Final Design Documents

==========

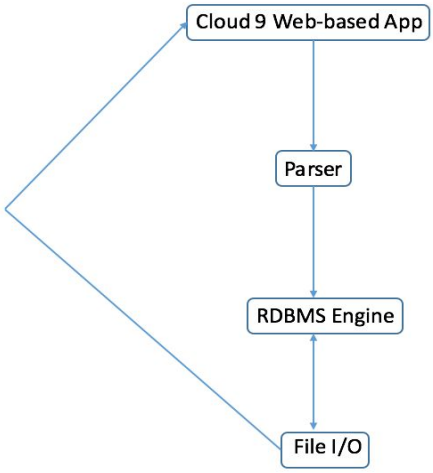
Purpose of the Project:

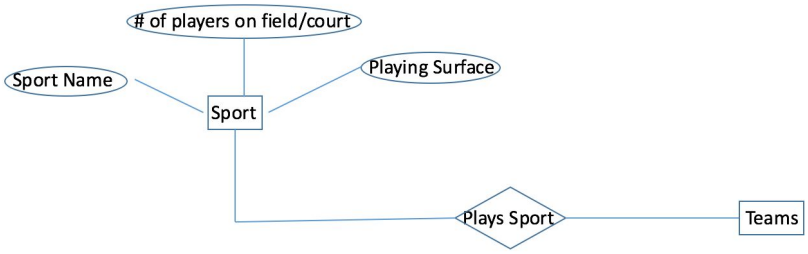
The purpose of this project is to create and implement a generic relational database management system. The database, being generic, will be able to handle adapt to different clients application. Our application was designed as a Sports Management system, allowing the user to store and manipulate information based on the sports, ~~the~~ players, ~~the~~ teams, and specific attributes for each of these entities. The player's given attributes include, ~~but are not limited to~~: name, age, team, ~~points scored~~, ~~and~~ jersey number, and position. The team's attributes include, ~~but are not limited to~~: team name, sport, wins, ~~and~~ losses, and ties. Lastly, the sport's attributes include, ~~but are not limited to:~~ sport name, ~~number of players on the field, and~~ playing surface, and country of origin.

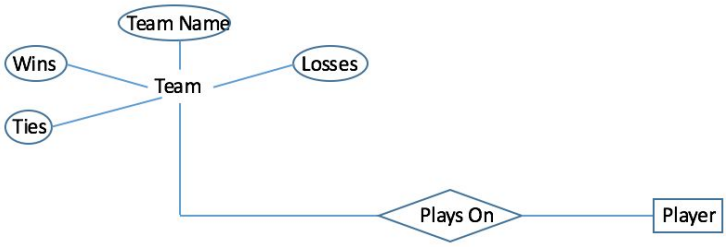
High Level Entities

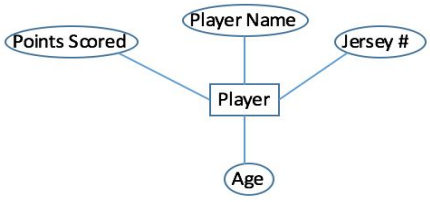
The picture below represents the Entity-Relationship diagram. This is simply a picture of how the information in our database works together.

Below is a visual representation of the High Level Entities.









We actually did not use a cloud 9 web-based app, but instead used a simple terminal-run application. The process starts with the interactive system when the user decides to input data. From there, the ~~parser will accept the pseudo-SQL commands generated by the interactive system~~

Interactive System Client generates an SQL command based on the given data. This command is then flushed through a socket to a server, which passes it to our parser.

Our parser takes this string and tokenizes it into an ArrayList of strings using predefined delimiters. From here, the parser hands the tokenized ArrayList to our Grammar class, which starts with the first token and iteratively moves forward. Whenever a token matches a codeword, it will call the matching method call from the engine based on the codeword. To allow for recursion, any query-based code word will send its second expression back into the parser to be evaluated again. This lets us work backwards on complicated, nested commands. The same method is applied to deal with nested comparisons. So once the parser has evaluated the token ArrayList, it calls the necessary engine function, giving it the proper data.

