LOW LEVEL DESIGN

College Placement Management System

Written By	Rudrayani Sakhare
Document Version	0.2
Last Revised Date	16-05-2024

Document Control

Change Record:

Version	Date	Author	Comments
0.1	15-March-2024	Rudrayani	Introduction & Architecture defined , Architecture & Architecture Description appended
0.2	16-May-2024	Rudrayani	Architecture & Architecture Description appended and updated

Review:

Version	Date	Reviewer	Comments
0.1	26 -April-2024	Ishita Yadav	Add more on Architecture

1. Introduction

1.1. What Is A Low Level Document ?

A low-level document, often referred to as a Low-Level Design (LLD) document, functions as a detailed blueprint for programmers. It delves into the specifics of how a system will be constructed, translating the broader goals of a high-level design into the nitty-gritty of the code. Here's a breakdown of its key characteristics:

- **Focus on Internal Design:** An LLD concentrates on the internal workings of a system. It outlines the specific components that will be built, the algorithms chosen to achieve functionality, and the data structures that will hold information. This detailed roadmap provides programmers with a clear understanding of how to translate the system's requirements into code.
- **Detailed and Actionable:** In contrast to a high-level document that offers a broad overview, an LLD dives into the specifics. It outlines exactly how different parts of the system will interact and function, including details like method specifications within classes and program specifications. This level of detail allows programmers to begin coding directly from the document, minimizing the need to interpret a more abstract big-picture view.
- **Benefits Development:** A well-crafted LLD document is a valuable asset during development. By providing a clear and detailed roadmap, it promotes efficient coding. Programmers can spend less time deciphering requirements and more time writing functional code. Additionally, an LLD can minimize debugging and testing cycles by catching potential issues early in the development phase.

1.2. Scope

The scope of a Low-Level Design (LLD) document is primarily focused on the internal workings of a specific software system. It translates the high-level goals and functionalities into the technical details that programmers need to implement the system. Here's a breakdown of the key areas covered in an LLD:

- **Components and their interactions:** An LLD outlines the individual software components that make up the system. This includes modules, classes, functions, and how they interact with each other to achieve the desired outcome.
- **Data structures and algorithms:** The LLD specifies the data structures chosen to store and manage information within the system. It also details the algorithms used to process data and perform specific tasks.
- **Detailed design choices:** The document dives deep into the technical details, like method specifications within classes and program specifications. This level of detail allows programmers to begin coding directly from the LLD.
- **Focus on functionality, not user interface:** While an LLD might mention interactions with the user interface (UI), its primary focus is on the internal functionalities of the system, not the way users will interact with it.

In essence, the LLD acts as a bridge between the broad concepts of a high-level design and the actual code written by programmers. It ensures everyone involved in the development process has a clear understanding of how the system will be built and function.

2. Architecture

2.1 Component Design:

Dataset overview:
Total rows: 13

	Column Name	Data Type	Mean	Std	Min	Max	Categories
0	Unnamed: 0	int64	107	62.21	0	214	N/A
1	secondary_percent	float64	67.30	10.83	40.89	89.4	N/A
2	secondary_branch	object	N/A	N/A	N/A	N/A	['Others', 'Central']
3	highschool_percent	float64	66.33	10.90	37	97.7	N/A
4	high_school_branch	object	N/A	N/A	N/A	N/A	['Others', 'Central']

Overview
The dataset sample_dataset.csv contains 215 rows and 13 columns, capturing various educational and employment metrics. Key components include secondary_percent, highschool_percent, degree_percent, and employment_test_percent.

Components and Their Functions

1. **Secondary Percent (**secondary_percent**)**
    - **Function**: Measures the percentage score obtained in secondary education.
    - **Problem Solved**: Provides a quantitative measure of academic performance at the secondary level.
    - **Operations**:
        - **Mean Calculation**: Average score (67.30).
        - **Standard Deviation**: Variability in scores (10.83).
        - **Range**: Minimum (40.89) and maximum (89.40) scores.
    - **Interactions**: Used in conjunction with other academic metrics to assess overall educational background.

2. **High School Percent (**highschool_percent**)**
    - **Function**: Measures the percentage score obtained in high school.
    - **Problem Solved**: Offers insight into academic performance at the high school level.
    - **Operations**:
        - **Mean Calculation**: Average score (66.33).
        - **Standard Deviation**: Variability in scores (10.90).
        - **Range**: Minimum (37.00) and maximum (97.70) scores.
    - **Interactions**: Often compared with secondary and degree percentages to evaluate academic progression.

