Name

Aaron Aaeng

Title

fantasy.connor.fun - A Fantasy Drafting System Officially Supported by the connor.fun Family Brand of Brands

Description

A website that allows users to compete in fantasy competition leagues. Users can create an account then start playing against other players in leagues. This will use a daily fantasy model. Drafters will be given an overall budget and can choose any player they can afford to draft.

Implemented Features

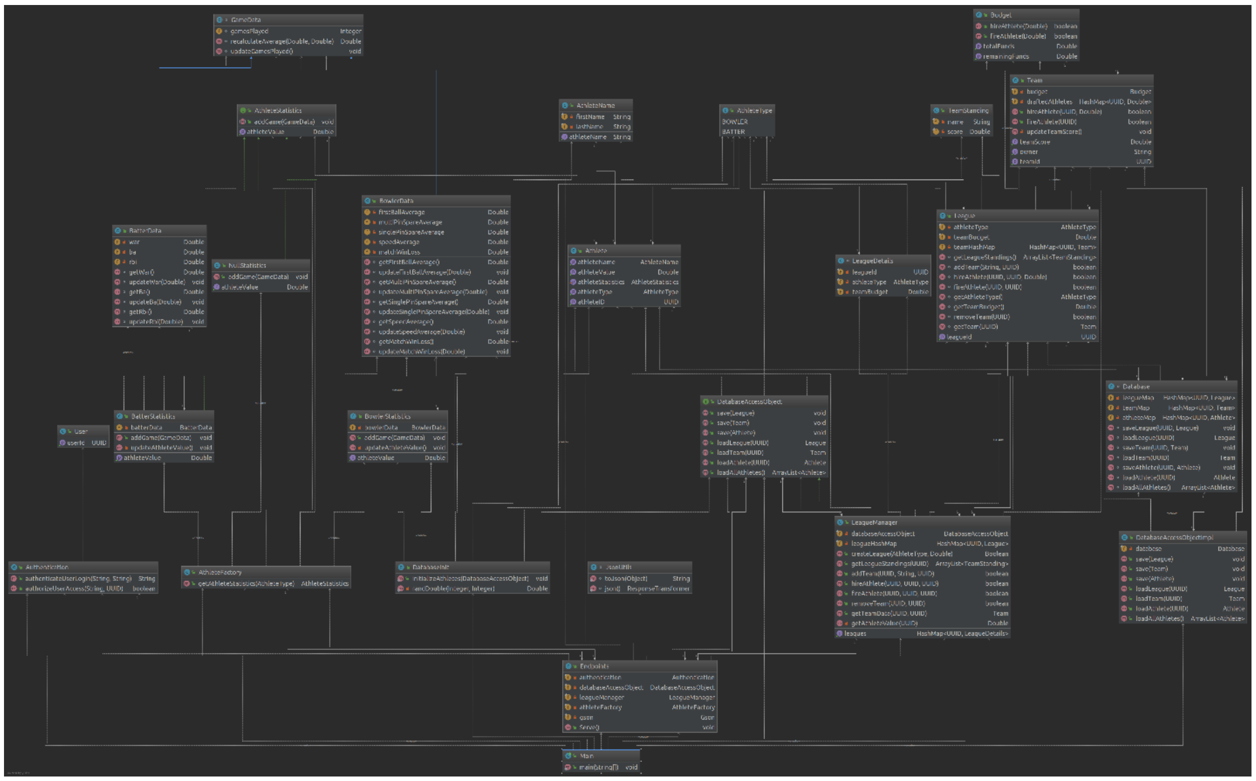
UR-ID	Description
UR-1a	The drafter needs to be able to view their current team lineup
UR-1b	Drafters need to be able to hire new athletes
UR-1c	Drafters need to be able to fire current athletes
UR-2a	A drafter can join a league
UR-2b	A drafter can leave a league
UR-3	Drafters can create a new team after joining a league
UR-4	Drafters need to see current and relevant statistics of all athletes
UR-7	The current standings of all players in all leagues should be accessible
UR-8	Drafters need to be able to see their remaining budget within a given league

Features Not Implemented

UR-ID	Description
UR-5	Athletes should be searchable
UR-6	League types should be searchable

These two features weren't implemented as they take place exclusively on the frontend. All necessary data is sent by the endpoint. I just did not write the JS to make it searchable. I am not a web developer.

