**Coding Competency: Logical & Maintainable**

**What is Logical & Maintainable?**

Amazon has a unique type of coding interview called “Logical and Maintainable” which is quite different than a typical Data Structure & Algorithms interview. In this type of coding interview, you are expected to write clean, maintainable, and testable code. However, similar to other coding questions, you are still expected to write a workable (vs. pseudo code) but with less focus on implementing a specific algorithm.

Usually, the interview starts with a very vague question. Candidates are expected to ask many clarifying questions before proposing any approach. Once the solution is implemented, the interviewer usually adds more requirements to the original question. If the code is maintainable, the candidate should be able to quickly update the code and address the new requirement.

The interviewer wants to see if you can write code. They want to know if you can take an idea and convert it into code. Also, do you know how to take an algorithm and turn it into working code.

For SDE II, we want to see more maintainability of the code. The maintainability we want to see are better naming, better dealing with edge cases, and making sure the code is not a monolith. Do you know how to move from a monolith to something you can maintain better? You have to break it to down to different methods. Or maybe focusing on reusability of the code? Overall, we want to see if your code is readable.

Prior to coding, we see bar raising candidates that spend 7-8 minutes with the interviewer having a discussion before they go to the code in this round. Once you asked clarifying questions, removed the ambiguity, you can now start coding which should be completed within 20-22 minutes. In total, this coding session should be completed in 30 minutes.

See the bullets below technical elements required to assess an L5 SDE II candidate:

* Readability - allows for code to be well structured, field names are clear/understandable, easy to follow the logical relationships.
* Complexity of methods/testability - allows for code Elements have clear single responsibility and usage of right assertions.
* Design Patterns - Question allows candidate to be able to speak to multiple different patterns and pros/cons of one over another. Can articulate reasoning for choice of specific pattern. Code is elegant, logical, and follows solid design principles.
* Extensibility - code allows for candidate to be able to articulate how it can be extended in the future beyond specified requirements.
* The problem should not require complex algorithms or insight to solve. This coding question should be focused on the logical and maintainable portion.
* Question should allow candidate to extend code and adapt to new use cases.
* Should be able to demonstrate "Clean Code" approach.
* The problem should be structured with incremental added requirements. This challenges candidate to be able to refactor of the code.

When interviewers build out an interview question for this Logical and Maintainable, they consider the following:

* The problem should not require complex algorithms or insight to solve.
* The problem should be structured with incremental added requirements. This challenges refactoring of the code.
* The problem should involve an API or interface for external callers.
* The problem should allow the use of one or more design patterns where appropriate.

Work through the following with the candidate:

* Ask the candidate how they will diagnose issues with their code at runtime.
* The candidate should be informed that success criteria involves developing a maintainable code that is extensible for future extensions.
* The candidate should be able to demonstrate "Clean Code" approach.

If time permits, the following is "Nice to have":

* Ask the candidate to write test code.
* Ask the candidate how they would extend to new use-cases.
* Ask the candidate for a scenario where the code receives many calls and identify bottlenecks in the code.

**How to Pass Logical & Maintainable Coding**

\*In the chart below, there is no absolute scoring on the criteria and they are equivalently important. It is about making decisions based on trade-offs and risks seen in the interview (and obviously the more mild strength/full strength data points the better).

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Mild Strength** | **Strength** |
| **Readability** | Provides readable code with some hints. e.g. "if you were to make this production level code, would you change the variable names?" | Code is well structured, field names are clear/understandable, easy to follow the logical relationships. |
| **Complexity of Methods/Testability** | Breaking things into smaller components but missed out on a few things. | Elements have clear single responsibility and usage of right assertions. |
| **Design Patterns** | Finds opportunities to encapsulate and simplify. Knows of design patterns, may need help using them. | Able to speak to multiple different patterns and pros/cons of one over another. Can articulate reasoning for choice of specific pattern. Code is elegant, logical, and follows solid design principles. |
| **Adaptability of Design** | Refactor some of the design to support use case. | Refactor design while simultaneously considering similar use cases and able to explain how refactoring makes adding future use cases easier. |
| **Coherent Solutions** | Solution is coherent, meets requirements. | Solution is coherent and elegant, able to articulate how it can be extended in the future beyond specified requirements. |
| **High Level Architecture (BONUS)** | Here is what API is, similar services it may be interacting with, understands broader component and where it fits into bigger picture | |

**What makes a Coding: Logical & Maintainable question different from other types of coding questions?**

When building an interview question for this competency, consider the following:

* The problem should *not* require complex algorithms or insight to solve.
* The problem should be structured with incremental added requirements. This challenges refactoring of the code.
* The problem should involve an API or interface for external callers.
* The problem should allow the use of one or more design patterns where appropriate.

