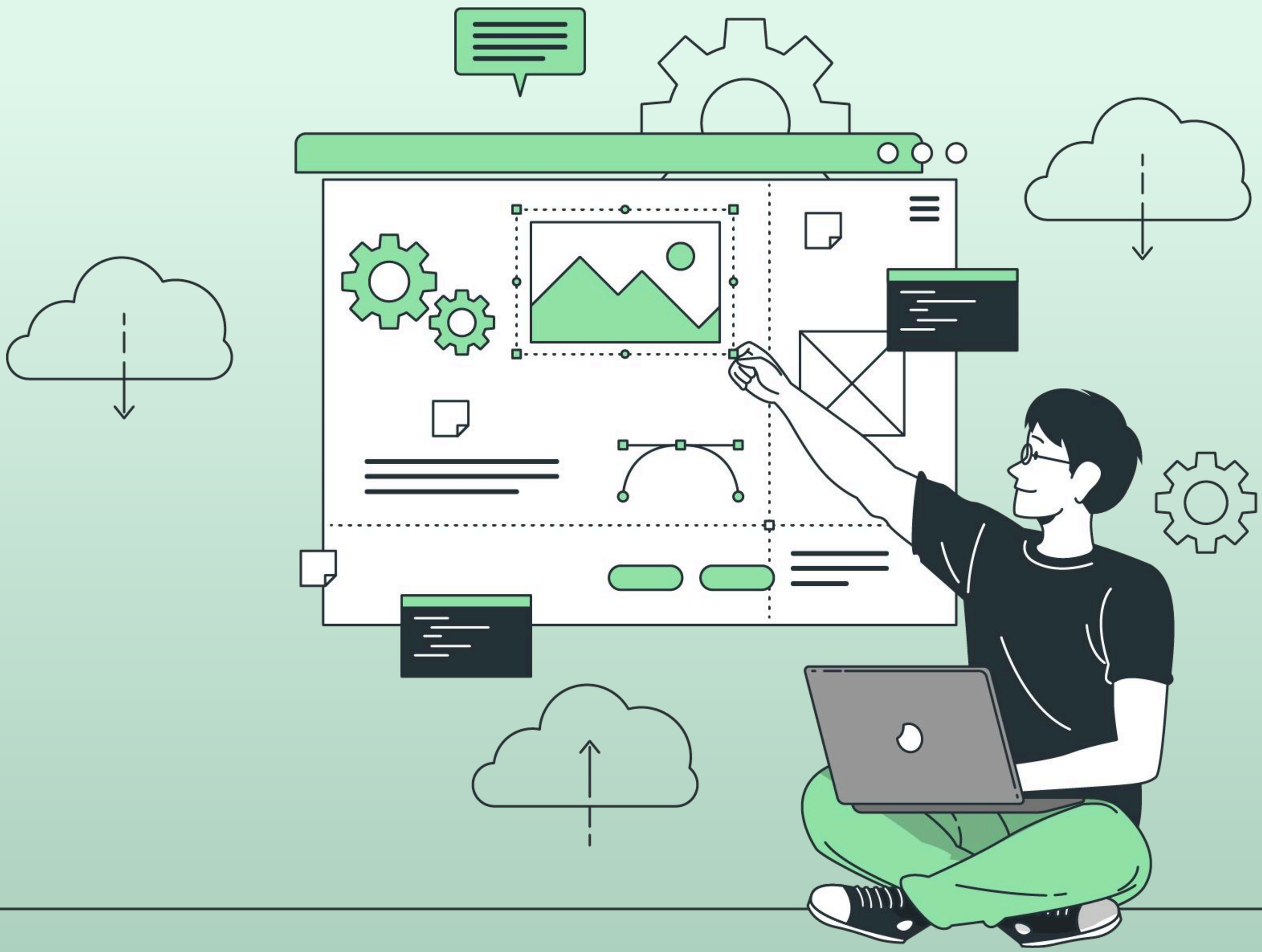
Tutort ACADEMY

CRACK LOW LEVEL DESIGN INTERVIEWS

BY MASTERING THESE **KEY QUESTIONS**



1 What is System design?

Software is typically created to address a specific business issue. Identifying the business issue, gathering the functional requirements for the identified issue, designing the overall architecture of the software/ system by defining the building blocks known as components, designing the individual components, actually writing the code, testing the software, releasing the software, and maintaining the software are the steps that make up the process of developing any software. This process is known as system design.