3. **Degree Percent (**degree_percent**)**
    - **Function**: Measures the percentage score obtained in undergraduate degree.
    - **Problem Solved**: Indicates academic performance at the tertiary education level.
    - **Operations**:
        - **Mean Calculation**: Average score (66.37).
        - **Standard Deviation**: Variability in scores (7.36).
        - **Range**: Minimum (50.00) and maximum (91.00) scores.
    - **Interactions**: Combined with other educational metrics to assess readiness for employment or further education.

4. **Employment Test Percent (**employment_test_percent**)**
    - **Function**: Measures the percentage score obtained in an employment test.
    - **Problem Solved**: Evaluates the candidate's aptitude and readiness for employment.
    - **Operations**:
        - **Mean Calculation**: Average score (72.10).
        - **Standard Deviation**: Variability in scores (13.28).
        - **Range**: Minimum (50.00) and maximum (98.00) scores.
    - **Interactions**: Used alongside academic scores to predict employment outcomes.

Interactions and Dependencies

- **Interfaces**: These components interact with other columns like status (employment status) and specialisation to provide a comprehensive view of a candidate's profile.
- **Dependencies**: Relies on accurate data entry and preprocessing steps. If dependencies like data preprocessing libraries (e.g.,

pandas) are unavailable, data analysis and operations will be hindered.
- **Performance Considerations**: Efficient handling of statistical operations to ensure quick insights.
- **Error Handling**: Mechanisms to handle missing or erroneous data, such as imputation or exclusion.

External Libraries and APIs
- **Libraries**: Likely uses pandas for data manipulation and numpy for statistical calculations.
- **APIs**: May interact with visualization tools (e.g., matplotlib) for graphical representation of data.

Error Handling and Performance
- **Error Handling**: Includes checks for missing values, outliers, and data type mismatches.
- **Performance**: Optimized data processing to handle large datasets efficiently.

Conclusion
These components collectively provide a detailed view of a candidate's academic and employment readiness, crucial for educational and employment analytics. Their interactions and dependencies ensure comprehensive and accurate data analysis, essential for informed decision-making.

2.2 Data Structures And Algorithms:

1. **Data Structures**
    - **Arrays and Lists**: For storing and accessing student records efficiently.
    - **Dictionaries**: For quick lookups and mapping student IDs to their respective records.
    - **DataFrames (Pandas)**: For handling large datasets, performing complex queries, and data manipulation.

2. **Algorithms**
    - **Sorting Algorithms**: QuickSort or MergeSort for ranking students based on their scores.
    - **Search Algorithms**: Binary Search for efficient retrieval of student records.
    - **Statistical Analysis**: Mean, median, and standard deviation calculations for performance analysis.

Justification

- **Efficiency**: DataFrames and dictionaries offer efficient data manipulation and quick lookups, which are essential for handling large datasets in CPMS.
- **Scalability**: The chosen data structures and algorithms can handle increasing amounts of data without significant performance degradation.
- **Suitability**: These structures and algorithms are well-suited for the operations required in a CPMS, such as filtering, sorting, and analyzing student data.

Dataset Components

1. **secondary_percent**: This component represents the percentage score obtained by a student in their secondary (10th grade) education. It helps in assessing the academic performance of students at an early stage.

2. **highschool_percent**: This component indicates the percentage score obtained by a student in their high school (12th grade) education. It is crucial for evaluating the academic consistency and preparedness of students for higher education.

3. **degree_percent**: This component shows the percentage score obtained by a student in their undergraduate degree. It is a key metric for understanding the student's performance in higher education, which is often a critical factor for campus placements.

4. **employment_test_percent**: This component represents the percentage score obtained by a student in an employment test, which

is typically conducted by companies during campus placements. It helps in gauging the employability and job readiness of students.

Functions and Problems Solved

- **secondary_percent**: Helps in early identification of academically strong students.
- **highschool_percent**: Assists in tracking academic progress and consistency.
- **degree_percent**: Provides insights into higher education performance, crucial for job placements.
- **employment_test_percent**: Evaluates job readiness and practical skills of students.

Operations and Interactions

- **Data Aggregation**: Combining scores from different educational stages to get a comprehensive view of a student's academic journey.
- **Filtering and Sorting**: Sorting students based on their scores to identify top performers for campus placements.
- **Statistical Analysis**: Performing statistical operations to understand trends and patterns in student performance.
- **Correlation Analysis**: Analyzing the correlation between different educational metrics and employment test scores to predict job readiness.

Dependencies

- **Data Integrity**: Accurate and consistent data entry is crucial for reliable analysis.
- **Data Privacy**: Ensuring that student data is handled with confidentiality and in compliance with data protection regulations.