Work through the following with the candidate:

* Ask the candidate how they will diagnose issues with their code at runtime.
* The candidate should be informed that success criteria involves developing a maintainable code that is extensible for future extensions.
* The candidate should be able to demonstrate ["Clean Code"](https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29) approach.

If time permits, the following is “Nice to Have”:

* Ask the candidate to write test code.
* Ask the candidate how they would extend to new use-cases.
* Ask the candidate for a scenario where the code receives many calls and identify bottlenecks in the code.

**Sample Questions & Solutions**

**Prompt**

The Unix find command allows you to search for files under a given directory. You can specify criteria for files you are interested in.

Imagine that you need to write code in a high level language like java, that does things similar to the find command. I would like you to focus on 2 uses cases at first.

* Find all files over 5 MB somewhere under a directory.
* Find all XML files somewhere under a directory.

I would like you to create a library that lets me do this easily. Keep in mind that these are just 2 uses cases and that the library should be flexible.

**Solution**

**Expected Solution - Milestones**

1. Understands basic file object interface (give them this if needed).
2. Come up with a basic FindFile API (class with find method).
3. Come up with interface for FileMatcher abstraction (interface with match method). If they can't abstract have them solve only 1 use case first (name match).
4. Implement simple recursive file search. Could be just name search or could use matcher abstraction.
5. Implement both use cases using matcher abstraction. Skip this based on time and candidate leveling. Go back to 3 if they hard coded name search during 4.
6. Implement logical operators for matcher (logical AND and OR). Skip this based on time and candidate leveling.
7. Write code snippets showing how to use FindFile API.
8. Handle symbolic links in our file search.

**Scaling**

* **SDE I**
  + hits milestones 1, 2, 4, 7 w/ minimal help
* **SDE II**
  + hits milestones 1, 2, 4, 7 w/o help
  + hits milestones 3, 5 w/ minimal help
  + may hit milestones 6, 8 w/ some help
* **SDE III**
  + hits milestones 1, 2, 4, 7 w/o help
  + hits milestones 3, 5, 6, 8 w/ minimal help

An **SDE II** will need to implement the first use case then they will see the matcher abstraction. An SDE II may need prompting to identify the need for logical combinations of matchers. An SDE II can talk about how to deal with symbolic links w/ some basic prompting and help.

**Example Code for a Bar Raising Solution**

**public** **interface** **FileRule** {  
    **public** boolean match(File f);  
}  
  
**public** **class** **FileRuleExtension** **implements** FileRule {  
    **final** **private** String ext;  
  
    **public** FileRuleExtension(String ext) {  
        **this**.ext = ext;  
    }  
  
    @Override  
    **public** boolean match(File f) {  
        **return** f.getName().endsWith("." + ext);  
    }  
}  
  
**public** **class** **FileRuleSize** **implements** FileRule {  
    **enum** FileRuleSizeOp {  
        LT,  
        LTE,  
        EQ,  
        GT,  
        GTE  
    };  
  
    **private** **final** int size;  
    **private** **final** FileRuleSizeOp op;  
  
    **public** FileRuleSize(int size, FileRuleSizeOp op) {  
        **this**.size = size;  
        **this**.op = op;  
    }  
  
    @Override  
    **public** boolean match(File f) {  
        **switch** (op) {  
            **case** LT:  
                **return** f.length() < size;  
            **case** LTE:  
                **return** f.length() <= size;  
            **case** EQ:  
                **return** f.length() == size;  
            **case** GT:  
                **return** f.length() > size;  
            **case** GTE:  
                **return** f.length() >= size;  
        }  
  
        **return** **false**;  
    }  
}  
  
**public** **class** **FileRuleAnd** **implements** FileRule {  
    **private** **final** List<FileRule> andRules;  
  
    **public** FileRuleAnd(FileRule ... andRules) {  
        **this**.andRules = Arrays.asList(andRules);  
    }  
  
    @Override  
    **public** boolean match(File f) {  
        **for** (FileRule rule : andRules) {  
            **if** (!rule.match(f)) {  
                **return** **false**;  
            }  
        }  
  
        **return** **true**;  
    }  
}  
  
**public** **class** **FileRuleOr** **implements** FileRule {  
    **private** **final** List<FileRule> orRules;  
  
    **public** FileRuleOr(FileRule ... orRules) {  
        **this**.orRules = Arrays.asList(orRules);  
    }  
  
    @Override  
    **public** boolean match(File f) {  
        **for** (FileRule rule : orRules) {  
            **if** (rule.match(f)) {  
                **return** **true**;  
            }  
        }  
  