From here, the engine will determine if the data is valid. We have an attribute class and a row class to allow for easy checks of VARCHAR (x) and INTEGER domains. Additionally, we always make sure the data exists before we try to operate on it. In some cases, we make sure it doesn't exist before a function is executed; for example: before we CREATE a table, we want to make sure it doesn't already exist. Our CREATE table also checks if the table exists as a saved file to the database, so our case handling is thorough.

~~The parser will then parse those commands and call the methods/functions from the engine. The engine will then determine if the java commands are valid and execute accordingly. Lastly, the engine passes all of the data to the json file where we will keep a record.~~

Engine

Purpose:

The Engine serves as the database library, providing the the bulk of all relation operations. It hosts a hashmap of strings and our custom class Table. This is where we store all our relations. The Parser calls the engine functions when a certain method is needed~~. This input will come in as a string and await execution within the engine.~~ The engine is also where we check for validity, so if the input is not valid, the ~~parser~~ engine will throw an error. ~~Furthermore, the engine will not work directly alongside the interactive system. Meaning that, the engine will work in between the json file and the parser. The engine does all of the operations (insert, delete, select, etc.).~~ The engine does not work directly with the interactive system. It does, however, work in conjunction with the parser and serial file database.

High Level Design:

The engine consists of 4 modules: Engine.java, Table.java, Row.java, and Attribute.java.

In the Engine class file, we have a method for every possible SQL command. In the Table class file, we store data specific to individual tables, like their rows, attributes, and domains. In the Row class file, we store a primary key and values in an ArrayList. In the Attribute class file, we store an attribute name and its domain.

Low Level Design:

~~The low level design will consist of the three objects: Player, Team, and Sport. Each entity will have a corresponding class file to be used within the engine. Inside each class file, the given~~

~~attributes for each entity will be declared as members. This approach will help to simplify the code as well as increase readability. The attributes for each entity are shown below:~~

- ~~Sport:~~
  - ~~Name~~
  - ~~Number of players on field~~
  - ~~Playing surface (gym, field, etc.)~~
- ~~Teams:~~
  - ~~Team name~~
  - ~~Location~~
  - ~~Wins~~
  - ~~Losses~~
  - ~~Ties~~
- ~~Player:~~
  - ~~Name~~
  - ~~Age~~
  - ~~Jersey number~~
  - ~~Points scored~~