Data Structures and Algorithms for CPMS

1. **Data Structures**:
    - **Arrays/Lists**: For storing and accessing student scores efficiently.
    - **Hash Tables**: For quick lookups and retrieval of student records.
    - **Trees (e.g., AVL Trees)**: For maintaining sorted order of student scores and enabling efficient range queries.
    - **Graphs**: For representing relationships between students and companies, facilitating efficient matchmaking.

2. **Algorithms**:
    - **Sorting Algorithms (e.g., QuickSort, MergeSort)**: For sorting student scores to identify top performers.
    - **Search Algorithms (e.g., Binary Search)**: For efficient retrieval of student records based on specific criteria.
    - **Machine Learning Algorithms**: For predictive analysis and identifying patterns in student performance data.
    - **Graph Algorithms (e.g., Dijkstra's, BFS/DFS)**: For optimizing placement processes and finding the best matches between students and companies.

Justification

- **Efficiency**: Data structures like hash tables and trees ensure quick access and manipulation of student records, which is essential for real-time placement processes.
- **Scalability**: Algorithms like sorting and search are optimized for handling large datasets, making them suitable for a CPMS with a growing number of students and companies.
- **Suitability**: Machine learning algorithms can provide valuable insights and predictions, enhancing the decision-making process in campus placements.

2.3 Detailed design choices:

1. Class and Method Specifications:

Class: Student

- **Attributes**:
    - name: String - student's name.
    - email: String - student's email address.
    - skills: List of Strings - student's skills.
    - gpa: Float - student's Grade Point Average.
- **Methods**:
    - register(name: str, email: str, skills: List[str], gpa: float) -> None:
        - Purpose: Registers a new student with the provided details.
        - Parameters: Name, email, skills, and GPA of the student.
        - Return: None.
    - updateProfile(newEmail: str, additionalSkills: List[str]) -> None:
        - Purpose: Updates the student's profile with new email and additional skills.

        - Parameters: New email and additional skills to be added.
        - Return: None.
    - Getters and setters for attributes name, email, skills, and gpa.

Class: PlacementSystem

- **Attributes**:
    - students_db: Dictionary - stores student information.
    - placements_db: Dictionary - stores placement details.
- **Methods**:
    - addStudent(student: Student) -> None:
        - Purpose: Adds a new student to the system.
        - Parameters: Student object.
        - Return: None.
    - placeStudent(student: Student, company: str) -> bool:
        - Purpose: Places a student in a specific company.
        - Parameters: Student object and company name.
        - Return: True if placement successful, False otherwise.

2. Program Specifications:

- The program structure includes classes for Student and PlacementSystem.
- The flow involves registering students, updating profiles, and placing students in companies.
- Interaction: Student objects are created, added to the system, and then placed in companies based on their skills and GPA.

3. Error Handling:

- InvalidDataInputError: Raised when incorrect data is provided during student registration or profile update.
- PlacementError: Raised when a student cannot be placed in a company due to skill mismatch.
- Mechanisms: Error messages displayed to users, logging errors for debugging, and graceful handling of exceptions.