It is categorised in two parts:

High-Level Design (HLD)

A system's proposed architecture is described in high-level design (HLD). The architecture diagram gives a comprehensive picture of a system by highlighting the key elements that would be created for the final product and their interfaces. The HLD may employ simple to moderately complex words that the system administrators should be able to comprehend.

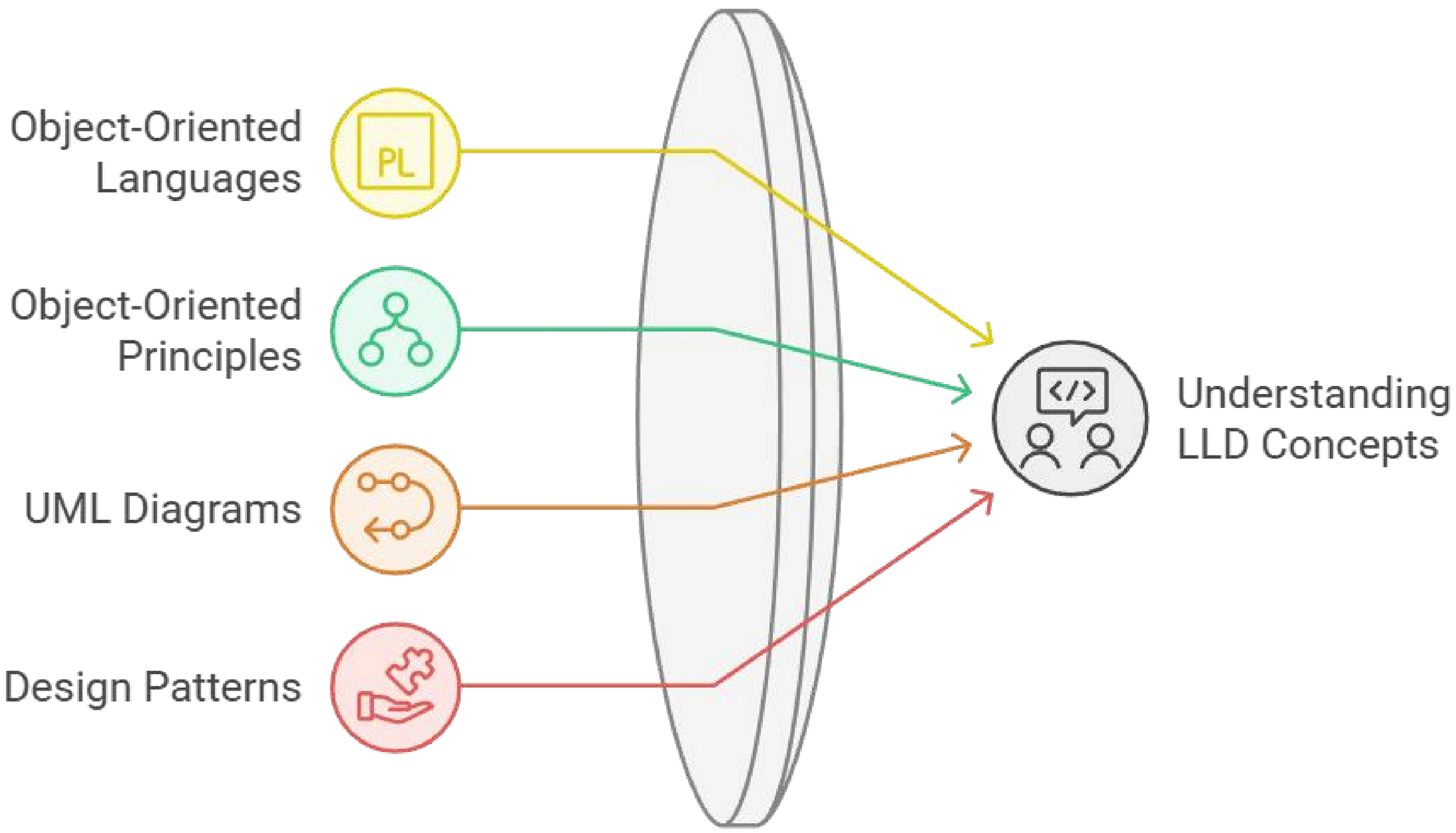
Low-Level Design (LLD)

LLD explains class diagrams using methods, relationships between classes, and program specifications. In order for the programmer to create the program directly from the document, it describes the modules. Due to the fact that various things have various character sets, it gives us the structure and behavior of a class.

2 What is the significance of low-level design?

A software is developed using low level design 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.