The Engine class itself is composed of the set of static functions listed below:

```
==========================================================================
// A function to create a new relation and add it to the database
// Parameters:
//   relation_name: The user-defined relation name
//   attributes: An array of Attributes, which have a name and domain
//   keys: An array of attribute names to determine how a key is formed
==========================================================================
public static Table createTable(String relation_name, ArrayList<Attribute> attributes,
ArrayList<String> keys)

==========================================================================
// A function to add a new row to an existing table
// Parameters:
//   relation_name: The name of the relation accepting the new row
//   values: A String Array of values that will make up the row
==========================================================================
public static void insertRow(String relation_name, ArrayList<String> values)

==========================================================================
// A function to update an existing row
// Parameters:
//   relation_name: The name of the relation the row belongs to
```

```
//          attributes: The attribute columns to update
//          values: The new attribute values. This ArrayList corresponds with the
//            attributes ArrayList
//          tokenized_conditions: The conditions a row must meet
================================================================================
public static void updateRow(String relation_name, ArrayList<String> attributes, ArrayList<String>
values, ArrayList<String> tokenized_conditions)

================================================================================
// A function to delete an existing row
// Parameters:
//   relation_name: The name of the relation the row belongs to
//          tokenized_conditions: The conditions a row must meet in order to be deleted
//==============================================================================
public static void deleteRow(String relation_name, ArrayList<String> tokenized_conditions)

================================================================================
// A function to return a new table from a selection
// Parameters:
//   relation_name: The name of the relation to perform the selection on
//          tokenized_conditions: The conditions a row must meet in order to be
//     selected
================================================================================
public static Table selection(String relation_name, ArrayList<String> tokenized_conditions)

================================================================================
// A function to return a new table from a projection
// Parameters:
//   relation_name: The name of the relation to perform the projection on
//          attributes: The list of attributes to keep from the original table
================================================================================
public static Table projection(String relation_name, ArrayList<String> attributes)

================================================================================
// A function to return a new table from with renamed attributes
// Parameters:
//   relation_name: The name of the relation with attributes to rename
//          attributes: The list of new, renamed attributes to apply to the relation
================================================================================
public static Table rename(String relation_name, ArrayList<String> attributes)

================================================================================
// A function to return a new table, created by taking the union of two other
```

```
//   tables
// Parameters:
//   new_relation_name: The parser-defined relation name to apply to the new table
//   relation_name1: The relation name of the first table
//   relation_name2: The relation name of the second table
================================================================================
public static Table setUnion(String new_relation_name, String relation_name1, String
relation_name2)

================================================================================
// A function to return a new table, created by taking the difference of two
//   other tables
// Parameters:
//   new_relation_name: The parser-defined relation name to apply to the new table
//   relation_name1: The relation name of the first table
//   relation_name2: The relation name of the second table
================================================================================
public static Table setDifference(String new_relation_name, String relation_name1, String
relation_name2)

================================================================================
// A function to return a new table, created by taking the cross product of two
//   other tables
// Parameters:
//   new_relation_name: The parser-defined relation name to apply to the new table
//   relation_name1: The relation name of the first table
//   relation_name2: The relation name of the second table
================================================================================
public static Table crossProduct(String new_relation_name, String relation_name1, String
relation_name2)

================================================================================
// A function to return a new table, created by taking the cross product of two
//   other tables
// Parameters:
//   new_relation_name: The parser-defined relation name to apply to the new table
//   relation_name1: The relation name of the first table
//   relation_name2: The relation name of the second table
================================================================================
public static Table naturalJoin(String new_relation_name, String relation_name1, String
relation_name2)

================================================================================
```

```
// A function show an existing relation
// Parameters:
//   relation_name: The name of the relation to be shown
================================================================================
public static String show(String relation_name)

================================================================================
// A function to open a table from a serialized file
// Parameters:
//   relation_name: The name of the relation to be opened
================================================================================
public static Table openTable(String relation_name)

================================================================================
// A function to write a table to a serialized file
// Parameters:
//   relation_name: The name of the relation to be written
================================================================================
public static void writeTable(String relation_name)

================================================================================
// A function to make the system forget about a table
// Parameters:
//   relation_name: The name of the relation to be closed
// ============================================================================
public static void closeTable(String relation_name)

================================================================================
// A function to remove an existing relation from the database and disk
// Parameters:
//   relation_name: The name of the relation to be dropped
================================================================================
public static void dropTable(String relation_name)

================================================================================
// A function to close the database, saving all tables to serialized files
================================================================================
public static void exit()

================================================================================
// A function to parse an ArrayList of tokenized conditions. This is the first
//   step of a 3 step process. This stage will break apart the tokenized
//   conditions into individual conditions that will be evaluated in the next
```

```
//   step.
// Parameters:
//   table: The table containing the rows to check against a condition
//   row_values: The attribute values for the row we are checking.
//   token_ArrayList: The tokenized conditions, containing all conditions
=============================================================================
public static Boolean parseConditions(Table table, ArrayList<String> row_values, ArrayList<String>
token_ArrayList)

=============================================================================
// This is the second step of a 3 step process. This stage will determine what
//   parts of the row to look at, either an attribute column or a specified
//    value.
// Parameters:
//   table: The table containing the rows to check against a condition
//   row_values: The attribute values for the row we are checking.
//   token_ArrayList: A single tokenized condition
=============================================================================
public static Boolean evaluateCondition(Table table, ArrayList<String> row_values,
ArrayList<String> condition_ArrayList)

=============================================================================
// This is the last step of the 3 step process. This stage will look at the
//   specified attribute value and the value from the row, and use the operator
//   to determine if it meets the condition or not.
// Parameters:
//   attribute1: The first, specified value we are checking against
//   operator: The operator that tells us how to compare the two values
//   attribute2: The value from the row
=============================================================================
public static Boolean checkCondition(String attribute1, String operator, String attribute2)

=============================================================================
// A simple function to return True or False based on whether or not a table
//   exists
// Parameters:
//   relation_name: The relation name of the table to be tested for existence
=============================================================================
public static Boolean tableExists(String relation_name)
```

The data structure that supports our relational database is as follows: HashMap<String, Table>. There are essentially three parts to this structure. The Table contains an ArrayList of Attributes, which holds the data like attribute domains and varchar lengths.

