Machine Coding Round. What's This?

The general interview process for software engineers consisted of 2–3 rounds focused on problem-solving, data structures and algorithms, coupled with 1–2 rounds focused on checking the culture fit of the candidate. However recently, more and more companies are adopting the ***Machine Coding Round***. It is a pretty crucial round since most of the people get eliminated in this round. It is completely different from what everyone generally prepares for.

Flipkart, Uber, Swiggy, Ola, Cred, etc. are some of the top tech companies where the first onsite round is the machine coding round.

What's It Like ?

It involves **solving a design problem** in a matter of a couple of hours. It requires designing and coding a clean, modular and extensible solution based on a specific set of requirements. An example of a machine coding problem could be to - ***Design a snake and ladder game with certain requirements and constraints***, which is followed by a code review process where an interviewer goes through the code and tries to understand the design decisions.

What's Expected From You?

Almost all the companies have similar judging criteria in the machine coding round. Some of them are :

1. Code should be **working and demonstrable**.
2. Code should be **functionally correct**.
3. Code should be **modular and readable**.
4. Separation of concerns should be **addressed**.
5. Code should easily **accommodate new requirements** with minimal changes.
6. There should be a **main method** from where the code could be **easily testable**.
7. A **UI** is generally **not required**.

How Shall You Prepare ?

Since machine coding round is a relatively **new interview format**, **very few** candidates prepare for it and hence **face elimination** in this round.

Let's look at the primary things you need to work on to crack it:

Learn Object-Oriented Programming

Here, the general expectation is that you'll write object-oriented code.

- Writing everything in a single file without any classes is **just not acceptable** in the interview. You are supposed to **create multiple classes each containing properties and methods only related to that particular class** (there should be a separation of concerns).
- Each class should have a file **of its own**. Single file is fine if you're coding in **C++**.
- You should be able to identify all the entities in the problem statement and create classes for them.
- The entity classes (or models) should ideally reside in a **'models**' folder. A model should only contain the **properties of the entity, a constructor, getters, and setters for the properties**.
- There **should not be any business logic** in any model. And so you should create different classes for the business logic layer. These classes interact with different models and contain the core logic of the solution. They are generally called **services** and are kept in the **'services' folder**.
- The **main method** which will be used to test your solution should reside in a **Driver class**. Anything that **does not deal with the core logic** of the solution, like taking user input, should reside here.

Learn to write readable code

Most of us are used to writing code that is only read by a computer. This is **not the case** in an interview or on the job where the **code is read by actual people** as well.

Hence, it becomes essential to:

- **Learn how to write proper and self-explanatory variable and method names**. The interviewer should be able to understand the purpose of the variable

or the method name just by reading its name. If that is not happening then you'll most likely **get rejected** even if your code is fully functional with a decent design.
- **Create method names** based on what is actually happening inside that method instead of why the caller is calling that method. Ensuring this makes the method/function reusable at multiple places & **allows the interviewer to skip going through it** and instead focus on the core logic.
- **Create smaller methods** and avoid having a lot of logic in a single method. Breaking the code in a method into multiple smaller methods makes your code **more readable and reusable**.

Practice

You're supposed to write a **clean, modular and extensible code** in a couple of hours in a machine coding round, which requires *Practice!!*

If you're able to write a **well-designed code in a couple of hours** then you're good to go. If not, practicing a few problems and improving with each attempt may help you get there.

Bonus: This tip is based on a few solutions in our 1st mock machine coding round. Either use **camelCase or underscore_separated** based on the language's convention. Please **do not use both** in your solution.

(*This is a basic guide on how to prepare for the round. You may have to learn more concepts related to design principles and patterns, multithreading, etc. based on your seniority level.*)

How To Ace The Round ?

Anything can be aced at if you dissect it into various stages & properly work out the required steps. Here, we've divided the process for you:

After getting the question (5-10 mins)

- **Read the problem statement carefully**. Try to clearly understand all the requirements.
- **Do not assume** anything that is not mentioned in the problem statement. If you want to make a specific assumption, discuss it with the interviewer at this stage.

- **Ask as many clarifying questions** as you can think of so as to make the requirements clear and remove any room for ambiguity or misinterpretation.