4. Coding Conventions:

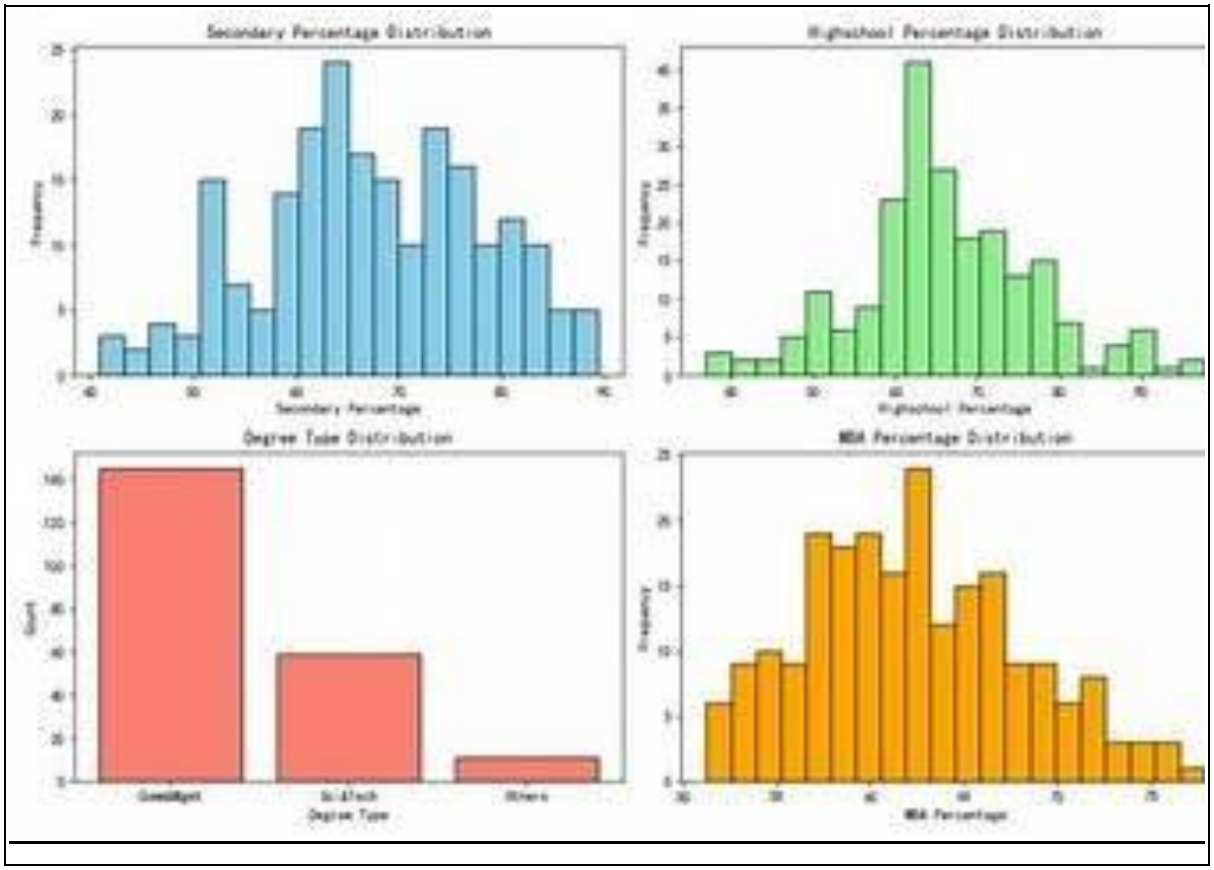
- Follow PEP 8 guidelines for Python coding style.
- Use meaningful variable and function names for clarity.
- Document code using docstrings for methods and classes.

5. Use Cases:

Use Case: Student Placement

- Scenario: A student applies for a job placement.
- Steps:
    1. The student registers with their name, email, skills, and GPA.
    2. The student updates their profile with additional skills.
    3. The PlacementSystem places the student in a company based on their skills and GPA.
- Interaction: Student class methods interact with PlacementSystem methods to facilitate the placement process.

2.4 Flowchart and diagrams:



Element 1: Secondary Percentage Chart

- **Purpose:** Displays the distribution of secondary school percentages in a histogram format.
- **Action:** Hovering over the bars will show the exact percentage and count of students.

Element 2: High School Percentage Chart

- **Purpose:** Shows the distribution of high school percentages in a histogram format.
- **Action:** Hovering over the bars will reveal the percentage and the number of students in that range.

Element 3: Degree Type Distribution Chart

- **Purpose:** Illustrates the count of students across different degree types in a bar chart.
- **Action:** Clicking on a bar filters the other charts to show data for the selected degree type.

Element 4: Employment_test Percentage Chart

- **Purpose:** Presents the distribution of MBA percentages in a histogram format.
- **Action:** Hovering over the bars provides detailed information about the MBA percentage distribution.

3. Architecture Description:

1. **Data Description**: The campus placement dataset contains information about students, including their academic performance, degree specialization, work experience, and placement status. It includes attributes such as secondary and high school percentages, degree type, MBA percentage, and other relevant metrics. This dataset provides insights into student profiles and their placement outcomes.
2. **Data Preprocessing**: In the data preprocessing phase, steps like handling missing values, encoding categorical variables, scaling numerical features, and splitting the data into training and testing sets will be performed. Additionally, data normalization, outlier detection, and feature engineering may be applied to prepare the dataset for model training.
3. **Data Clustering**: Data clustering involves grouping similar student profiles based on various features like academic performance, degree specialization, and work experience. Techniques like K-Means clustering can be utilized to segment students into clusters, enabling targeted analysis and personalized recommendations for career opportunities.
4. **Model Building**: Model building entails selecting appropriate algorithms like logistic regression, decision trees, or random forests to predict student placement outcomes. The dataset will be used to train and evaluate machine learning models to forecast the

likelihood of a student getting placed based on their academic and personal attributes.

5. **Deployment**: Once the model is trained and validated, it can be deployed to a production environment, such as a web application or API, to provide real-time placement predictions for students. Deployment on cloud platforms like AWS allows for scalability and accessibility, ensuring seamless integration of the predictive model into the campus placement system.