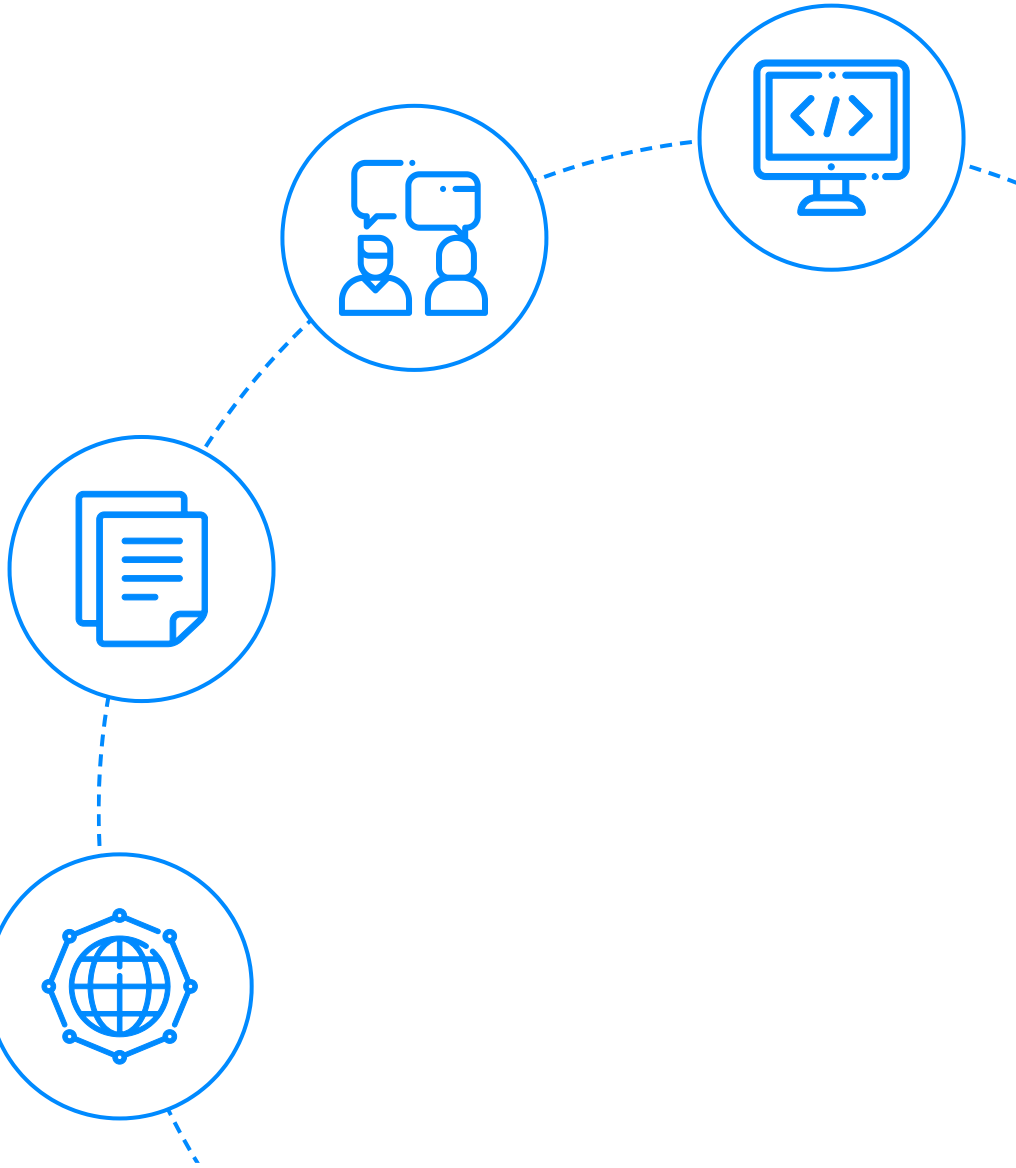
InterviewBit

Low Level Design Interview Questions



To view the live version of the page, click here.