The following is a list of knowledge a software developer should have in order to understand LLD concepts:



3 Explain LLD Concepts.

Classes

Blueprints for creating objects with defined properties and methods.

Inheritance

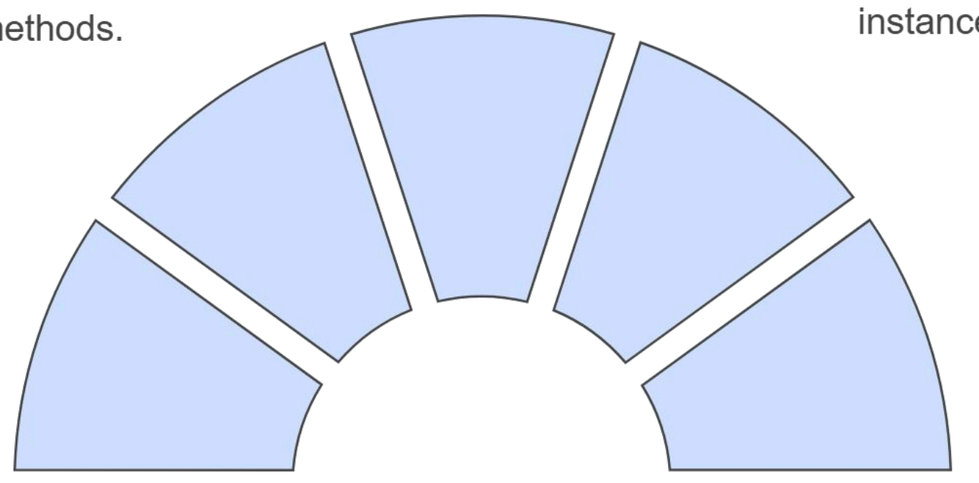
Mechanism for creating new classes based on existing ones.

Polymorphism

Ability of different classes to be treated as instances of the same class.

Objects

Fundamental units representing real-world entities in programming.



Abstraction

Simplifying complex systems by modeling classes based on essential qualities.

i **Object-Oriented Language(s)**: Object-oriented programming is a programming paradigm based on the concept of objects, which can contain data and code: data in the form of fields, and code in the form of procedures. A common feature of objects is that methods are attached to them and can access and modify the object's data fields. 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.

ii **Object-Oriented Principles** : Object-Oriented Programming Principle is the strategy or style of developing applications based on objects. Anything in the world can be defined as an object. And in the OOPs, it can be defined in terms of its properties and behaviour. Every operation that is going to be functional is considered in terms of classes and objects. That provides a better programming style because you need not write code that needs to run anytime. Instead, you can create classes defining the functionality and call that function by creating an object of that.

Some of the well-known principles are :

- **YAGNI**: YAGNI principle ("You Aren't Gonna Need It") is a practice in software development which states that features should only be added when required. As a part of the extreme programming (XP) philosophy, YAGNI trims away excess and inefficiency in development to facilitate the desired increased frequency of releases.

- **DRY**: "Don't repeat yourself" (DRY) is a principle of software development aimed at reducing repetition of information which is likely to change, replacing it with abstractions that are less likely to change, or using data normalization which avoids redundancy in the first place.

- **SOLID**: These are sets of 5 principles(rules) that are strictly followed as per requirements of the system or requirements for optimal designing.
  In order to write scalable, flexible, maintainable, and reusable code:

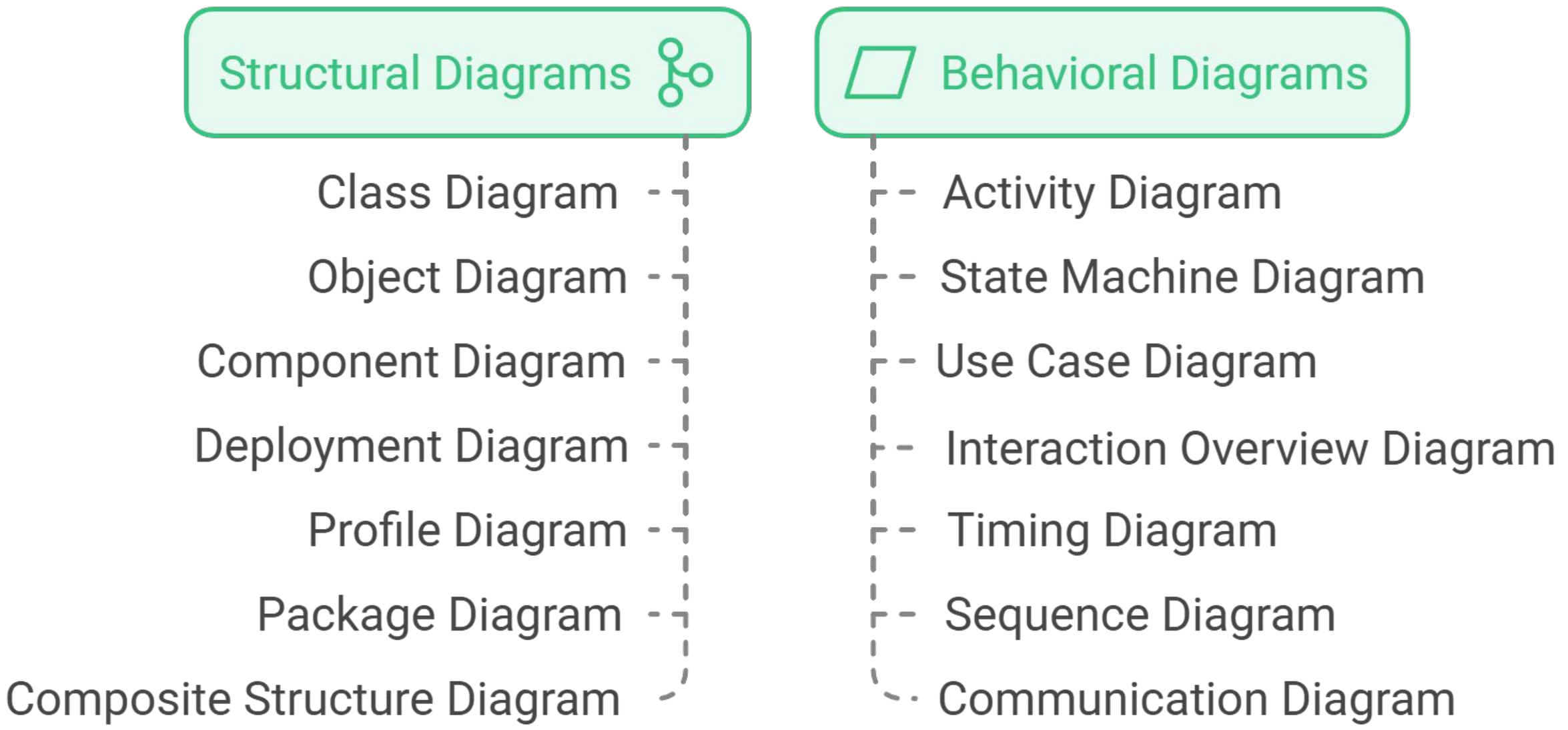
  1) Single-responsibility Principle (SRP)
  2) Open-closed Principle (OCP)
  3) Liskov's Substitution Principle (LSP)
  4) Interface Segregation Principle (ISP)
  5) Dependency Inversion Principle (DIP)

iii **UML Diagrams**: UML Stands for unified modeling language. It is not a programming language but a tool used for modeling, visualizing, and creating the software system.

There are 2 types of UML Diagrams:

- **Structural UML diagram**: These types of diagrams basically defines how different entities and objects will be structured and defining the relationship between them. They are helpful in representing how components will appear with respect to structure. These include the static view of the system