Class Diagram



As this diagram is illegible at this size, a high-resolution version can be found in the top level of the repository. It is titled “Final Class Diagram.png”

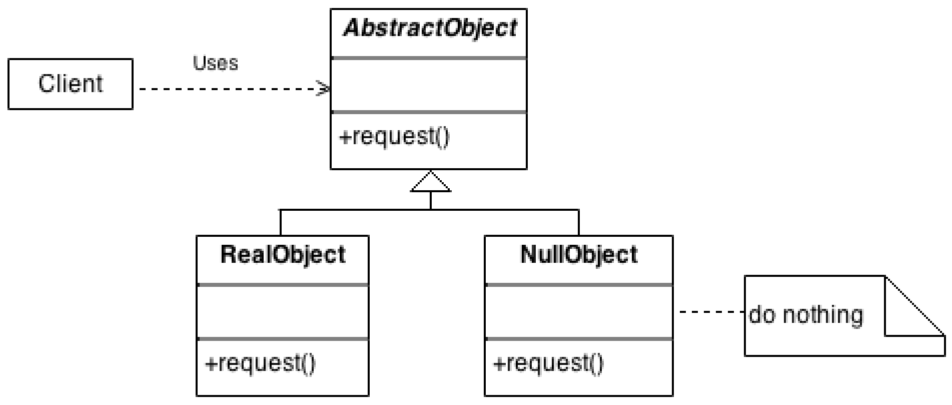
I made major changes between my original and final class diagrams. These all took place because of problems I encountered with the serialization and deserialization of my athletes. I needed to use polymorphism in order to avoid conditional statements for every different sport an athlete could play. In the original design, I was using generic types to dictate what type of athlete was instantiated. This worked great for static code, but I ran into a problem at runtime. After compilation, the templated type is lost. This meant that it was impossible to deserialize athletes that either came from the REST endpoints or were loaded from the database. To remedy this, I implemented a factory to dynamically select the right sports statistics strategy at runtime.

That being said, the rest of the design stayed pretty stagnant. The interfaces established between the different objects held up throughout the development process. I’m happy that I planned those early. Otherwise, I would have encountered serious problems as I updated athlete construction and the database being used. I also didn’t have to add many objects over the course of the project. By determining everything within the domain of my project beforehand, I already knew most of what I would have to implement.