Contents

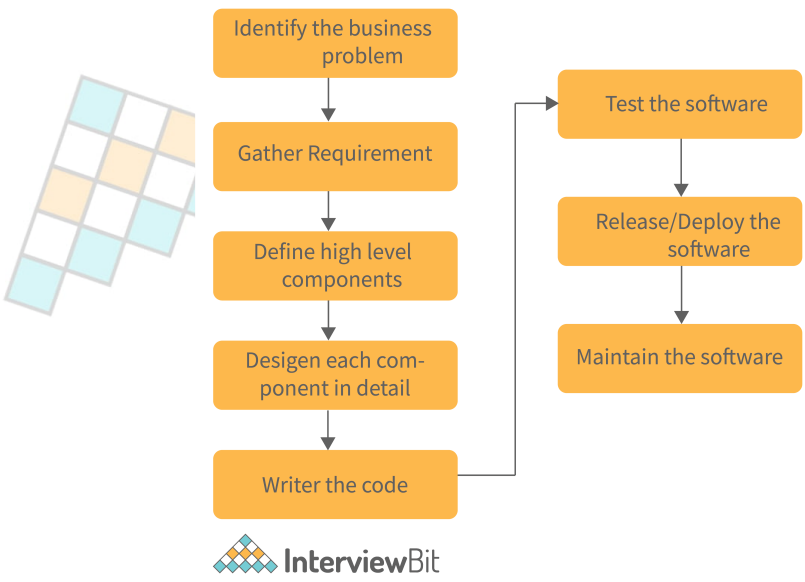
Low Level Design Interview Questions: Freshers & Experienced

Tips on Cracking the Low Level Design (LLD) Interview

Let's get Started

Introduction

Generally, we develop software to solve a particular business problem. Developing any software follows a procedure as identifying the business problem, collecting the functional requirements for the identified problem, designing the overall architecture of the software/system by defining the building blocks called components, designing the individual components, actually writing the code, testing the software, deploy/release the software and maintain the software.



Before writing the actual code, we do 2 crucial steps. One is defining the high-level components typically called High-Level Design (HLD) and another is designing each component in detail typically called Low-Level Design (LLD).

What is High-Level Design (HLD)?

In HLD, the focus is more on designing the high-level architecture of the system, defining the high-level components with their interactions, and also the database design. HLD converts the business requirements to a high-level solution.

What is Low-Level Design (HLD)?

In LLD, the focus is more on designing each component in detail such as what classes are needed, what abstractions to use, how object creation should happen, how data flows between different objects, etc. LLD converts the high-level design into detailed design (ready to code) components.

It is common nowadays to have a **Low-Level Design round** (or) **Pair Programming round** (or) **Machine Coding round** in tech interviews.

In this article, we discuss why LLD is important, **how to prepare for low-level design interview questions for freshers and experienced** candidates with some examples, and finally conclude with some tips.

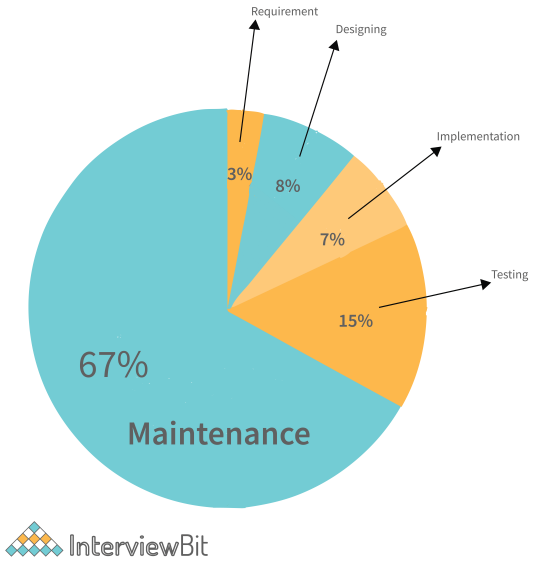
Low Level Design Interview Questions: Freshers & Experienced

1. Why LLD is Important?

As a software developer in any company, the objective is to build software for the business of the company. Building software is a complex task. Change is the only constant thing in the world. As the software progresses over the years, it needs to incorporate a lot of changes as the requirements keep on changing. We can choose to build software in many ways but we need to build it in such a way that the software is maintainable and extensible over the years easily.