Getting to the solution (10-15 mins)

- Spend ~10 minutes **thinking about the design** of your solution. This is important. You do not want to start coding before a proper design.
- While designing, think about how you can **make it extensible** to accommodate the optional requirements or any common extension that you can think of.
- **Estimate how much time it will take you to code** with all the requirements. **Prioritize** which ones to do so as to at least solve the most critical ones.
- Apart from the design, estimate some time for creating the **main method or API/CLI interface** as mentioned or as clarified with the interviewer.
- Also, estimate some time for **testing** and making changes to have the solution working.
- If your current solution will take a lot of time to code, try to think of a simpler design that is good enough and will take less time to code. This is one of the biggest trade-offs in a machine coding round (and in general, software development).
- **Do not prioritize** an optional requirement over a mandatory requirement. If you focus more on extensibility and future requirements and are unable to complete a mandatory requirement, it's a red flag.
- **Optional:** If possible, draw a UML diagram of your design for clarity.

Coding (60-75 mins)

If you've designed the solution and are comfortable in coding, this step should be fairly easy, just make sure to:

- **Code fast** so as to complete as many requirements as possible.
- Have a **working code** in the end**.** Be ready with good **sample examples** to demonstrate your solution.
- **Gracefully handle exceptions** and other corner cases. You do not want your code to fail during the demonstration.
- **Write readable code** with proper names. Use comments, if possible. You are writing the code for your interviewer to read and understand.
- Use a powerful IDE that you are comfortable with. Choose one where you can generate most of the boilerplate code to save time.

Demonstration

While demonstrating, make sure to:

- Give a **high-level overview** of your solution. **Do not** explain each and every line of the code. Your code should be **modular and self-explanatory.**
- **Tell the interviewer** about all the requirements that you've completed and if your solution is extensible for other requirements.
- After running your solution on sample input, it may be a good idea to **ask the interviewer** if they want you to test with any other input.

Need Some Practice Questions ?

We've created a few machine coding problems that you can practice to get prepared for the actual machine coding round interview of top tech companies like Flipkart, Uber, Swiggy, Udaan, Razorpay.

Design Snake & Ladder

Problem Statement

Create a snake and ladder application. The application should take as input (from the command line or a file):

- Number of snakes (s) followed by s lines each containing 2 numbers denoting the head and tail positions of the snake.
- Number of ladders (l) followed by l lines each containing 2 numbers denoting the start and end positions of the ladder.
- Number of players (p) followed by p lines each containing a name.

After taking these inputs, you should print all the moves in the form of the current player name followed by a random number between 1 to 6 denoting the die roll and the initial and final position based on the move.

Format: <player_name> rolled a <dice_value> and moved from <initial_position> to <final_position>

When someone wins the game, print that the player won the game.

Format: <player_name> wins the game

Rules of the game

- The board will have 100 cells numbered from 1 to 100.
- The game will have a six sided dice numbered from 1 to 6 and will always give a random number on rolling it.
- Each player has a piece which is initially kept outside the board (i.e., at position 0).
- Each player rolls the dice when their turn comes.
- Based on the dice value, the player moves their piece forward that number of cells. Ex: If the dice value is 5 and the piece is at position 21, the player will put their piece at position 26 now (21+5).
- A player wins if it exactly reaches the position 100 and the game ends there.
- After the dice roll, if a piece is supposed to move outside position 100, it does not move.
- The board also contains some snakes and ladders.
- Each snake will have its head at some number and its tail at a smaller number.
- Whenever a piece ends up at a position with the head of the snake, the piece should go down to the position of the tail of that snake.
- Each ladder will have its start position at some number and end position at a larger number.
- Whenever a piece ends up at a position with the start of the ladder, the piece should go up to the position of the end of that ladder.
- There could be another snake/ladder at the tail of the snake or the end position of the ladder and the piece should go up/down accordingly.

Assumptions you can take apart from those already mentioned in rules

- There won't be a snake at 100.
- There won't be multiple snakes/ladders at the same start/head point.
- It is possible to reach 100, i.e., it is possible to win the game.
- Snakes and Ladders do not form an infinite loop.