Implemented Design Patterns

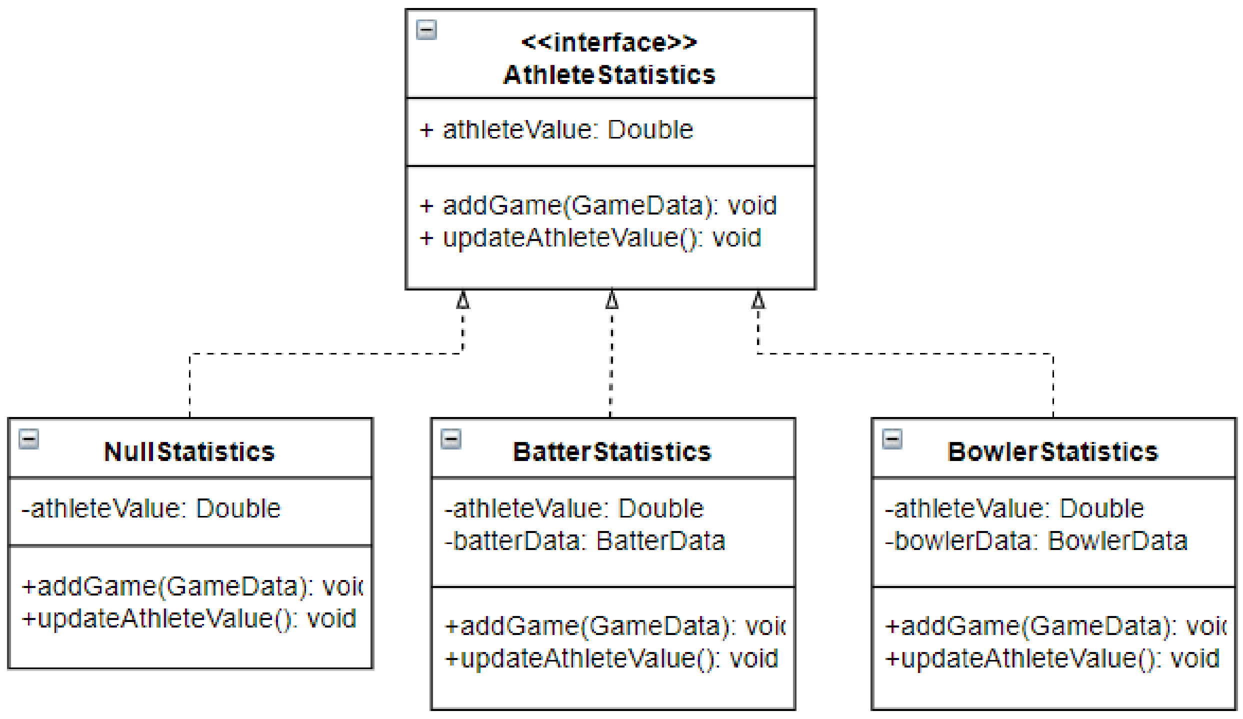
Null Object

Standard UML



https://sourcemaking.com/design_patterns/null_object

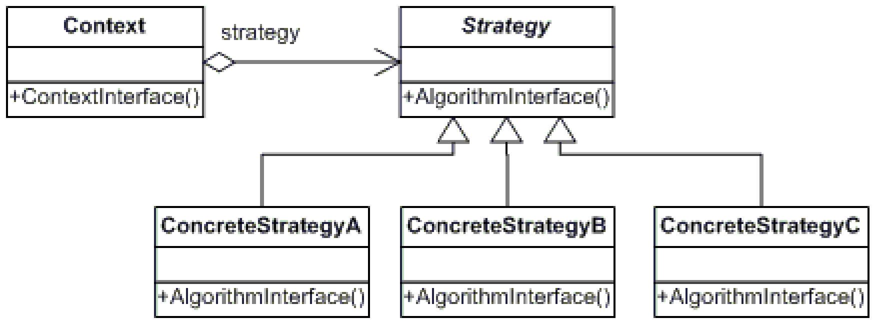
Implementation UML



I implemented this exactly as intended. The NullStatistics object is simply a filler that exists to prevent a runtime error. Because the strategy being used is dynamically selected by the factory, there exists the possibility that the input given to the factory is corrupted. Therefore, there needed to be a way to ensure this would not cause a fatal crash to take place should a corrupted POST request come through the endpoints. This allows the server to keep functioning and can let me check to see if there was an error in another location.