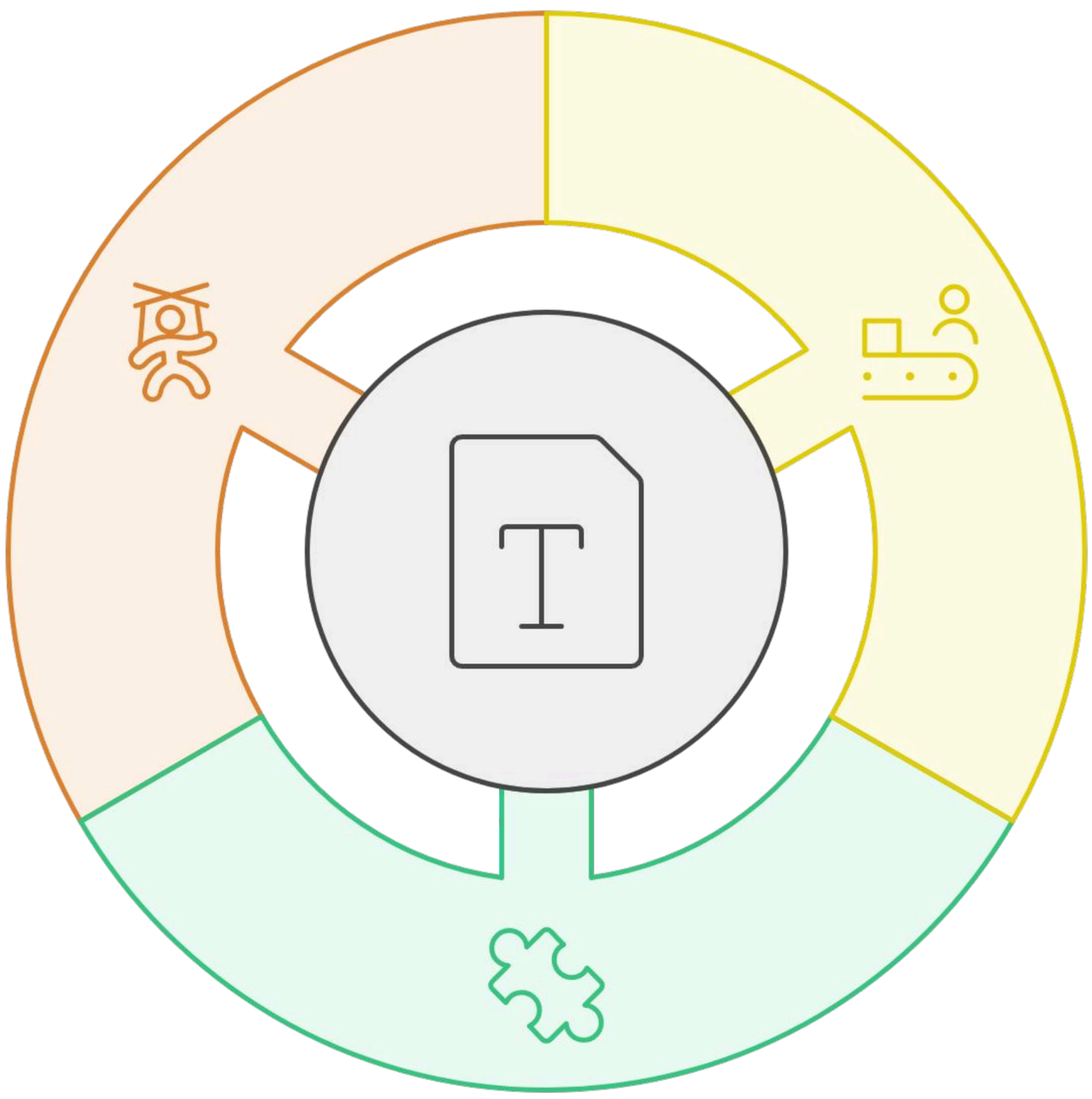
- **Behavioral UML diagram:** These types of diagrams basically defines what are the different operations that it supports. These include the dynamic view of the system.



iv **Design Patterns**: Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Design Pattern is mainly categorized in 3 parts:

Behavioral Patterns
Concern with object interaction and responsibility.



Creational Patterns
Focus on object creation mechanisms.

Structural Patterns
Deal with object composition and structure.

- **Creational Patterns:** Class instantiation is the central theme of these design patterns. This pattern can be further broken down into patterns for creating classes and patterns for creating objects. While class-creation patterns successfully employ inheritance throughout the instantiation process, object-creation patterns successfully use delegation.

- **Structural Patterns:** These design patterns are centered on the composition of classes and objects. Inheritance is used by structural class-creation patterns to create interfaces. Object composition techniques using structural object patterns are described in order to create new functionalities.

- **Behavioral Patterns:** The communication between Class's objects is the focus of these design patterns. The behavioral patterns that are most specifically focused on inter-object communication.

4 What is the difference between a sequence diagram and an activity diagram?

- A sequence diagram represents the objects to visualize the flow of the design, but an activity diagram represents all the cases and flows in a single diagram.

- A Sequence diagram gives the overview of the whole system, whereas an activity diagram Gives a holistic picture of the system.

- In the case of a movie booking system, the sequence diagram includes a booking controller, a booking manager, and a payment processor. But, an activity diagram includes the detailed process, from the display of a list of movies and theaters to the reservation of seats.

5 What are the most widely used design patterns?

The most globally used design patterns are:

- **Factory pattern**: Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

- **Abstract Factory Pattern**: Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

- **Singleton Pattern**: Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
  This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

- **Observer Pattern**: Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

6 How to tackle LLD problems in interviews?

1) Understand the Problem:

Ask clarifying questions to ensure you have a clear understanding of the problem statement.
Identify the key requirements, constraints, and any specific technologies or frameworks you should consider.