Most of the software developer interviews have Data Structures & Algorithms round(s). But it is not often for software developers to use these data structures and algorithms in the real world while building software. (If you are an experienced candidate, can you recall when was the last time you used the Dijkstra algorithm? If you are fresher, ask any of your senior colleagues the same question.)

On the other hand, Low-Level Design is a must for building a piece of software. As LLD focuses on how to build software from a set of requirements, how different components interact with each other, what responsibilities each component has etc, it is a vital exercise to be done before actually writing the code. Writing code is the easiest part of building software if we have the designs already in place. Maintaining and Extending software will be easy if designs are well thought of.



As we can see from the above picture, Maintenance takes almost 70% of the effort in the software development life cycle. So the software should be easily maintainable and extensible. LLD plays an important role in the design of software and hence directly contributes to the maintainability and extensibility of software.

2. How to prepare for Low-Level Design Interviews?

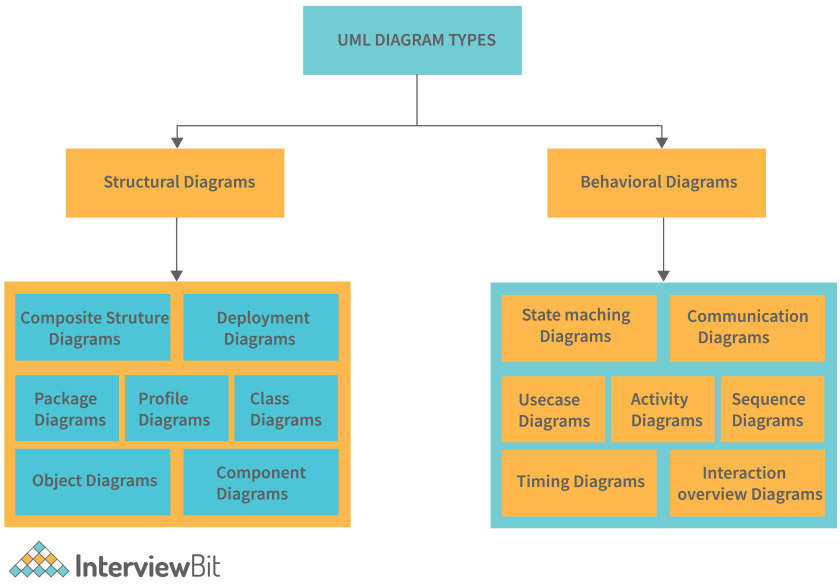
Software Developers need to learn Low-Level Design not only to crack the Interviews but also to build modular, extensible, reusable, and maintainable software. It is the thought process that developers need to develop to effectively build the software from a set of requirements.

Here is the list of things that a software developer needs to learn for LLD Interviews:

- **Object-Oriented Language**: The general expectation in the machine coding round (or) LLD round is that candidate writes Object Oriented Code. As the name suggests, Object-Oriented Languages revolve around everything about Objects. The candidate should identify the different entities that need to be created from the problem statement and model the classes as per the required entities along with choosing the right set of data structures depending on the use case.
  Learning any Object-Oriented Language is a must as the candidate needs to write code using that language. Some of the popular Object-Oriented Languages are Java, C#, C++, and Python.
- **Object-Oriented Principles**: Knowing an Object-Oriented Language itself is not sufficient for writing extensible and maintainable code. Many Object-Oriented principles are designed to make the software extensible and maintainable. Some of the well-known principles are :
  - **YAGNI**: **Y**ou **A**in't **G**onna **N**eed **I**t is a practice in software development that states that features should only be added when required and thus help trim away excess and inefficiency development to facilitate the desired increased frequency of releases)
  - **DRY**: **D**on't **R**epeat **Y**ourself is a principle that states that don't repeat the code again and again. If there is a code that is duplicated then whenever a change needs to be made, the change has to be done in both places. So the developer has to remember all the places where the code is duplicated which are difficult and error-prone. Thus this principle states that the code should never be duplicated and has to refactor into its method or class and all other places need to use this method/class instead of duplicating the code.
  - **SOLID**: These are a set of 5 principles. **S**ingle Responsibility, **O**pen-Closed, **L**iskov Substitution, **I**nterface Segregation, and **D**ependency Inversion.