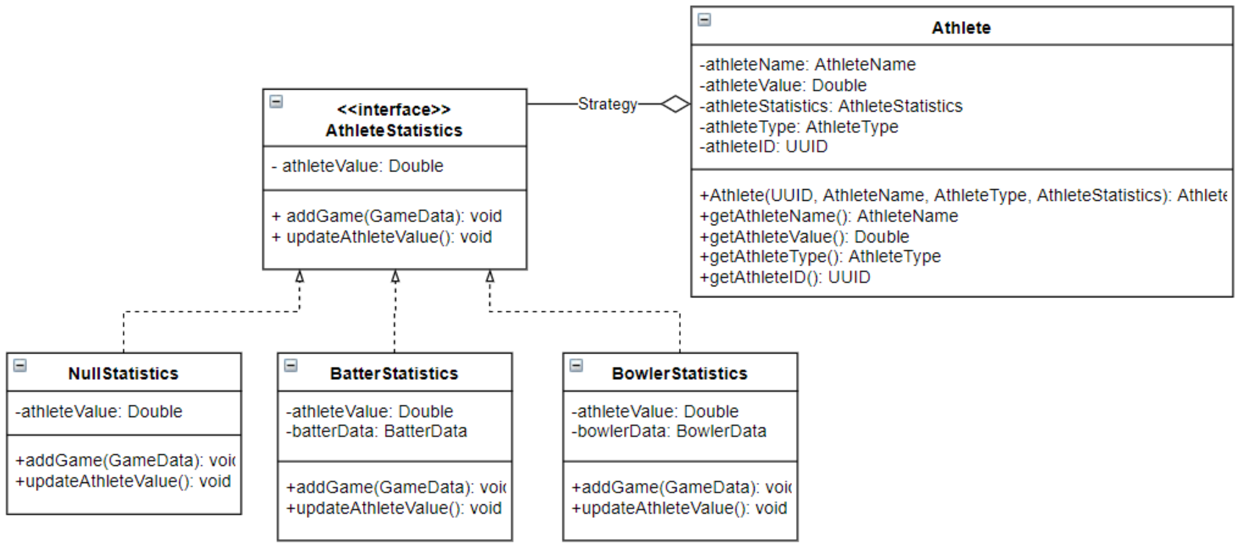
Strategy

Standard UML



https://www.researchgate.net/figure/UML-class-diagram-for-Strategy-pattern_fig27_249885094

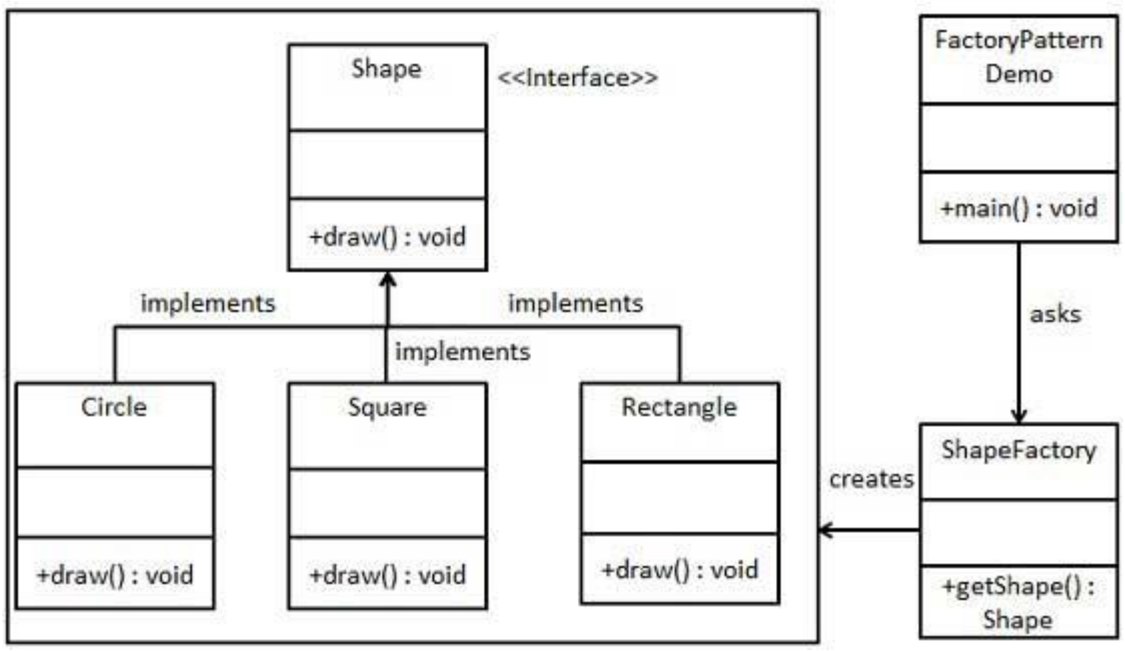
Implementation UML


<<interface>>
AthleteStatistics
- athleteValue: Double
+ addGame(GameData): void
+ updateAthleteValue(): void
Strategy
Athlete
-athleteName: AthleteName
-athleteValue: Double
-athleteStatistics: AthleteStatistics
-athleteType: AthleteType
-athleteID: UUID
+Athlete(UUID, AthleteName, AthleteType, AthleteStatistics): Athlete
+getAthleteName(): AthleteName
+getAthleteValue(): Double
+getAthleteType(): AthleteType
+getAthleteID(): UUID
NullStatistics
-athleteValue: Double
+addGame(GameData): voi
+updateAthleteValue(): void
BatterStatistics
-athleteValue: Double
-batterData: BatterData
+addGame(GameData): voi
+updateAthleteValue(): void
BowlerStatistics
-athleteValue: Double
-bowlerData: BowlerData
+addGame(GameData): voi
+updateAthleteValue(): void

The strategy pattern was used to solve the problem with athlete types. Instead of thinking of athletes from different sports as different entities entirely, I instead had to think of the different sports as different strategies that athletes have. I created a standardized athlete class that had a statistics module. This module had a standardized interface to abstract the different types of calculations and statistics endemic to each sport. It solved my deserialization problem and worked well with how I dynamically constructed athletes.

