Feedback*) What was your favorite topic covered during the course?

What is one thing you like about this class?

Navigation

(*Feedback*) What was your least favorite course topic (or the thing you would most recommend that we change for future semesters)?

What is one thing you dislike about this class?

(*Guest Lecture*) List one thing you learned from guest speaker Dr. Ciera Jaspan of Google or otherwise convince us that you paid careful attention during that lecture.

Jaspan guest lecture.

(*Optional Reading 1*) Identify a single optional reading that was assigned after Exam 1. Write a sentence about it that convinces us you read it critically.

Optional Reading 1

(*Optional Reading 2*) Identify a different single optional reading that was assigned after Exam 1 or a "long instructor post" that was posted after Exam 1. Write a sentence about it that convinces us you read it critically.

Optional Reading 2

(*Guest Lecture*) List one thing you learned from guest speaker Dr. Kevin Leach that was not listed on an introductory summary slide or otherwise convince us that you paid careful attention during that lecture.

Leach guest lecture

(*Optional Lectures*) List one thing you learned from a "Game Theory", "World Building", and/or "Quantum Computing and Romance Novels" lecture that was not listed on an introductory summary slide.

Optional bonus lectures