Apart from these, there are well-known principles such as favouring composition over inheritance, Encapsulating what varies, Striving for loosely coupled classes, etc.

- **UML Diagrams:** UML is an acronym that stands for Unified Modeling Language and is a standard for modelling, visualizing, and documenting the artefacts of software systems. It is not a programming language but a tool to visually depict different types of diagrams that can be useful for building the software.
    - There are 14 different types of UML Diagrams and these 14 diagrams are classified into 2 high-level groups.
        - **Structural Diagrams:** They represent the static view of the system. Class Diagram, Composite Structure Diagram, Object Diagram, Component Diagram, Deployment Diagram, Profile Diagram, and Package Diagram are part of Structural Diagrams.
        - **Behavioural Diagrams:** They represent the dynamic view of the system. Activity Diagram, State Machine Diagram, Use Case Diagram, Interaction Overview Diagram, Timing Diagram, Sequence Diagram, and Communication Diagram are part of Behavioral Diagrams.



Class Diagram & Use Case Diagram are generally used in LLD rounds.

- **Design Patterns**: Design Patterns are typical solutions to common problems in software design. "Don't reinvent the wheel" is a well-known phrase in software engineering. If a problem is already solved, don't solve the same problem to get the same solution. Instead, use the solution to the problem. Similarly, there are a lot of common problems in software design that are solved and are given as a toolkit called Design Patterns. Design Patterns are not specific to a particular language. They are the general solutions that specify how a design problem can be solved and hence can be implemented in any language. These design patterns are proven and tested and hence make the development process smooth and easy.
- **Practice LLD Questions:** Try to practice as many low-level design interview questions as possible. Solving different problems gives different perspectives and thereby improves design skills. There is no right solution for design problems as opposed to data structures and algorithmic problems. So solving different problems gives different ideas and thus develops the overall thought process for solving any design problem.

3. How to Solve Low-Level Design problems in Interviews?

Solving an LLD problem in an interview can be done easily if we divide the solution into multiple stages and focus on solving these stages one by one. LLD problems can be broadly categorized into 2 types: Standalone applications, and web applications.

Solving an LLD problem can be divided into mainly 3 stages:

1. Clarify & Gather Requirements: The low-level design interview questions are intentionally open-ended/ unstructured similar to real-life conditions. Ask the relevant questions regarding the problem (good questions are the ones that help to understand more about the system behaviour, features, and what features are expected to be added to the system in the future) and gather the complete requirements of the system. Don't assume anything beforehand and always clarify with the interview the assumptions you want to take. Write down the requirements & assumptions and discuss them with the interviewer.

2. Class Diagram & Use Case Diagram & Schema Diagram (If required): Once the requirements are gathered, define the core classes and objects, how the different classes interact with each other, and actors who interact with the system, the use cases for each actor, etc. Draw the class diagram and use a case diagram (can be optional, ask the interviewer) for the system. Define the relations between classes by observing the interactions between them. If Class Diagram & Use Case Diagram helps in representing the requirements of systems and the relationship between different classes, a Schema Diagram (also called an ER Diagram) helps in how the data models look and how to store the data in the database. Interviewers may not ask to write the code to store the data in the actual database but they may be interested in how the model looks like and the relationship between them. It is easy to create the Schema Diagram from the Class Diagram. Each class in the class diagram becomes a table in Schema Diagram and the relationship between classes becomes the multiplicity between tables.