Additionally, Table also holds an ArrayList of Rows, and Rows hold ArrayLists of Strings containing attribute values. The decision to use ArrayLists was based on simplicity of the dynamic data structure, while having essential functions like: add, contains, remove, etc.

~~The main data structures in this database are hashmaps. We decided to go with hashmaps because the <key, value> setup allows for easy identification of what values are actually stored within the structure. Furthermore, hashmaps allow for quick insert/delete/lookup function calls at just O(1). This is crucial for a database considering that managing data revolves around the ability to insert, delete, and lookup data.~~

The process of giving each entity its own key will be related to combining two static attributes to create a unique ID for each object. The key is saved in the Row class, for easy lookup and changeability. Lastly, the data will be written and saved in a .ser file to prevent loss of data in between sessions. The format for reading and writing this data comes from Java's class-implements-serializable support. We chose this over .json because it is compact and easy to use with ArrayLists.

~~With all that being said, the three entity requirement has been met. Furthermore, each entity will have its own table. Each of our two relations will also be represented in table, also. This means that we will have met the five table requirement, as well.~~

Benefits/Assumptions/Risks/Issues:

The benefits of this approach rests in the beautiful consistency and concrete design. ~~Although the design relies heavily on the engine, creating player, team, and sport objects will help to organize the code and make it readable~~ The creation of the Table, Row, and Attribute classes enables highly-organizable, beautiful, and clean code. Furthermore, as said before, the time complexity of the hashmap functions allow for quick operation. The main assumption that is being made is that the time complexity of the given hashmap functions operate under O(1). Under the worst circumstances, it could take up to O(n). ~~However, the average case states otherwise.~~

Another benefit of this implementation is the quickness of the find and add functions of the vector class. Both functions have the time complexity of O(1). Of course, these are just assumptions based on the best/average cases, however, only in the worst case will the time complexity be O(n). The same is true of an ArrayList. ~~One of the risks of this implementation include the heavy reliance on the engine. As mentioned earlier, the engine is doing all of the dirty work while the parser and interactive system simply work in accordance with it.~~

One of the risks being taken is the use of only two files. By doing this, a lot of code is being placed into each, Table.java and Engine.java. This is a risk in the sense of comprehension to anyone, other than the developers, viewing the source code. If the code was broken down even further, this might have helped simplify the design.

Parser

Purpose:

The parser is used to interpret input (pseudo-SQL commands) from Standard input, or the Interactive System. It tokenizes a single string into an ArrayList of token strings, and then iterates through each token and calls correlating functions from the engine. ~~In essence, the parser's primary objective is to translate from SQL to Java. The output is then brought back to the engine to await execution.~~

High Level Modules

~~The parser will be written in Python, because Python has excellent i/o support. The parser will consist of three primary modules: one to read in input line by line, another to parse the said input, and a third to decide which engine function to use with the parsed input and then pass the data to that function.~~

We wrote the parser in Java because Java has a beautiful built-in StringTokenizer. Additionally, a Java parser was able to communicate with our Java engine much easier than a Python parser would have been able to.

Our Parser consists of 4 high-level modules: Parser, Grammar, Commands, and Queries.

The Parser module is the smallest of them all, but it is the primary one. It receives the input from Standard Input (or from the Server) and does a quick, shallow check to filter out invalid commands. It sends the commands one-by-one to the Grammar module.

The Grammar module is where the String is tokenized and looked at. It's also the home of our recursive evaluation of expressions, and a few other commonly used operations. Once the string is tokenized into an ArrayList, each token is looked at individually, and when a token matches a code word, the correlating function is then called to parse the remaining tokens. The correlating function will either be a command or a query.

