# Sports League Manager - Project Documentation

## Table of Contents

---

## Project Overview

### What It Does

The Sports League Manager is a Java-based command-line application that allows users to manage sports teams and players. The system provides functionality for:

- **Team Management**: Register new teams with names and cities
- **Player Management**: Register players with personal information and positions
- **Assignment Operations**: Assign players to teams and remove them when needed
- **Data Retrieval**: View team rosters, search players, and generate league statistics
- **Validation**: Ensure teams don't exceed maximum capacity (15 players) and prevent duplicate assignments

### Core Architecture

The application follows a clean, object-oriented design with the following classes:

1. **`LeagueManager`** : Central business logic coordinator
2. **`Team`** : Represents individual teams and manages player rosters
3. **`Player`** : Represents individual players with team assignment tracking
4. **`LeagueManagerCLI`** : Command-line interface for user interaction

---

## How It Works

### System Flow

1. **Initialization**: The CLI creates a `LeagueManager` instance to handle all operations
2. **User Interaction**: A menu-driven interface guides users through available operations
3. **Data Management**: All data is stored in memory using `ArrayList` collections

### Key Features

- **Team Registration**: Prevents duplicate team names
- **Player Assignment**: Validates team capacity and player availability
- **Search Functionality**: Case-insensitive name searching
- **Statistics**: Real-time league statistics and player counts

---

## Running the Application

### Method 1: IntelliJ IDEA (Recommended)

1. **Open the project** in IntelliJ IDEA
2. **Navigate** to `src/main/java/com/sportsleague/LeagueManagerCLI.java`
3. **Right-click** on the `LeagueManagerCLI.java` file in the editor
4. **Select** "Run 'LeagueManagerCLI.main()'" from the context menu
   - *Alternatively, click the green play button (▯) next to the `main` method*
   - *Or use the keyboard shortcut: Ctrl+Shift+F10 (Windows/Linux) or Cmd+Shift+R (Mac)*

### Method 2: Maven Command Line

```
# Navigate to project directory
cd sports-league-manager

# Compile and run
mvn compile exec:java
```

### Method 3: Command Line (After Building)

```
# Build the project
mvn clean package

# Run the JAR file
java -cp target/sports-league-manager-1.0-SNAPSHOT.jar
com.sportsleague.LeagueManagerCLI
```

---

## Clean Code Practices

### Example 1: Meaningful Method Names and Single Responsibility

```java
public boolean assignPlayerToTeam(int playerId, int teamId) {
    Player player = findPlayerById(playerId);
    Team team = findTeamById(teamId);

    if (player == null || team == null) {
        return false;
    }

    return team.addPlayer(player);
}
```

**Clean Code Principles Demonstrated:**

- **Descriptive Method Name**: `assignPlayerToTeam` clearly indicates the method's purpose
- **Single Responsibility**: The method does one thing - assigns a player to a team
- **Clear Boolean Return**: Returns `true` for success, `false` for failure

### Example 2: Consistent Naming Conventions and Encapsulation

```java
public class Player {
    private int playerId;
    private String firstName;
    private String surName;
    private String position;
    private int teamId;

    public boolean isAssignedToTeam() {
        return teamId != -1;
    }

    @Override
    public String toString() {
        return String.format("Player{id=%d, firstName='%s', surName='%s',
position='%s', teamId=%d}",
                playerId, firstName, surName, position, teamId);
    }
}
```

**Clean Code Principles Demonstrated:**

- **Encapsulation**: All fields are private with public accessor methods
- **Consistent Naming**: Uses camelCase consistently throughout
- **Obvious intention**: `isAssignedToTeam()` clearly indicates what the method checks
- **String Formatting**: Uses `String.format()` for clean, readable string construction

### Example 3: Input Validation and Error Handling

```java
private void registerTeam() {
    System.out.println("\n--- Register New Team ---");
    System.out.print("Enter team name: ");
    String teamName = scanner.nextLine().trim();

    if (teamName.isEmpty()) {
        System.out.println("Team name cannot be empty!");
        return;
    }

    System.out.print("Enter city: ");
    String city = scanner.nextLine().trim();

    if (city.isEmpty()) {
        System.out.println("City cannot be empty!");
        return;
    }

    Team team = leagueManager.registerTeam(teamName, city);
    if (team != null) {
        System.out.println("Team registered successfully!");
        System.out.println(team);
    } else {
        System.out.println("Failed to register team. Team name might already exist.");
    }
}
```

**Clean Code Principles Demonstrated:**

- **Input Validation**: Checks for empty strings before processing
- **User Feedback**: Provides clear success/failure messages
- **Defensive Programming**: Handles both successful and failed registration scenarios

---

# Test Cases

## Unit Test Strategy

The project uses JUnit 5 for comprehensive testing of all business logic components. The testing approach combines multiple related assertions within individual test methods to validate complete workflows.

#### LeagueManager Test Cases

1. **Team Registration Tests**
    - `testTeamRegistration()`: Comprehensive test that validates:
        - Successful team registration with proper ID assignment
        - Team name and city assignment
        - Total team count tracking
        - Prevention of duplicate team registration (returns null for duplicates)
2. **Player Management and Assignment Tests**
    - `testPlayerRegistrationAndAssignment()`: Validates the complete player lifecycle:
        - Player registration with proper ID assignment
        - Total player count tracking
        - Initial unassigned state verification
        - Successful player-to-team assignment
        - Assignment state updates (player and team)
        - Assigned player count tracking
        - Error handling for invalid team/player IDs
3. **Player Removal Tests**
    - `testPlayerRemovalFromTeam()`: Tests player removal workflow:
        - Player assignment verification
        - Team player count before removal
        - Successful removal operation
        - State cleanup after removal
4. **Search Functionality Tests**
    - `testSearchFunctionality()`: Comprehensive search testing:

        - Exact first name matching
        - Exact last name matching
        - Partial name matching
        - Case-insensitive search
        - Empty results for non-existent players

5. **Unassigned Player Tracking Tests**

    - `testUnassignedPlayersTracking()`: Validates unassigned player management:
        - Initial unassigned player list
        - List updates after player assignments
        - Complete assignment scenario (empty unassigned list)

6. **League Statistics Tests**

    - `testLeagueStatistics()`: Tests statistical tracking:
        - Initial empty league state
        - Team and player count updates
        - Assigned player count tracking
        - Statistics accuracy after various operations

**Team Class Test Cases**

1. **Team Creation Tests**

    - `testTeamCreation()`: Validates initial team state:
        - Team ID, name, and city assignment
        - Initial player count (zero)
        - Team capacity status

2. **Player Addition Tests**

    - `testAddPlayerToTeam()`: Tests successful player addition:
        - Player assignment state before/after
        - Team player count updates
        - Player team ID assignment
        - Roster inclusion verification

3. **Duplicate Assignment Prevention Tests**

    - `testCannotAddPlayerAlreadyAssigned()`: Ensures:
        - Already assigned players cannot be re-added
        - Player count remains unchanged on failed addition

**Player Class Test Cases**

1. **Player Creation Tests**

    - `testPlayerCreation()`: Validates initial player state:
        - Player ID, name, and position assignment
        - Initial unassigned state (teamId = -1)
        - Assignment status verification

2. **Team Assignment Tests**

    - `testPlayerTeamAssignment()`: Tests assignment state management:

    - Initial unassigned state
    - Team ID assignment
    - Assignment status update

### Test Implementation Example

```java
@Test
public void testPlayerRegistrationAndAssignment() {
    Team team = league.registerTeam("Raptors", "Toronto");
    Player player1 = league.registerPlayer("Kyle", "Lowry", "Point Guard");
    Player player2 = league.registerPlayer("DeMar", "DeRozan", "Shooting Guard");

    assertNotNull(player1);
    assertNotNull(player2);
    assertEquals(1, player1.getPlayerId());
    assertEquals(2, player2.getPlayerId());
    assertEquals(2, league.getTotalPlayers());
    assertEquals(0, league.getAssignedPlayers());
    assertFalse(player1.isAssignedToTeam());

    // Test successful player assignment
    assertTrue(league.assignPlayerToTeam(player1.getPlayerId(), team.getTeamId()));
    assertTrue(player1.isAssignedToTeam());
    assertEquals(team.getTeamId(), player1.getTeamId());
    assertEquals(1, league.getAssignedPlayers());
    assertEquals(1, team.getPlayerCount());

    // Test error handling
    assertFalse(league.assignPlayerToTeam(player2.getPlayerId(), 999));
    assertFalse(league.assignPlayerToTeam(999, team.getTeamId()));
}
```

### Test Coverage Summary

The test suite provides comprehensive coverage of:

- **Entity Creation**: Team and Player instantiation
- **Registration Logic**: League management of teams and players
- **Assignment Workflows**: Player-to-team assignment and removal
- **Search Capabilities**: Name-based player search with partial matching
- **State Management**: Tracking assigned/unassigned players
- **Statistics**: League-wide metrics and counts
- **Error Handling**: Invalid ID scenarios and duplicate prevention
- **Data Integrity**: Consistent state across all entities

---

## Dependencies and Build Configuration

### Project Dependencies

#### JUnit 5 Testing Framework

```xml
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
</dependency>
```

**Source**: Maven Central Repository

**Purpose**:

- `junit-jupiter-api` : Provides annotations ( `@Test` , `@BeforeEach` , etc.) and assertion methods
- `junit-jupiter-engine` : Runtime engine that executes JUnit 5 tests
- `scope>test</scope>` : Dependencies only available during test compilation and execution

---

## Maven Plugins Explanation

The `<build>` section in Maven defines how the project should be compiled, tested, and executed. Without proper plugin configuration, Maven might use outdated defaults or lack necessary functionality.

### Plugin Breakdown

**1. Maven Compiler Plugin**

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.11.0</version>
    <configuration>
        <source>23</source>
        <target>23</target>
    </configuration>
</plugin>
```

**Purpose**:

- Ensures compilation uses Java 23 features
- Without this, Maven might default to older Java versions
- Matches our `<properties>` section settings

**2. Maven Surefire Plugin**

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.1.2</version>
</plugin>
```

**Purpose**:

- Executes unit tests during `mvn test` phase
- Generates test reports and handles test failures properly
- **Critical for GitHub Actions**: Ensures tests run automatically in CI/CD pipeline

**3. Exec Maven Plugin**

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.1.0</version>
    <configuration>
        <mainClass>${exec.mainClass}</mainClass>
    </configuration>
</plugin>
```

**Purpose**:

- Enables running the main class with `mvn exec:java`
- Uses the `exec.mainClass` property we defined (`com.sportsleague.LeagueManagerCLI`)
- Provides a standardized way to execute the application
- **Essential for CLI applications**: Allows easy execution without manual classpath management

## Integration Benefits

**For Development**

```
mvn compile          # Compiles with Java 23
mvn test             # Runs JUnit 5 tests with proper reporting
mvn exec:java        # Launches the CLI application
```

**For GitHub Actions**

The plugins ensure that automated workflows can:

1. **Compile** the project with correct Java version
2. **Test** using JUnit 5 with proper test discovery
3. **Execute** the application for integration testing
4. **Report** test results and coverage metrics

**Example GitHub Actions Integration**

```
- name: Run tests
  run: mvn test

- name: Run application
  run: mvn exec:java
```