3. Code: Finally once the thoughts are structured using Class Diagram, Use Case Diagram, and Schema Diagram (if required), the Candidate can start writing the code. Apply the Design Patterns, Object-Oriented Principles & SOLID Principles wherever is possible to make the system reusable, extensible, and maintainable. Make sure the code is well structured and follow the clean coding practices to make the classes and method clean. Don't try to fit design patterns to code but check if a given problem can be solved using any available design pattern. Take care of the readability of code while taking care of all of the above. Yes, software engineering is hard.

4. How to design Snake and Ladder?

Here is how the "**Snake and Ladder problem**" can be solved by using the above approach.

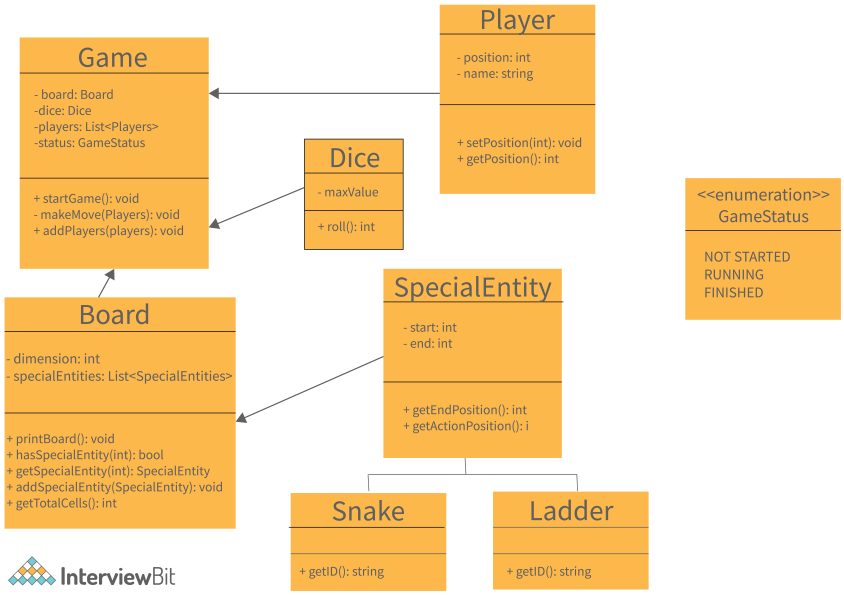
1. Gathering Requirements: Create a multiplayer Snake and Ladder game in such a way that the game should take input N from the user at the start of the game to create a board of size N x N. The board should have some snakes and ladders placed randomly in such a way the snake will have its head position at a higher number than the tail and ladder will have start position at smaller number than end position. Also, No ladder and snake create a cycle and no snake tail position has a ladder start position & vice versa.

The players take their turn one after another and during their turn, they roll the dice to get a random number and the player has to move forward that many positions. If the player ends up at a cell with the head of the snake, the player has to be punished and should go down to the cell that contains the tail of the same snake & If the player ends up at a cell with the start position of a ladder, the player has to be rewarded and should climb the ladder to reach the cell which has a top position of the ladder. Initially, each player is outside of the board (at position 0). If at any point of time, the player has to move outside of the board (say player at position 99 on a 100 cell board and dice rolls give 4) the player stays at the same position.

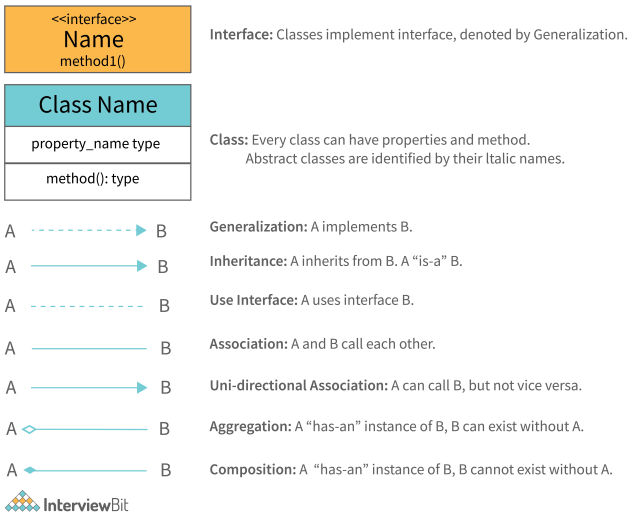
Possible future extension:

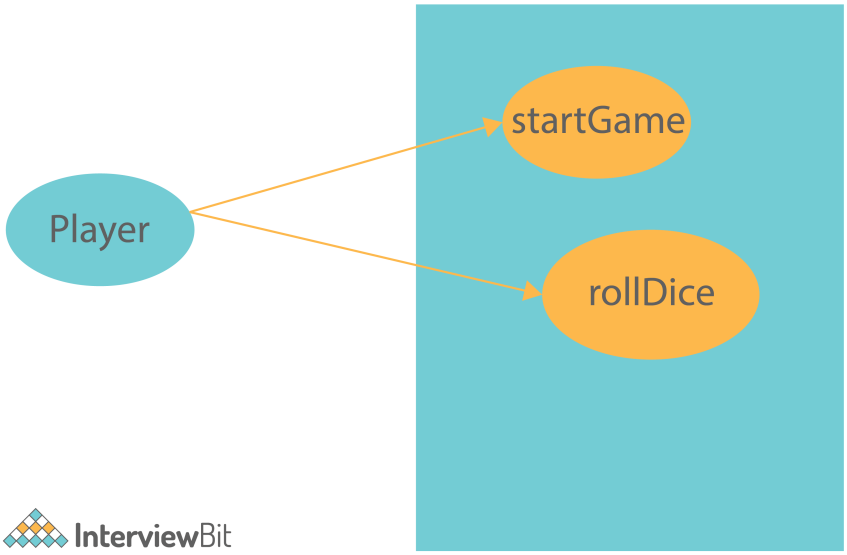
- The game can be played by more than one dice. (i.e. if there are two dices then the numbers from 2 to 12 will be generated).
- On getting a 6, you get another turn and on getting 3 consecutive 6s, all three of those get cancelled.

2. Class Diagram & Use Case Diagram:



UML CONVENTIONS





3. Code Implementation:

Driver.java

```
package snakeLadder;


public class Driver {

    public static void main(String[] args)
    {
        SpecialEntity snake1 = new Snake(12, 28);
        SpecialEntity snake2 = new Snake(34, 78);
        SpecialEntity snake3 = new Snake(6, 69);
        SpecialEntity snake4 = new Snake(65, 84);

        SpecialEntity ladder1 = new Ladder(24, 56);
        SpecialEntity ladder2 = new Ladder(43, 83);
        SpecialEntity ladder3 = new Ladder(3, 31);
        SpecialEntity ladder4 = new Ladder(72, 91);

        Board board = new Board(10);
        board.addSpecialEntity(snake1);
        board.addSpecialEntity(snake2);
        board.addSpecialEntity(snake3);
        board.addSpecialEntity(snake4)

        board.addSpecialEntity(ladder1);
        board.addSpecialEntity(ladder2);
        board.addSpecialEntity(ladder3);
        board.addSpecialEntity(ladder4);

        Dice dice = new Dice(6);

        Game game = new Game(board, dice);

        Player player1 = new Player("p1");
        Player player2 = new Player("p2");
        Player player3 = new Player("p3");

        players = List<Player>(){player1, player2, player3};
        game.addPlayers(players);

        game.launch();
    }

}
```