        **return** **false**;  
    }  
}  
  
**public** **class** **FindFile** {  
    **public** **static** void find(String dir, FileRule rule) {  
        File baseDir = **new** File(dir);  
        **if** (baseDir.isDirectory()) {  
            **for** (File f : baseDir.listFiles()) {  
                **if** (rule.match(f)) {  
                    System.out.println(f.getName());  
                }  
            }  
        }  
    }  
  
    **public** **static** void main(String[] args) {  
        FileRule rule = **new** FileRuleOr(  
                            **new** FileRuleExtension("java"),  
                            **new** FileRuleAnd(  
                                **new** FileRuleExtension("zip"),  
                                **new** FileRuleSize(5711472, FileRuleSize.FileRuleSizeOp.EQUAL)  
                            )  
                        );  
        find("/home/local/ANT/lestrozi", rule);  
    }  
}

**Sample Coding – Logical & Maintainable questions**

***Question 1: Currency Denominations***

**Prompt**

*Given a sum of money compute the minimum number of bills and coins that equal that sum. Assume you only have the following denominations:*

*Bills: 20, 10, 5, 1,*

*Coins: .25, .1, .05, .01*

*For example, given 6.35 the solution would be One 5, One 1, One .25, One .1*

**NOTES for interviewer:**

* These coding questions are geared to focus on logical and maintainable, not data structures or problem solving as the crux of the problem. Please set expectations with the candidate on what you are looking for out of this coding round, so that they can focus on logical and maintainable code.
* candidate defining output is a critical part of the problem
* Uses USD as a currency but is not explicit about it, can be understood by anyone around the world and currency understanding specifically is not needed, just math skills.
* As an SDE II, defining the problem statement may mean defining the meaning of bills and coins. They can be viewed as denominations, not bills and coins if needed.
* most engineers will not have all design patterns memorized and that is ok, as long as you write a greedy algorithm. This question is not often solved with a known design patterns.
* first part of this question is straight forward, but critical part of this problem to get L5 SDE data is the extension part of the problem.

**Follow up questions as the interviewer for L4 SDE data points:**

* Getting through initial question and implementing cash drawer assuming good input should be bar raising at L4

**Follow up questions as the interviewer for L5 SDE data points:**

* You will now be given a cash drawer that defines how much of each bill and coin you have to make change with, please alter your solution to account for this.
* Edge Cases: Candidate should identify or be hinted towards two edge cases, not having enough money, and not being able to make exact change.
* For exact change edge case, throwing error is not enough, they should be able to return more change than asked (customer obsession) within a certain range (Don't give a 20 if 1 cent is owed, but a nick might be ok)

**Follow up questions as the interviewer for L6 SDE data points:**

* Same as SDE2 level, expect little to no hinting on edge cases or implementation

***Question 2 – Calculator Notation***

**Prompt**

*We can define a calculator notation in which the operators follow the operands, for example where a normal calculator input would be input as (2 + 3) \* 5, our calculator can take input like 2 3 + 5 \* to avoid the use of parenthesis.*

*The task is to implement this calculator for plus, minus, multiply and divide operations.*

*Assume the input will be passed as an array of String => ["16","5","+"].*

**NOTES for interviewer:**

* These coding questions are geared to focus on logical and maintainable, not data structures or problem solving as the crux of the problem. Please set expectations with the candidate on what you are looking for out of this coding round, so that they can focus on logical and maintainable code.
* candidate may need to ask follow up questions on what the calculator is or what operators vs operands. They can be confused easily. Operands = numbers and operators are +, -, divide, etc.
* Basic implementation requires knowledge of a stack, which can be fairly easily hinted to a minimally qualified candidate.
* Candidates/interviewers might need hinting on virtual functions for C++ - L5 we expect them to know we need to do a virtual function/polymorphism but its ok if they need some hints on what the functions are and if we put them in the bank the interviewer can provide virtual functions for some of these program languages to assist and keep problem moving
* Candidates/interviewers might need hinting interfaces for python and js ^^ same as above
* this question targets candidate ability to write code that allows for code Elements have clear single responsibility and usage of right assertions. look for clear data point around separation of concern and object responsibilities.
* first part of this question is straight forward, but critical part of this problem to get L5 SDE data is the extension part of the problem.

**Follow up questions as the interviewer for L4 SDE data points:**

* SDE1 should be able to implement up until the interface development which is the second follow up question. they should complete square root part, and also implement interface part. Also OK if they can describe how to do interface part and not code it.