2) Gather Requirements:

List down the functionalities and features that the component should support. Consider performance, scalability, reliability, and other non-functional requirements.

3) Identify Classes/Modules:

Start by breaking down the component into smaller classes or modules. Consider the Single Responsibility Principle (SRP) and other design principles.

4) Define Class Interfaces:

For each class/module, specify the public methods and attributes. Think about what information each class needs to perform its tasks.

5) Handle Data:

Determine how data will be stored, retrieved, and manipulated. This might involve database design or in-memory data structures.

6) Define Relationships:

Identify how different classes/modules interact with each other. Consider composition, inheritance, or other forms of relationships.

7) Consider Design Patterns:

Utilize design patterns where appropriate (e.g., Singleton, Factory, Observer, etc.) to solve common design problems.

Akash Shrivastava

From NUANCE → To Microsoft

MAANG

8) Code:

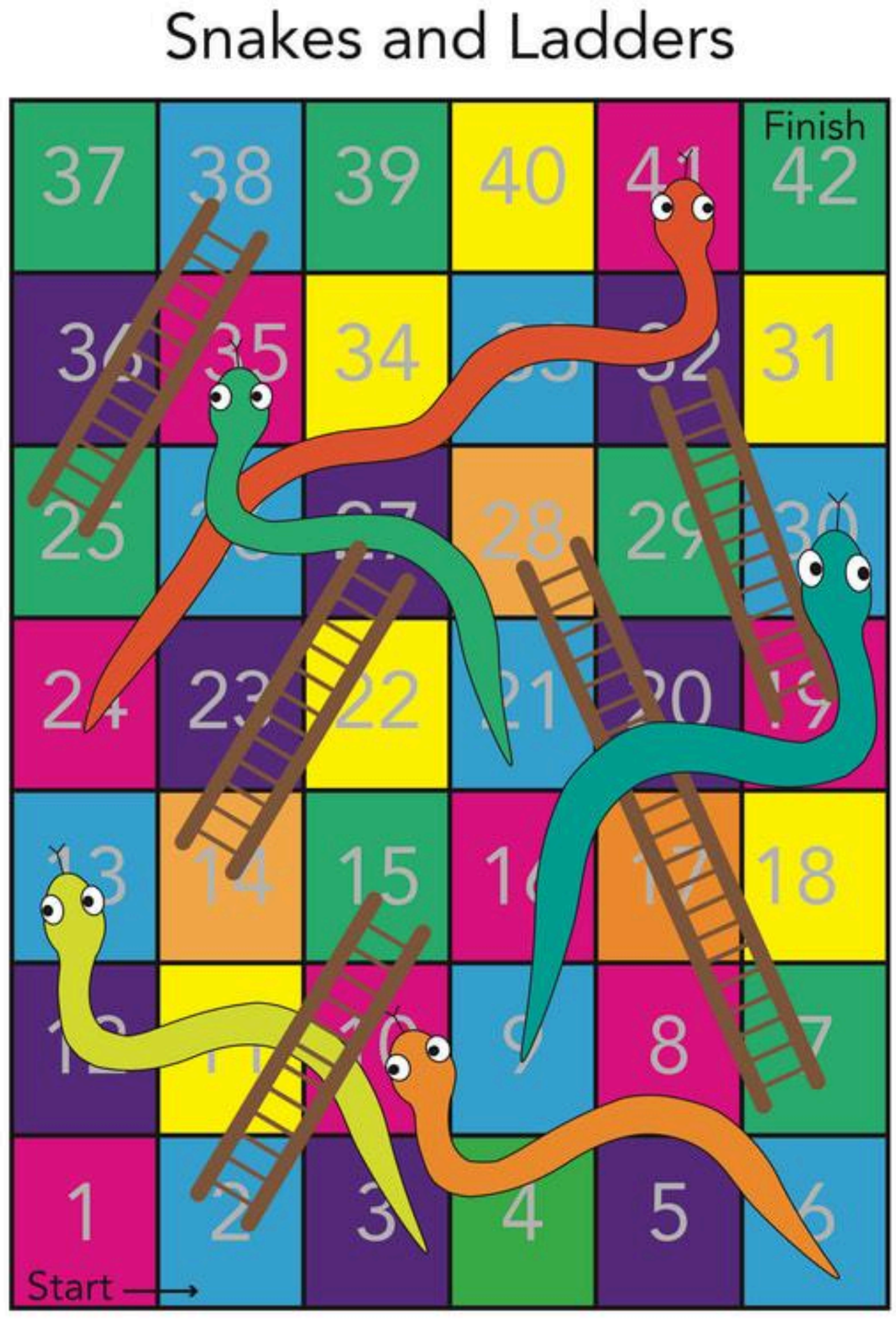
Finally once the thoughts are structured using Class Diagram, Use Case Diagram, and Schema Diagram (if required), the Candidate can start writing the code. Apply the Design Patterns, Object-Oriented Principles & SOLID Principles wherever is possible to make the system reusable, extensible, and maintainable. 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.