The Command module houses all of the commands (non-queries) listed in the online Grammar. From CREATE TABLE to SHOW, this is where these kinds of SQL commands are parsed. Once parsed, the module will call the corresponding Engine function with the retrieved data, and the Engine will determine if it is valid.

The Query module is essentially the same as the Command module, but this one houses all of the queries and algebraic manipulations listed in the online Grammar. From SELECT to CROSS PRODUCT, this is where these kinds of SQL commands are parsed. Once parsed, the module will call the corresponding Engine function with the retrieved data, and the Engine will determine if it is valid.

Low Level Design:

~~The parser will be implemented through the use of a recursive descent parser. This is because the SQL can be nested. For example, SELECT * FROM (SELECT * FROM <table>) is a valid SQL call. That is why the parser must be recursive.~~

Firstly, the parser will take in a statement from Standard Input or the Server. These statements are based on the given pseudo-SQL DML language.

Secondly, the parser will evaluate the statement. ~~We plan to accomplish this using regex to determine if the input is creating a table, selecting a query, and so on. We will only parse a certain part of the input at a time, enabling the parser to know if it is dealing with input that will contain conditions, so it can look for those, or if it is simply creating a table, so it can look for a table name~~

~~and attribute list. Additionally, there will be a case for recursion. If needed, the parser can call its regex-based parsing function again, and return tables/relations/selections/etc from that. This will be done using a switch case block, and nested if statements. This will allow us to know exactly what engine function to use.~~

We used Java's StringTokenizer to tokenize the input and examine that, not regex. The Tokenizer was much easier, as it essentially does what we were trying to do with regex automatically.

~~Lastly, once we have the parsed input, we send in the data to the engine. This will be accomplished by sending an array of strings to the engine, always setting the first element to the function name in the engine. All other elements will be the necessary parsed data.~~

Once the input has been parsed, we call the correlating engine function and pass in the parsed data.

The Parser consists of Parser.java, Grammar.java, Commands.java, and Queries.java. The Grammar class itself is composed of the set of static functions listed below:

```
==========================================================================
// The general grammar function. Essentially, this takes a line and tokenizes
//   it, removing some unnecessary ones and rebuilding others.
// Parameters:
//   line: A String received from Standard Input, ready to be tokenized
==========================================================================
public static String parseLine(String line)

==========================================================================
// A function to figure out which method to call from the tokenized input
// Parameters:
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
==========================================================================
private static String callMethod(ArrayList<String> sql_tokens)

==========================================================================
// A function to move the token_index *PAST* a specified token, effectively
//   skipping it
// Parameters:
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
//   token: The specified token to skip
==========================================================================
public static Integer skipTokens(ArrayList<String> sql_tokens, String token)

==========================================================================
// A function to move the token_index *TO* a specified token, effectively
//   skipping *TO* it
// Parameters:
```

```
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
//   token: The specified token to skip to
=============================================================================
public static Integer skipToToken(ArrayList<String> sql_tokens, Integer token_index, String token)

=============================================================================
// A function to retrieve all tokens from a certain token up to a specified
//   token
// Parameters:
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
//   token_index: The starting token's index
//   token: The ending token
//   value: A value to determine whether or not to include the ending token in
//     the returned ArrayList
=============================================================================
public static ArrayList<String> retrieveTokens(ArrayList<String> sql_tokens, Integer token_index,
String token, Boolean value)

=============================================================================
// A function to evaluate an expression recursively. Essentially, whichever
//   query function it comes across first, it evaluates, and then continues with
//   the evaluated expression until it reaches an end or an already-existing
//   relation name.
// Parameters:
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
=============================================================================
public static Table evaluateExpression(ArrayList<String> sql_tokens)

=============================================================================
// A function to determine if a given ArrayList of tokens only contains an
//   already-existing relation name. This allows the above function to check
//   if an expression is algebraic or not.
// Parameters:
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
=============================================================================
public static Boolean isRelationName(ArrayList<String> sql_tokens)

=============================================================================
// A function that allows certain commands (DROP, SHOW, WRITE, etc.) to quickly
//   retrieve the relation name or determine if an expression is present
// Parameters:
//   sql_tokens: An ArrayList containining tokenized psuedo-SQL
=============================================================================
```

public static String getRelationName(ArrayList<String> sql_tokens)