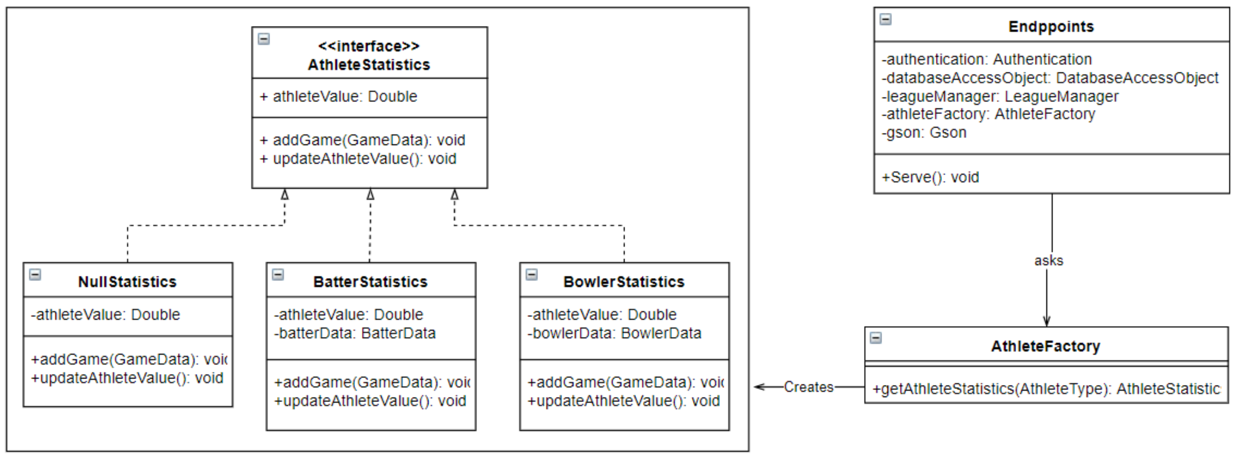
Factory

Standard UML



https://www.tutorialspoint.com/design_pattern/images/factory_pattern_uml_diagram.jpg