**Follow up questions as the interviewer for L5 SDE data points:**

* Now add square root operations (twist because it tests logical and maintainable because if building assuming it would always be 2, they would have to re-write a lot of code and it only takes 1
* Now build an interface for users to define their own operations (this is where L5 becomes bar raising, now the concepts are introduced with operators with 1 operand, but what if we take the calc to the next level - users finding their own functions. How would we write a function that takes in 4 operands instead? We are looking for them to realize they should write a functional that generically handles this RPN format and make it logical/maintainable. At the end of the exercise, the candidate should have code with a function that is only responsible for RPN processing and an interface and method of definition of any user defined function

**Follow up questions as the interviewer for L6 SDE data points:**

* Same as SDE2 level, expect little to no hinting on edge cases or implementation

***Question 3 – Online Search***

**Prompt**

Amazon products are stored hierarchically within their respective categories.

Products have various metadata, like current price or if it's prime-eligible.

E.g. a dress might be located under Clothing > Women > Dresses, with a price of 20 and it is prime-eligible.

It should be easy to add additional criteria to search by in the future.

But initially only needs to handle querying by category or max price.

**NOTES for interviewer:**

* These coding questions are geared to focus on logical and maintainable, not data structures or problem solving as the crux of the problem. Please set expectations with the candidate on what you are looking for out of this coding round, so that they can focus on logical and maintainable code.
* The question is asked ambiguously at first, but specific inputs/outputs are easy to provide. (this ties back to section 1/question 1). Add these follow up prompts on question bank to help interviewers if this happens.
* Question is fairly ambiguous at first if the candidate doesn't already know about searching on a shopping website.  
  It is expected the candidates asks clarifications about how the products are stored to be searched, since we do not provide that interface up front.
* Question is pretty ambiguous at first if the candidate doesn't already know about the find command and about their language's file interface.  
  It should be totally acceptable for them to ask even general questions about those.
* It should be easy to tell where the search criteria are checked and how the product traversal algorithm works.
* Adding additional criteria and the AND checks shouldn't require modifying their code very much.

**Follow up questions as the interviewer for L4 SDE data points**

* Now ask them to extend it to handle shipping speed. They should be able to build atop their existing code with minimal re-architecture to pivot to more of an SDE I calibration, give them some of the structure and nudge them towards something that can be easily extended. Focus on just one part. break down the question a little more for them. Example: Give them the recursion, then ask them to focus on the rule set? What other examples can we outline here? Now ask them to extend it to handle file size. They should be able to build atop their existing code with minimal re-architecture.

**Follow up questions as the interviewer for L5 SDE data points:**

* Ask candidate to extend it to handle shipping speed and AND on multiple criteria.

**Expectation:** They should be able to build atop their existing code with minimal re-architecture for any of extensions. Look for data points that the candidate can extend and still make the code easy to read.

* Things interviewers may need to assist the candidate with (interviewer may not be familiar with the exact language candidate is using: 1) For each language, what might your file interface look like? write this out. 2) list of possible extensions? let interviewer/candidate pick from it. pivot to another extension if candidate is already familiar with implementing it?

**Follow up questions as the interviewer for L6 SDE data points**

* Same as SDEII plus they can discuss scaling to handle large data sets. How would you scale your functional so that it meets 10's of thousands of TPS? \*\*get into scalability discussion) - NOTE: SDE II cannot evaluate an SDE III, this SDE II would go through an SDE III mini loop. Looking for a generic solution without much hinting.

***Question 4 – Calculator Design***

**Prompt**

Design a basic calculator, or arithmetic expression evaluator.

Given an input String, return an integer.

At first it should only handle addition and subtraction operators, but expect for additional operators to be added.

**NOTES for interviewer:**

* These coding questions are geared to focus on logical and maintainable, not data structures or problem solving as the crux of the problem. Please set expectations with the candidate on what you are looking for out of this coding round, so that they can focus on logical and maintainable code.
* interview needs to be careful to not focus too much on the DS and implementing it but try and focus writing logical and maintainable code and getting data points here. This question could be asked for DS and Algo question as well. L&M parts would be separating a driver/evaluator (OOD) and then adding on extensions to the problem. common trap is adding in multiple IF statements make sure interviewer is probing for if they can avoid this for maintainable code. how easy is it to come in and add a new functionality to the code?
* Only basic data structures are necessary, and some kind of extensible algorithm. there are multiple ways to solve this problem
* Libraries exist that do this for Java, C++, C#, Go, do not let them use those libraries.
* JS and Python both have libraries to evaluate arithmetic strings, do not let them use those libraries..

**Follow up questions as the interviewer for L4 SDE data points:**

* Ask them to handle multiply and divide, preserving operation order.