Benefits/Assumptions/Risks/Issues:

Using a recursive descent parser helps to simplify the code. If done right, this will eliminate all of the tedious, time-consuming implementation of parsing each command individually. However, recursive parsers tend to be hard to implement. One of the biggest risks/issues is the lapse in understanding that can sometimes come with recursion.

Having our parser handle all recursion did help immensely. Our biggest challenge here was successfully implementing it, as we first tried to hardcode it in each function.

Interactive System

Purpose:

The purpose of the interactive system is to allow the user to manage and manipulate data without having to deal with SQL through our Application, Sports Management. The interactive system allows the user to create tables for players, sports, games, teams, etc. The user is also able to create new tables based on relations, such as players on a certain team, or teams that play a certain sport, etc. We also gave the user the ability to trade players, merge teams, or see which players on only on one team (if they're in the middle of being drafted). The user can also view all players, teams, sports, or a certain player, certain team, etc.

~~We have two plans for the interactive system, Plan A and a backup Plan B. Plan A is to go for the extra credit and host the interactive system online, using HTML and JavaScript to render the UI. Plan B is to simply use the terminal and have the user type out data and operations. Despite these differences, the overall idea is relatively the same - a Sports Management application.~~

~~The interactive system will serve as the bridge between the common man and our program. It will present itself as a Sports Management application, and we hope to be able to publish it to a free hosting service that can be accessed on any browser. If that proves too difficult, we will revert to simply using a terminal-based application. If we are successful with the website, however, all the user will have to do is input the URL. From then, the user see several tables, with perhaps arrows between them to represent relations, and the user will have the option to either add new data or manipulate current data. As far as the Terminal application goes, it would simply consist of the user typing in data in a directed format (The user will not have to be familiar with any SQL grammar).~~

High Level Modules

The Interactive System consists of 2 primary modules: The Client and the Server. The Client speaks to the user and generates SQL commands based on input, while the Server receives the SQL command via a Socket and sends it to the Parser.

~~Low Level Design~~:

The interactive system will start by giving the user a list of basic commands. From then, it will take in input from the user, and continue to ask the user for data, until it has enough data to form an SQL command. From here, it will send the SQL command to the parser.

Something else the Interactive System will handle is the display of tables and relations. When the user asks for the system to show table, the table will be returned to the interactive system to be displayed in *that* environment. This is the only case when the parser call returns a value (the value is given as a String), so whenever it does return a value, we know the user is trying to view a relation. This allows us to take in the String and send it through the socket to the Client, which will be listening. ~~This will be accomplished by having a format() function that takes in a table to be shown and formats it to fit aesthetically on the Interactive Systems window. This will be done by having the table come in as an array of arrays, which we will then iterate through and space each element appropriately.~~

~~If time permits, it will be hosted on https://c9.io/, a free website hosting service. We will use HTML to create a simple canvas, and then JavaScript to render the UI that the user will interact with. The UI will primarily consist of several tables, representing the different entities and their data. There will also be a menu with a hierarchy of buttons and drop-down menus with predefined inputs. From these, the user will easily be able to input and manipulate data.~~