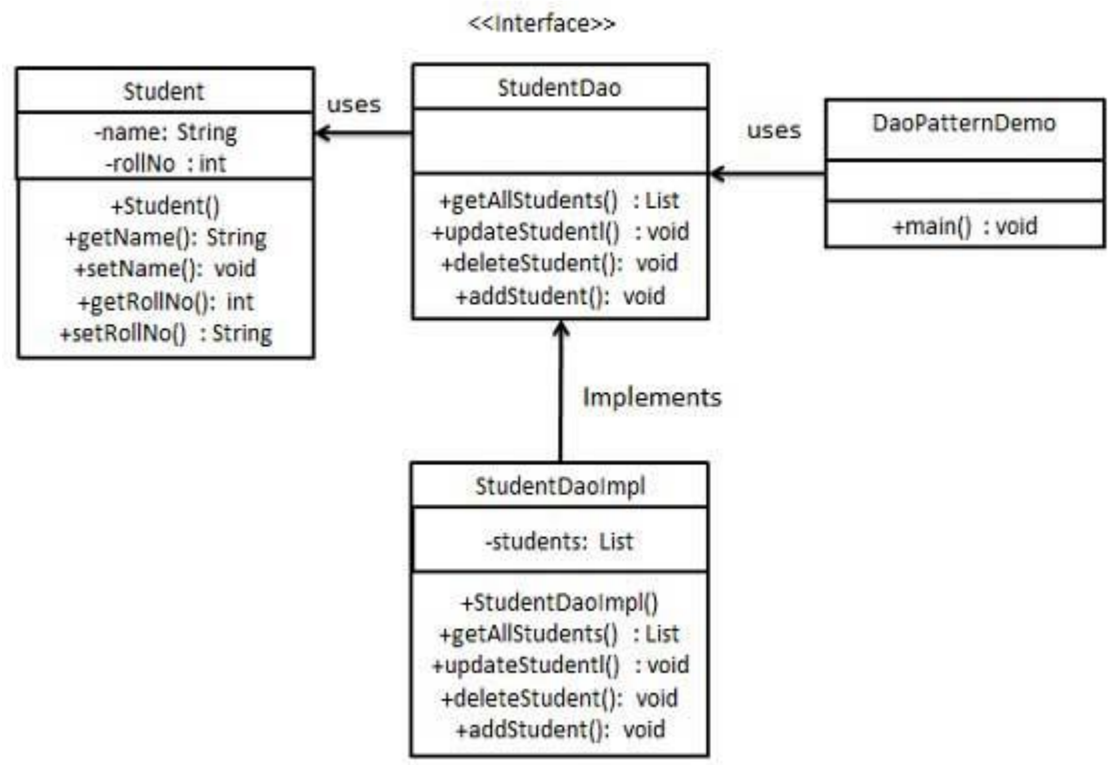
Implementation UML


<<interface>> AthleteStatistics
+ athleteValue: Double
+ addGame(GameData): void
+ updateAthleteValue(): void
NullStatistics
-athleteValue: Double
+addGame(GameData): voi
+updateAthleteValue(): void
BatterStatistics
-athleteValue: Double
-batterData: BatterData
+addGame(GameData): voi
+updateAthleteValue(): void
BowlerStatistics
-athleteValue: Double
-bowlerData: BowlerData
+addGame(GameData): voi
+updateAthleteValue(): void
Endppoints
-authentication: Authentication
-databaseAccessObject: DatabaseAccessObject
-leagueManager: LeagueManager
-athleteFactory: AthleteFactory
-gson: Gson
+Serve(): void
asks
AthleteFactory
+getAthleteStatistics(AthleteType): AthleteStatistic
Creates

The factory pattern was used to allow for the proper dynamic selection of statistics modules at runtime. It provides a method that takes in an athlete type and returns the proper statistics module. This can then be given to an athlete in order for them to have proper statistics available. Because different types can come in at runtime, potentially form two different sources, I needed a way to handle the input well. This worked very well with the strategy pattern as they both exist to solve problems in dynamic code. The factory allows for the dynamic selection while the strategy allows for the dynamic use.