Game.java

```
package snakeLadder;

import snakeLadder.GameStatus;
import snakeLadder.Player;
import snakeLadder.Dice;
import snakeLadder.Board;

import java.util.Queue;
import java.util.Scanner;
import java.util.LinkedList;

public class Game {

    Board board;
    Dice dice;
    Queue<Player> players;
    GameStatus staus;

    public Game(Board board, Dice dice)
    {
        this.board = board;
        this.dice = dice;
        this.players = new LinkedList<Player>();
        this.status = GameStatus.NOT_STARTED;
    }

    public void startGame()
    {
        this.status = GameStatus.RUNNING;
        board.printBoard();

        // Run until we have only 1 player left on the board
        while(players.size() > 1)
        {
            Player player = players.poll();

            makeMove(currPlayer);

            if(player.getPosition() == board.getTotalCells())
                System.out.println(player.getName() + " has completed the game!");
            else
                players.add(player);
        }

        this.status = GameStatus.FINISHED;

    }

    private void makeMove(Player player) {

        System.out.println();
        System.out.println(currPlayer.getUserName()+"'s turn.");
        System.out.println("Press anything to roll the dice.");

        Scanner sc = new Scanner(System.in);
        char c = sc.next().charAt(0);
```

Board.java

```
package snakeLadder;

import lombok.Getter;

public class Board{

    @Getter
    int dimension;

    HashMap<Integer, SpecialEntity> specialEntities;

    public Board(int dimension)
    {
        this.dimension = dimension;
    }

    public void printBoard()
    {
        int totalCells = dimension*dimension;
        for(int i=totalCells; i > 0; i--)
        {
            System.out.print(" | " + i + " ")

            if(hasSpecialEntity(i))
                System.out.print(specialEntities.get(i).getID());

            System.out.print(" |");
            if(totalCells % 10 == 0)
                System.out.println();
        }
    }

    public int getTotalCells()
    {
        return this.dimesion * this.dimesion;
    }

    public void addSpecialEntity(SpecialEntity entity)
    {
        int actionPosition = entity.getActionPosition();

        specialEntities.put(actionPosition, entity);
    }

    public boolean hasSpecialEntity(int position)
    {
        return specialEntities.containsKey(position);
    }

    public SpecialEntity getSpecialEntity(int position)
    {
        if(hasSpecialEntity(position))
            return specialEntities.get(position);

        return null;
    }
}
```

Dice.java

```
package snakeLadder;

import lombok.Getter;

public class Dice{

    int maxValue;

    public Dice(int maxVal)
    {
        this.maxValue = maxVal;
    }

    public int roll()
    {
        return (int) Math.floor(Math.random()*maxValue + 1);
    }
}
```

GameStatus.java

```
package snakeLadder;

public enum GameStatus {
    NOT_STARTED,
    RUNNING,
    FINISHED
}
```

SpecialEntity.java

```
package snakeLadder;

import lombok.Getter;
public abstract class BoardEntity {

    private int start;
    private int end;

    public SpecialEntity(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public abstract String getID();

    public int getActionPosition()
    {
        return this.start;
    }

    public int getEndPosition()
    {
        return this.end;
    }
}
```

Snake.java

```
package snakeLadder;

public class Snake extends SpecialEntity{

    public Snake(int start, int end) {
        super(start, end);
    }

    @Override
    public String getID() {
        return "S_"+ this.getEnd();
    }
}
```

Ladder.java

```java
package snakeLadder;

public class Ladder extends SpecialEntity{

    public Ladder(int start, int end) {
        super(start, end);
    }

    @Override
    public String getID() {
        return "L_"+ this.getEnd();
    }
}
```

Player.java

```java
package snakeLadder;

import lombok.Getter;
import lombok.Setter;

public class Player{

    @Getter
    @Setter
    int position;

    @Getter
    String name;

    public Player(String name)
    {
        this.name = name;
        this.position = 0;
    }
}
```

GameAlreadyStartedException.java

```
package snakeLadder;

public class GameAlreadyStartedException extends Exception {
    public GameAlreadyStartedException(String message) {
        super(message);
    }
}
```

It is important to structure the code into different classes with each class having a single responsibility (Remember S in SOLID Principles) and also having relations between the classes so that they can be easily extensible.