~~The week objectives for this portion of the assignment are as follows:~~

- ~~Create the HTML index file rendering a canvas~~
- ~~Use JavaScript to fill the canvas with tables, buttons, and menus that, when interacted with, will send SQL- based statements to be parser via API.~~

```
==========================================================================
// The Main Method call for the Client class. This just starts the Client, which
//   attempts to connect to the Server
// Parameters:
//   args: Needed Parameter for main method call, not used
==========================================================================
public static void main(String args[])

==========================================================================
A function to set up the Client and continually check for input from the
//   Server. This is the core of the Client-Server connection
==========================================================================
private static void run()

==========================================================================
A function to read in a message from the Server via the InputStream and print
//   it out neatly
==========================================================================
private static void readMessage()
```

```
================================================================================
A function to send a message to the Server via the OutputStream Socket
// Parameters:
//   message: The String to send to the Client
================================================================================
private static void sendMessage(String message)

================================================================================
A function to disconnect from the Server cleanly
================================================================================
private static void disconnect()

================================================================================
A function to print out a welcoming statement on Client startup
================================================================================
private static void welcomeUser()

================================================================================
A function to create/open all tables from the previous session
================================================================================
private static void resume()

================================================================================
A function to print out the help menu, which lists all available commands
================================================================================
private static void showHelp()

================================================================================
A function that will prompt the user for a command, and then enter that
//   command's function when necessary. If an invalid input is given, it will
//   continue to prompt until 'Quit' is given
================================================================================
private static void promptUser()

================================================================================
A function to ask the user for a specific value. It will perpetually ask
//   the user for the value until it is given a valid input, or until the user
//   quits the program.
// Parameters:
//   prompt: The String to prompt the user with
================================================================================
private static String getUserInput(String prompt)
```

```
=============================================================================
A method to generate an SQL command for adding a sport
=============================================================================
private static void addSport()

=============================================================================
A method to generate an SQL command for adding a team
=============================================================================
private static void addTeam()

=============================================================================
A method to generate an SQL command for adding a player
=============================================================================
private static void addPlayer()

=============================================================================
A method to generate an SQL command for removing for adding a player
=============================================================================
private static void removePlayer()

=============================================================================
A method to generate an SQL command for updating for adding a team
=============================================================================
private static void updateTeam()

=============================================================================
A method to generate an SQL command for updating for adding a player
=============================================================================
private static void updatePlayer()

=============================================================================
// A method to generate an SQL command for merging for adding two teams
// This is the set union method
=============================================================================
private static void mergeTeams()

=============================================================================
A method to generate an SQL command for splitting two teams
// This is the set difference method
=============================================================================
private static void splitTeams()
```

```
================================================================================
A method to generate an SQL command for joining two teams
// This is the natural join method
================================================================================
private static void joinTeams()

================================================================================
A method to generate an SQL command for trading for adding a player
================================================================================
private static void tradePlayer()

================================================================================
A method to generate an SQL command for viewing a team
================================================================================
private static void viewTeam()

================================================================================
A method to generate an SQL command for viewing a team's roster (list of
//   player names)
// This is the projection method
================================================================================
private static void viewTeamRoster()

================================================================================
A method to generate an SQL command for viewing a player
// This is the selection method
================================================================================
private static void viewPlayer()
```

Interaction Specification

The Interactive System is a terminal-based application. The user, after activating the system, will be greeted with a friendly list of possible commands (These will not be the SQL commands, they will be worded to fit our Application theme: Sports Management). An example command would be something like

$ add player

From then, the system will ask the user for a player name, which team the player plays on, how old the player is, what the player's jersey number is, and which position the player plays. After it has all its required data, it will build the SQL command and add the player to the database by inserting it into a relation, like a team roster table.

Benefits/Assumptions/Risks/Issues:

The main benefit of using a terminal program is its simplicity. It will allow the user to input raw text as opposed to having, say, some UI-based system that requires a method for each

button/menu. Regarding assumptions and risks, we will have to assume that the user knows how to use a CLI. We are not assuming he knows SQL. The only risk we took here is the display of tables, which required us to implement a return system within the parser to return tables if the SHOW command is invoked.

~~One of the hardest challenges to tackle with this will be the API. Since we will be using JavaScript to send the SQL-based grammar to our Java parser, we will need to make sure it is done as efficiently as possible. If each SQL operation is represented as a button on the canvas, we will be able to then grab data from text boxes and send that to the parser. However, finding a way to aesthetically do this might be difficult.~~

~~I think in the long run, an easy-to-access website will be extremely beneficial to the end user, and since JavaScript isn't that difficult to handle, we should be fine. We just need to manage our time wisely and make sure the parser and engine function as designed here.~~