Scenario:

The Basketball Stats Tracker is developed to operate as a comprehensive system for recording and analyzing basketball player and game statistics. It helps coaches, players, and analysts track performance trends, make strategic decisions, and monitor improvements over time. For example, a coach can input stats for players after a game, such as points scored and rebounds, and use the application to identify high-performing players and areas for improvement. Game-specific stats, such as total fouls and turnovers, can also be monitored to gauge team performance.

Design Paradigm:

This application demonstrates the following functionalities:

1. **Adding Player Stats**:
   Users can input player details (name, points, assists, rebounds, blocks, etc.) based on their position (Guard, Forward, Center) using specific methods like addStats(points, assists, rebounds, blocks).

2. **Tracking Game Stats**:
   Users can input general game information (date, location, total score) and add individual or team stats for the game.

3. **Sorting and Filtering**:
   Players are sorted by points using Comparable and can be sorted by rebounds or assists using a Comparator. Streams allow filtering players based on criteria, such as those who scored above 20 points.

4. **Displaying Stats**:
   Stats for players and games are displayed using polymorphic methods like displayStats(), which provide customized outputs for different positions (e.g., highlighting assists for Guards).

5. **Data Persistence**:
   Stats can be saved to and loaded from text files using StatsFileManager.

6. **Error Handling**:
   Invalid inputs, such as negative stat values, are managed with custom exceptions like InvalidStatException.

Expected Output:

Users will be able to:

1. Input player and game statistics, categorized by player positions and team performance.
2. View sorted player lists by metrics, such as highest points or rebounds.
3. Filter players based on criteria, such as those with more than 10 assists or blocks.
4. Save all statistics to text files for persistence and reload them for continued analysis.
5. Handle invalid inputs with error messages (e.g., "Points cannot be negative").
6. Obtain tailored summaries for each position (Guard, Forward, Center) and overall team metrics.

Hierarchies:

1. **PlayerStats Hierarchy**:
    a. **Base Class**: PlayerStats
        - Shared attributes: name, points, assists, rebounds, turnovers.
        - Shared methods: addStats(), displayStats(), calculateAveragePoints().
    b. **Subclasses**:
        - GuardStats: Includes steals and assist-to-turnover ratio calculations.
        - ForwardStats: Includes field goal percentage and efficiency calculations.
        - CenterStats: Includes blocks and block rate calculations.

2. **GameStats Hierarchy**:
    a. **Base Class**: GameStats
        - Shared attributes: gameDate, location, totalScore.
        - Shared methods: displayStats(), addScore().
    b. **Subclasses**:
        - PlayerGameStats: Tracks individual player contributions in a game.
        - TeamGameStats: Tracks team-level stats like total fouls and turnovers.

Interface:

Name: StatsOperations

- **Purpose**:
    - Standardize core operations like displayStats() across all stats-related classes.
    - Enforce consistency and reusability in displaying stats for players and games.

Runtime-Polymorphism Methods:

1. **Method Overriding**:
    - displayStats() is defined in PlayerStats and overridden in GuardStats, ForwardStats, and CenterStats to provide position-specific outputs. For example, a GuardStats object highlights assists and steals, while a CenterStats object emphasizes blocks and rebounds.
    - Similarly, displayStats() is overridden in GameStats subclasses to display detailed game-specific stats.

2. **Method Overloading**:
    - calculateStats() in TeamGameStats is overloaded to compute metrics like team efficiency or scoring averages.

Text I/O:

Class: StatsFileManager

- **Purpose**:
    - Save player stats to a text file (stats.txt) for persistence. Example: Write PlayerStats objects with attributes like name, points, rebounds to the file.
    - Load saved stats from the text file into the application for analysis. Example: Read stats into an ArrayList<PlayerStats> and display them.

Comparable:

Implementation: The Comparable interface is implemented in the PlayerStats class to allow sorting players by their total points scored in ascending order.

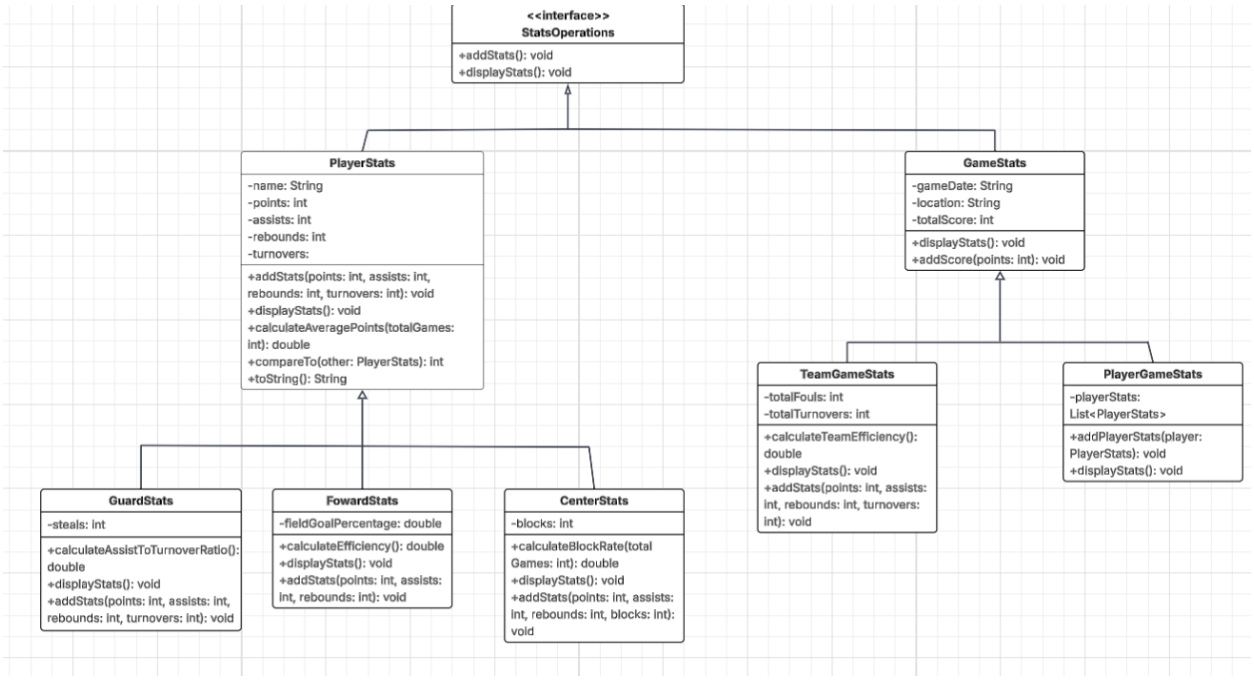
How It Works: The compareTo() method in PlayerStats is overridden to define the comparison logic based on the points attribute.

Comparator:

Implementation: A separate class, StatComparator, implements the Comparator interface to allow sorting players by different attributes, such as rebounds or assists.

How It Works: The compare() method is overridden to provide comparison logic for the specified attribute.

UML CLASS DIAGRAM:



Deliverable 2 (50% Implementation):

The following components will be implemented for Deliverable 2:

- **Classes**:
  - PlayerStats, GameStats
  - Subclasses: GuardStats, ForwardStats, CenterStats, PlayerGameStats.

- **Interface**:
  - StatsOperations

- **Methods**:
  - Implement runtime-polymorphism (displayStats()) in PlayerStats and its subclasses.
  - Implement file management methods (saveToFile() and loadFromFile()).

- **Unit Testing**:
  - Test methods like addStats() and displayStats() to ensure correctness and accuracy.