Sample Input

9

62 5

33 6

49 9

88 16

41 20

56 53

98 64

93 73

95 75

8

2 37

27 46

10 32

51 68

61 79

65 84

71 91

81 100

2

Gaurav

Sagar

Sample Output

Gaurav rolled a 6 and moved from 0 to 6

Sagar rolled a 1 and moved from 0 to 1

Gaurav rolled a 6 and moved from 6 to 12

Sagar rolled a 4 and moved from 1 to 5

Gaurav rolled a 4 and moved from 12 to 16

Sagar rolled a 6 and moved from 5 to 11

Gaurav rolled a 5 and moved from 16 to 21

Sagar rolled a 4 and moved from 11 to 15

Gaurav rolled a 1 and moved from 21 to 22

Sagar rolled a 6 and moved from 15 to 21

Gaurav rolled a 6 and moved from 22 to 28

Sagar rolled a 2 and moved from 21 to 23

Gaurav rolled a 6 and moved from 28 to 34

Sagar rolled a 6 and moved from 23 to 29

Gaurav rolled a 5 and moved from 34 to 39

Sagar rolled a 2 and moved from 29 to 31

Gaurav rolled a 2 and moved from 39 to 20

Sagar rolled a 5 and moved from 31 to 36

Gaurav rolled a 3 and moved from 20 to 23

Sagar rolled a 5 and moved from 36 to 20

Gaurav rolled a 6 and moved from 23 to 29

Sagar rolled a 3 and moved from 20 to 23

Gaurav rolled a 2 and moved from 29 to 31

Sagar rolled a 3 and moved from 23 to 26

Gaurav rolled a 3 and moved from 31 to 34

Sagar rolled a 5 and moved from 26 to 31

Gaurav rolled a 3 and moved from 34 to 37

Sagar rolled a 4 and moved from 31 to 35

Gaurav rolled a 2 and moved from 37 to 39

Sagar rolled a 5 and moved from 35 to 40

Gaurav rolled a 2 and moved from 39 to 20

Sagar rolled a 5 and moved from 40 to 45

Gaurav rolled a 2 and moved from 20 to 22

Sagar rolled a 6 and moved from 45 to 68

Gaurav rolled a 3 and moved from 22 to 25

Sagar rolled a 3 and moved from 68 to 91

Gaurav rolled a 5 and moved from 25 to 30

Sagar rolled a 2 and moved from 91 to 73

Gaurav rolled a 5 and moved from 30 to 35

Sagar rolled a 6 and moved from 73 to 79

Gaurav rolled a 5 and moved from 35 to 40

Sagar rolled a 1 and moved from 79 to 80

Gaurav rolled a 4 and moved from 40 to 44

Sagar rolled a 2 and moved from 80 to 82

Gaurav rolled a 5 and moved from 44 to 9

Sagar rolled a 4 and moved from 82 to 86

Gaurav rolled a 1 and moved from 9 to 32

Sagar rolled a 6 and moved from 86 to 92

Gaurav rolled a 3 and moved from 32 to 35

Sagar rolled a 4 and moved from 92 to 96

Gaurav rolled a 1 and moved from 35 to 36

Sagar rolled a 1 and moved from 96 to 97

Gaurav rolled a 1 and moved from 36 to 37

Sagar rolled a 5 and moved from 97 to 97

Gaurav rolled a 6 and moved from 36 to 42

Sagar rolled a 3 and moved from 97 to 100

Sagar wins the game

Expectations

- Make sure that you have a working and demonstrable code
- Make sure that the code is functionally correct
- Code should be modular and readable
- Separation of concern should be addressed
- Please do not write everything in a single file
- Code should easily accommodate new requirements and minimal changes
- There should be a main method from where the code could be easily testable
- [Optional] Write unit tests, if possible
- No need to create a GUI

Optional Requirements

Please do these only if you've time left. You can write your code such that these could be accommodated without changing your code much.

- The game is played with two dice instead of 1 and so the total dice value could be between 2 to 12 in a single move.
- The board size can be customizable and can be taken as input before other input (snakes, ladders, players).
- In case of more than 2 players, the game continues until only one player is left.
- On getting a 6, you get another turn and on getting 3 consecutive 6s, all the three of those get canceled.
- On starting the application, the snakes and ladders should be created programmatically without any user input, keeping in mind the constraints mentioned in rules.