Some frequently asked low-level design examples to practice are:

- Design Parking Lot
- Design Splitwise
- Design Tik Tok Toe Game
- Design Car Rental System
- Design Bookmyshow
- Design Pub Sub System
- Design Coffee Vending Machine

Tips on Cracking the Low Level Design (LLD) Interview

5. Interview Preparation Tips

- **Don't be in hurry:** Once the requirements are listed, there may be too many requirements to consider and the 40 or 45 minutes may not be enough to design, and code. Don't be in a hurry to complete the design, and code for all of the requirements. Discuss with the interviewer and mention only the core requirements that you want to consider for the design. It is better to stand and drink water than drink milk by running.
- **Practice:** Take out a low-level design problem and try to solve them all by yourself. Gather requirements (be your judge here), create class & schema diagrams and write code that is extensible, reusable, and maintainable. Once you complete solving the problem, look at different solutions for the same problem out there on the internet and improve the weakness. Practice as many problems as you can before the interview. If you are lucky, you may even get the problem you already solved in the interview. If not, you already have a thought process developed to solve any new problem. Win-Win situation
- **Plan for each stage:** Unfortunately, it is not enough to solve the given problem, we also need to complete it within the time limit. As, we have different stages in solving the low-level design interview question like gathering requirements, class diagram, schema diagram, code, test, etc. Plan on how much percentage of interview time you want to spend on each stage. The more you practice, you may even analyze the time you are taking at each stage and check where you need to improve.
- **Write Testcases & Handle Exceptions:** Not every interviewer expects you to write the test cases for the code. But if you can go a step further and write down the test cases, it gives you a huge advantage over other candidates. Gracefully handle exceptions and other corner cases.
- **Get comfortable with tools:** As we may need to draw class diagrams, schema diagrams, use case diagrams, etc, design different components and their interaction between them, it is common that companies schedule their design interviews with whiteboard links & code editor. It may be difficult to use the whiteboard apps/ code editors without getting accustomed to them. So get comfortable with these tools before the interview so that you don't have to waste time figuring out how to do things with these tools.

Conclusion

In this article, we discussed what Low-Level Design is and how it is different from High-Level Design interviews. Low-Level Design focuses on the class-level design of an application & clean, readable, maintainable, and extensible code. No software is built for fixed requirements and as the software evolves there will be new requirements and hence the software needs to be changed. Change is the only constant in software development. Hence the software needs to have a good design. The key to cracking Low-Level Design interviews is to understand the importance & use of Object Oriented Principles and Design Patterns in the development of real-world applications and how these concepts make the application readable, maintainable, and extensible.

Clearly understand the low-level design problem, jot down the requirements of the problem, fine-tune the requirements and pick the core requirements that can be implemented in the given time, make note of different entities with their behaviours & actors from the core requirements, create the use case diagram and class diagram, make the schema diagram if the problem statement requires a type of web application to be built (no need to create a web application in an interview but the code can be structured in that way by creating controllers, services, and repositories) and then finally implementing the core features by using Object Oriented Principles and Design Patterns to make the application extensible and readable.

Practice as many low-level design interview questions as possible as each problem gives a different perspective and learn how to apply Object Oriented Principles and Design Patterns to solve the problems.

Interview Resources

- System Design Interview Questions
- Software Engineering Interview Questions
- Best System Design Courses
- Design Patterns Interview Questions
- Java Interview Questions
- OOPs Interview Questions
- Interview Preparation Resources

Links to More Interview Questions

C Interview Questions	Php Interview Questions	C Sharp Interview Questions
Web Api Interview Questions	Hibernate Interview Questions	Node Js Interview Questions
Cpp Interview Questions	Oops Interview Questions	Devops Interview Questions
Machine Learning Interview Questions	Docker Interview Questions	Mysql Interview Questions
Css Interview Questions	Laravel Interview Questions	Asp Net Interview Questions
Django Interview Questions	Dot Net Interview Questions	Kubernetes Interview Questions
Operating System Interview Questions	React Native Interview Questions	Aws Interview Questions
Git Interview Questions	Java 8 Interview Questions	Mongodb Interview Questions
Dbms Interview Questions	Spring Boot Interview Questions	Power Bi Interview Questions
Pl Sql Interview Questions	Tableau Interview Questions	Linux Interview Questions
Ansible Interview Questions	Java Interview Questions	Jenkins Interview Questions