**Follow up questions as the interviewer for L5 SDE data points:**

* Ask them to handle multiply, divide, and parentheses, preserving operation order.

**Follow up questions as the interviewer for L6 SDE data points:**

**Other Considerations**

**Calibration**

The code has lots of logical and maintenance problem to notice. We do not expect candidate can catch everything. Candidate might miss some parts during the interview, especially on coding style and testing (they might just focus on the interface and performance). It is okay to give candidate a hint. For example: "do you have feedback on my coding style?" or "do you think I should add test cases?" And then see whether the candidate can drive deeply discussion/conversation.

**Concurrency**

**Meet the Bar**

* Fix synchronization bugs
* Propose prefer concurrent data structure than synchronization or synchronized data structure with well-reasoned justification, e.g. better concurrency performance.

**Raise the Bar**

* Propose correct non-blocking approaches. A common answer for non-blocking purging is to have the class spin up a background thread plus a hand wave. A possible bar raising answer recognizes the possibility of contention and/or the purging thread getting starved. For example, add and contains method can still be blocked on the background thread, leading to the same expected performance. In addition, question about the concurrency story, e.g. is there a thread per instance of this class? What if I have many instances of this class? How to handle thread pool life-cycle and make sure the background thread is doing the work as expected?

**Interface**

**Meet the Bar**

* Question the use case to evaluate if the code is appropriate (solving the right problem).
* Discussion of whether add and contains should both doing purge.
* Fix misuse of exceptions in contains method and discuss on when you would or would not throw exception and let caller handle it.

**Raise the Bar**

* Question whether purgeExpiredKeys should be public. Sometimes, candidate might simply push the hard decisions to the caller and so has not solved the problem in the grand scheme of things. For example, exposing a public purge method and letting caller decide when and how to purge the expired data. This is potentially a red flag. Question around does the caller have to call purge at a certain time, or at a certain frequency, for the data structure to operate correctly? Is it a good thing?
* Question whether class should be written at all and/or if class should be made more abstract or generic (interface, generic keys, key/value store). Discussion of when you would or would not recommend making a class more generic. Discussion of whether you would or would not recommend a class for broader consumption, and what additional consideration come into play if it will be.

**Coding Style**

**Meet the Bar**

* Fix coding style issues, e.g. white-spacing, braces, indentation, method/variable casing, method/variable naming.

**Raise the Bar**

* Well-reasoned justification for coding style fix, i.e. consistency and preventing bugs trumps aesthetics and individual preference.
* Discussion of automated code formatting tools to achieve benefits while avoiding time wasted on these aspects.
* Discussion about how to install best practices in the team: on-boarding, style guides & tools. *(Hire and Develop the Best)*
* Discussion about the benefit of separating minor/stylistic feedback from content-based feedback.

**Code Review**

**Meet the Bar**

* Ask for clear description of use case, code change/code purpose, etc. in Code Review description and document of all public methods.
* Ask for tests and consuming code to be included in the same Code Review.

**Raise the Bar**

* Code Review description includes availability and durability concerns, operational excellent artifacts (e.g. metrics, logging), and testing done, etc..
* Discussion about how to install & process best practices like pair programming or intentionally choosing code reviewers.
* Discussion of how to give constructive Code Review feedback or effectively mentor engineers. *(Hire and Develop the Best)*

**Testing**

**Meet the Bar**

* Implementation needs test cases, e.g. input validation test cases, simple add and contains test cases, multi-threads add and contains test cases.
* Testing code should be included in the same Code Review.

**Raise the Bar**

* Propose dependency injection to easily test expiry function. For example, abstract the expiration date as an interface which allows to inject dependency like a virtual clock. The virtual clock can simulate any expiration period without having to wait for physical time to pass.
* Expose methods for determine the number of expiry and unexpired entries. Otherwise we cannot actually verify that purging function. Discuss appropriate access level for those methods, public/package/protected/private/@VisibleForTesting etc.. A bar raising approach is to extract an interface for add and contains, and have those methods as public in the implementation.
* Discussion for more than unit tests to validate correctness, particularly w/r/t concurrency, e.g. adding a component test, load test, fuzz test.

**Example Use Case**

* Throttle bad actors, e.g. keeping a list of "bad" hosts or users (IP string, hosts names, user names, etc.) that expire after a timeout - based on some heuristic (incoming request rate, incoming request volume, too many login attempts, etc.).
* Exclude peers after failure and/or maintaining healthy peers (i.e. health checks) in a distributed system.
* Enforce at-most-once processing of items over some window, with a time-based or size-based bound on the de-duplication window.
* Modify the example into a Map/Dictionary gives you a type of cache.