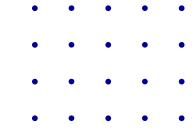


FootStat Analytics System

Object-Oriented programming Java



By:
Diyar Ossebay
Assem Ibragim
Kuanysh Mussabekuly



Content

- SOLID Principles
- Dependency Injection
- Database Connection (JDBC)
- Design Patterns
- Language Features (Streams & Lambdas)
- Database Schema
- Use Case Diagram

Single Responsibility Principle (SRP)

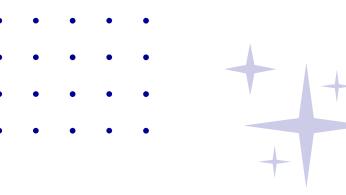


Every class should have a single responsibility.

We separated our architecture into strict layers. Entities hold data, Repositories handle SQL, and Services handle logic.

```
1 package entity;  
2  
3 public class Player extends BaseEntity { 40 usages & asemibragim32-ui +1  
4     private String name; 4 usages  
5     private String position; 4 usages  
6     private int teamId; 3 usages  
7     private String teamName; 4 usages  
8  
9     public Player(int id, String name, String position, String teamName)  
10        super(id);  
11        this.name = name;  
12        this.position = position;  
13        this.teamName = teamName;  
14    }  
15  
16    public Player(String name, String position, int teamId) { 1 usage & a:  
17        this.name = name;  
18        this.position = position;  
19        this.teamId = teamId;  
20    }
```

```
public class PlayerRepository implements IPlayerRepository { 1 usage & Solidiguana  
    private Connection con = DatabaseConnection.getInstance().getConnection(); 5 usages  
    @Override 10 usages & Solidiguana  
    public List<Player> findAll() {  
        List<Player> players = new ArrayList<>();  
        String sql = "SELECT p.id, p.name, p.position, t.name as team_name FROM players p J  
        try (Statement stmt = con.createStatement(); ResultSet rs = stmt.executeQuery(sql))  
            while (rs.next()) {  
                players.add(mapRow(rs));  
            }  
        } catch (SQLException e) { e.printStackTrace(); }  
        return players;  
    }
```



Open/Closed Principle (OCP)

Software entities should be open for extension, but closed for modification.

We used the **Strategy Pattern** for rating calculations. If we need to add a "Goalkeeper" rating logic later, we create a new class without modifying the existing **StatService**.

```
public class StrategyFactory { 2 usages & Solidiguana

    public static RatingStrategy getStrategy(String position) { 1 usage & Solidiguana
        if (position == null) return new DefenderStrategy();

        switch (position.trim().toUpperCase()) {
            case "FORWARD":
                return new AttackerStrategy();

            case "MIDFIELDER":
                return new MidfielderStrategy();

            case "DEFENDER":
            case "GOALKEEPER":
                return new DefenderStrategy();

            default:
                return new MidfielderStrategy();
        }
    }
}
```

```
public class DefenderStrategy implements RatingStrategy { 2 usages & Solidiguana

    @Override 1 usage & Solidiguana
    public double calculate(int goals, int assists, int minutes) {
        double score = 6.0;

        if (minutes >= 80) score += 1.0;

        score += (goals * 0.5) + (assists * 0.5);

        return Math.min(score, 10.0);
    }
}
```



Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

In our StatService, we rely on the RatingStrategy interface. Whether the factory returns AttackerStrategy or DefenderStrategy, the service works correctly because both strictly follow the contract.

```
public StatService(IStatRepository statRepo, IPlayerRepository playerRepo) { 1 usage & Solidiguana
    this.statRepo = statRepo;
    this.playerRepo = playerRepo;
}

public String processStats(int pid, int mid, int goals, int assists, int minutes) { 1 usage & Solidiguana
    if (minutes < 0 || minutes > 120) return "ERROR: impossible minutes";
    if (goals < 0 || assists < 0) return "ERROR: negative goals or assists";

    Player player = playerRepo.findById(pid);
    if (player == null) return "ERROR: Player not found";

    RatingStrategy strategy = StrategyFactory.getStrategy(player.getPosition());

    double rating = strategy.calculate(goals, assists, minutes);

    PlayerStat stat = new PlayerStat(pid, mid, goals, assists, minutes, rating);
    return statRepo.save(stat) ? String.format("Saved! Rating: %.2f", rating) : "DB error";
}
```

```
public interface RatingStrategy { 10 usages 3 implementations & Solidiguana
    double calculate(int goals, int assists, int minutes);
}
```

```
public class AttackerStrategy implements RatingStrategy { 1 usage & Solidiguana
    @Override 1 usage & Solidiguana
    public double calculate(int goals, int assists, int minutes) {
        double score = 6.0 + (goals * 1.0) + (assists * 0.8);

        if (minutes < 20) score -= 1.0;

        return Math.min(score, 10.0);
    }
}
```

Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.

Instead of one massive `IGlobalRepository`, we created specific interfaces.
`ITeamRepository` has `findByLeague`, while `IUserRepository` has `findByUsername`.

```
public interface IRepository<T> { 5 usages 10 implementations
    List<T> findAll(); 10 usages 5 implementations
    T findById(int id); 9 usages 5 implementations
    boolean save(T entity); 5 usages 5 implementations
    T mapRow(ResultSet rs) throws SQLException;
}
```

```
public interface IUserRepository extends IRepository<User> { 6 usages 1 implementation
    User findByUsername(String username); 2 usages 1 implementation
}
```

```
public class UserRepository implements IUserRepository { 1 usage
    private Connection con = DatabaseConnection.getInstance().getConnection(); 4 usages
}
```