9) Practice, Practice, Practice:

Solve LLD problems in mock interviews or on platforms like LeetCode, HackerRank, or GeeksforGeeks to build your skills.

7 Design Snake & Ladder game.

1 UNDERSTAND THE PROBLEM



2 GATHER 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.

3 IDENTIFY CLASSES:

Board:

This class represents the game board and contains the layout of the snakes and ladders.

Player:

Represents a player in the game. It stores the current position of the player on the board.

Dice:

Simulates the rolling of a dice. It should have a method to roll the dice and return a random number between 1 and 6.

Game:

Orchestrates the game, managing player turns, movements, and checking for game over conditions.

Snake:

Represents a snake on the board. It has a start position and an end position.

Ladder:

Represents a ladder on the board. It has a start position and an end position.

4 DEFINE CLASS INTERFACES/ METHODS:

Board:

- addSnake(Snake snake): Adds a snake to the board.
- add Ladder(Ladder ladder): Adds a ladder to the board.

Player:

- move(int steps):
  Moves the player by the given number of steps.
- getPosition(): Returns the current position of the player.

Dice:

- roll(): Returns a random number between 1 and 6.

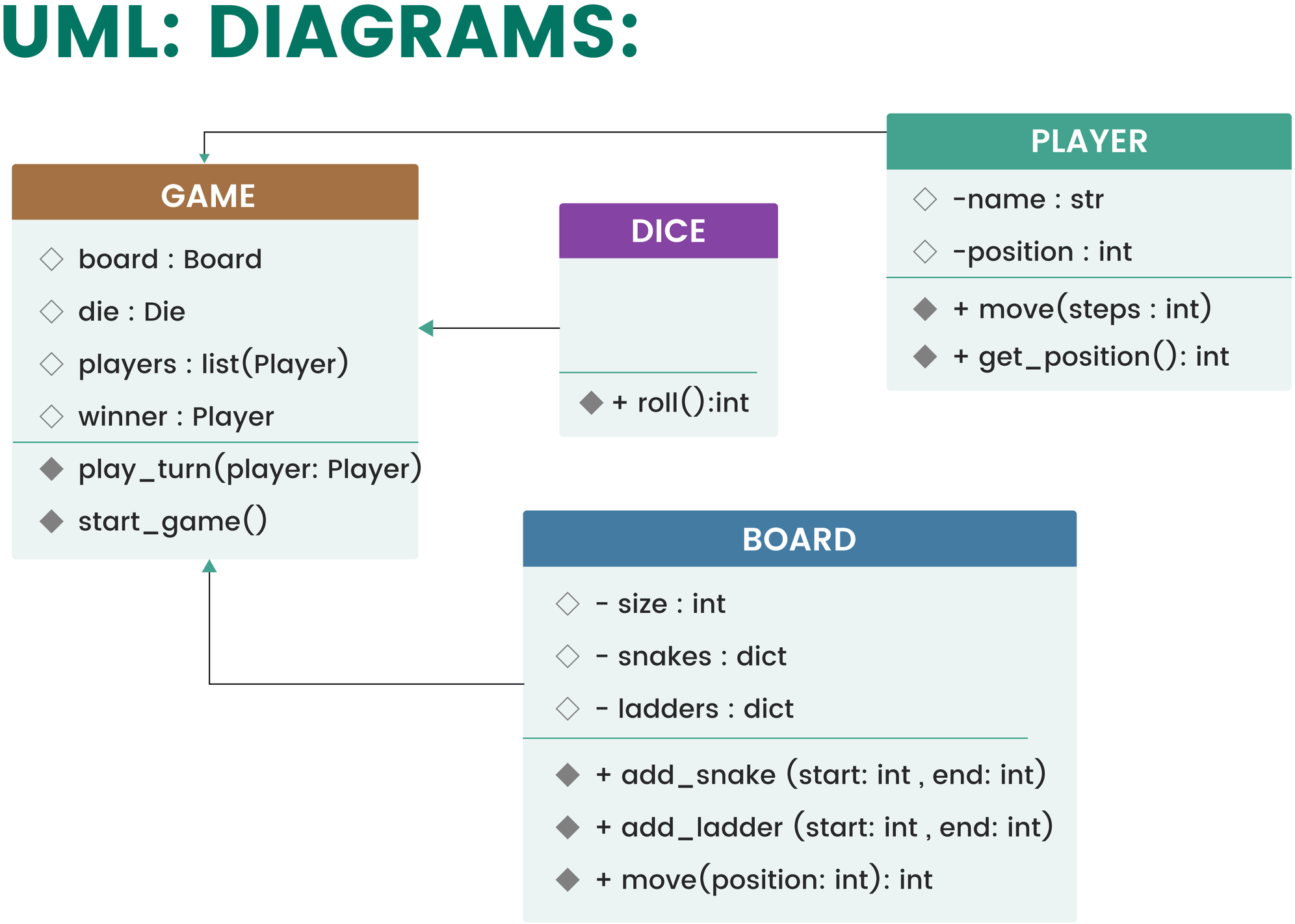
Game:

- startGame(): Starts the game loop.
- endGame(): Ends the game.
- playTurn(Player player):
  Manages a player's turn, including rolling the dice and moving the player.

5 GAME FLOW:

- Initialize the board and add snakes and ladders.
- Create players and place them at the starting position.
- Start the game loop:
  - For each player's turn, roll the dice and move the player accordingly.
  - Check for snakes or ladders at the new position and adjust the player's position if needed.
  - Check if a player has won (reached the last position).

6 UML: DIAGRAMS:



7

CODE:

Below is a basic Python implementation of the Snake and Ladder game.

class Board:

```
- def __init__(self, size):
- self.size = size
-    self.snakes = {}
-    self.ladders = {}
- def add_snake(self, start, end):
-    self.snakes[start] = end
- def add_ladder(self, start, end):
-    self.ladders[start] = end
- def move(self, position):
- if position in self.snakes:
-    return self.snakes[position]
-    elif position in self.ladders:
- return self.ladders[position]
- return position
```

class Dice:

```
- def roll(self):
  return random.randint(1, 6)
```

class Player

```
def __init__(self, name):
  self.name = name
  self.position = 1
def move(self, steps):
  self.position += steps
def get_position(self):
  return self.position
```

class Game:

```
def __init__(self, board, dice, players):
self.board = board
self. dice = dice
self.players = players
self.winner = None

def play_turn(self, player):
steps = self. dice.roll()
player.move(steps)
player.position = self.board.move(player.position)
if player.position == self.board.size:
self.winner = player

def start_game(self):
while not self.winner:
for player in self.players:
self.play_turn(player)
print(f"{player.name} rolled a {player.get_position()}")
print(f"{self.winner.name} wins!")
```

```
if __name__ == "__main__":
    # Create board and add snakes/ladders
    board = Board(100)
    board.add_snake(16, 6)
    board.add_snake(47, 26)
    board.add_snake(49, 11)
    board.add_ladder(3, 22)
    board.add_ladder(5, 8)
    board.add_ladder(20, 24)
    # Create players
    player1 = Player("Player 1")
    player2 = Player("Player 2")
    # Create game
    game = Game(board, Dice(), [player1, player2])
    # Start game
    game.start_game()
```

More LLD interview questions to practice:

- Parking lot design (Solution Reference)
- Design library management system
- Google docs design
- Design a web crawler
- Design social media platforms
- Design e-commerce sites
- v Design Youtube

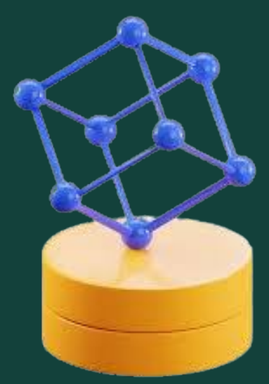
Why Tutort Academy?

1250+ Career Transitions	**350+** Hiring Partners	**2.1CR** Highest CTC

Tutort ACADEMY

Start Your Upskilling with us

Explore our courses



Data Structures and Algorithms with System Design

Full Stack Specialisation In Software Development

www.tutort.net

Watch us on Youtube

Read more on Quora

Follow us on

LinkedIn Instagram YouTube

Phone
+91-8712338901

E-mail
contact@tutort.net