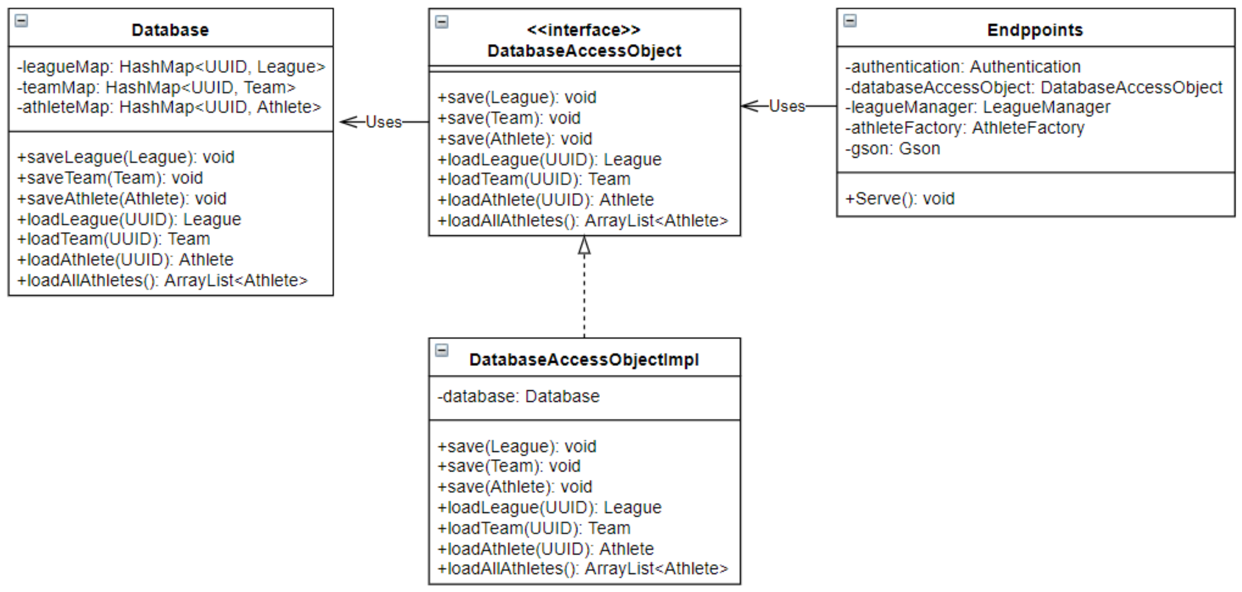
Data Access Object

Standard UML



https://www.tutorialspoint.com/design_pattern/images/dao_pattern_uml_diagram.jpg

Implementation UML



Database
-leagueMap: HashMap<UUID, League>
-teamMap: HashMap<UUID, Team>
-athleteMap: HashMap<UUID, Athlete>
+saveLeague(League): void
+saveTeam(Team): void
+saveAthlete(Athlete): void
+loadLeague(UUID): League
+loadTeam(UUID): Team
+loadAthlete(UUID): Athlete
+loadAllAthletes(): ArrayList<Athlete>
Uses
<<interface>>
DatabaseAccessObject
+save(League): void
+save(Team): void
+save(Athlete): void
+loadLeague(UUID): League
+loadTeam(UUID): Team
+loadAthlete(UUID): Athlete
+loadAllAthletes(): ArrayList<Athlete>
Uses
Endpoints
-authentication: Authentication
-databaseAccessObject: DatabaseAccessObject
-leagueManager: LeagueManager
-athleteFactory: AthleteFactory
-gson: Gson
+Serve(): void
DatabaseAccessObjectImpl
-database: Database
+save(League): void
+save(Team): void
+save(Athlete): void
+loadLeague(UUID): League
+loadTeam(UUID): Team
+loadAthlete(UUID): Athlete
+loadAllAthletes(): ArrayList<Athlete>

I implemented many DAOs across my project. This one just had the clearest UML. If you're curious, you can take a look at the full class diagram. The LeagueManager and League objects both serve as data access objects. That aside, I implemented this design pattern to facilitate cleaner database access. Throughout my time with the project, the database I was using changed many times. I jumped between MySQL, MongoDB, and just a filler database class in the final implementation. Even though my database changed multiple times, I only ever had to update the code in the DatabaseAccessObjectImpl object. The interface I defined was universal to the point that no changes needed to be made elsewhere. I selected this pattern for that very reason. It meant that I did not have to make sweeping changes across my source code when I made major architectural changes. It saved a lot of time making minor changes that later would've been overwritten. I decided to overload the save method in order to cleanly show what was happening in the code. The load methods could also be updated to be overloaded. I did not do that as my quick implementation of a database object did not make checking which table something was in very easy. A proper database would have solved this problem.

What I learned

I learned a few key things from this project. This was my first major project in Java. Not only was I writing a new language, I was working on my own instead of with a small team. I found myself second-guessing decisions far more often. This was usually for the better, but I realized that not having a team made the process of iterating over designs much slower. Designing an entire system solo is much harder than working with a group. However, planning it all out beforehand made it a lot easier. It meant that I didn't have to spend time figuring out how anything would fit together while I was implementing it. Finally, I realized that there are a lot of different facets that go into making a large system. All aspects of a system are equally important. The amount of time I spent on the entire project could easily be spent on the auth model to ensure maximum security and efficiency. This meant that I had to exercise some self-control to ensure that I didn't start working on an implementation just because I thought it would be cool. I didn't have enough time to work on something just because it would be cool.

A note on docs

The documentation for this project can be found in the docs/ directory at the top level of the repository. These can also be viewed at this link: https://aaronaaeng.github.io/fantasy.connor.fun-Backend/overview-summary.html