Group Project Requirements Specification

Note: I make mistakes

I've tried my best to think of everything to keep you focused on learning outcomes and programming instead of frustrations due to errors in requirements. I make mistakes, and so I reserve the right to alter this document and inform you of changes when I need to correct those mistakes. I will always endeavour to do so in a fair way.

Welcome to Canada eh?

Some of you may not be able to study in Canada this semester. So, this semester we will bring a little bit of Canada to you! **Hockey!** James Creighton invented the sport of hockey in Windsor, Nova Scotia just a 45 minute drive from Halifax!

First read this: https://simple.wikipedia.org/wiki/Ice_hockey

If you want a short introduction to Hockey because you aren't much of a sports fan, here are 9 minutes of highlights from one of the greatest games of all time. This is game 3, Canada versus the USSR in the 1987 Olympics. The only series where two of the greatest players of all time, Mario Lemieux and Wayne Gretzky, played on the same team. Here's the best part: the soviets were so good that even with the dream team of Lemieux and Gretzky it still was a very close game! Watch all the way to the end!
https://www.youtube.com/watch?v=M_Ip46b4nTY

If you'd like a longer introduction, here is a classic game: Montreal Canadiens vs. Boston Bruins, game 7 of the 1979 Stanley Cup final. These two teams are part of the "Original 6", teams that have been around since the early days of hockey. When they play each other it's always a real battle and you see some good hockey. This is a famous game because of the mistake made by Boston's coach Don Cherry, who later goes on to become an infamous Canadian icon:
https://www.youtube.com/watch?v=15p0nZYNJFE

Those old timers were crazy playing without all the pads and helmets the players wear these days.

If you are interested in the details of the National Hockey League:
https://en.wikipedia.org/wiki/National_Hockey_League

Application Description

At Electronic Arts when I worked on the NHL franchise one of the most fun things I ever worked on was called **Dynasty Mode**. I was tasked with creating the logic that drove the user interface and "AI" behind this mode. It was very interesting and a lot of work. We will reproduce some of this mode in our project this semester.

We will pretend that you are the simulation team on a large video game team, but we're going to take a lot of liberties and let our imaginations go a little wild, for example:

- Major PC and console video games are not made with Java
- We'll use a remote DB, which would be unlikely in a game (but this is slowly changing!)
- Video games often sacrifice code quality for code performance, we will never make this choice

Imagine that the team as a whole is working on a full hockey game, one that a player can play while controlling the players in 3D and everything. Your group of programmers is the simulation team, responsible for creating the logic that will simulate hockey seasons for a fictional hockey league (DHL, the Dalhousie Hockey League). The company you worked for has issued a new directive for development of all of their video games: *whenever possible, games must be created in a modular fashion so that components can be reused in multiple modes of delivery*. For example, if you create something like Dynasty Mode you must make it work such that it could be plugged into a console game written for the Xbox or a mobile game running on iOS.

At the end of this semester we will have a full command-line based simulator capable of reproducing the successes and failures of a hockey team over decades of hockey seasons. It will be capable of interesting things like simulating trading players, player injuries & retirements, and game & league results. We will build this project using Agile methodology, meaning it will evolve over three iterations. We do not need to define every single thing the app will do now; we only need to pick an initial direction with a high-level overview of what we want and then "GO!". It will be very interesting to see how close we can get to the real thing! Coincidentally this is how game development works too, except in video games they barely even start with a word document like this.

CSCI5308 Specific Technical Requirements

This course requires certain key technical functionality in any project we choose as our group project. These key aspects of the program were chosen by Rob to force your group to debate, design, implement and test a solution that will require you to exercise each of the best practices learned in the course.

- Multiple users (**in this case, the possibility of multiple player created teams that can be in different instances of the imported league, at different levels of progress**)
- A remote database (we will use a MySQL variant hosted at Dalhousie), one for each of 3 different environments the app is deployed to
- Error handling and logging
- Configurable business logic (you will learn more about this later)

During the course the requirements your group is given for each milestone will ensure we explore each of these areas, beginning with the remote database and this project's version of "multiple users".

Additionally, in this course all code must be implemented using Test-Driven Development. At minimum, all business logic in this course must be 100% covered by unit tests.

Milestone 1 Requirements

What may seem like brevity in this document is an intentional omission of detail to provide students with experience in the uncertain nature of developing software for a customer using Agile methodology. Right now your client, imaginary lead software architects who are directing your development here, have provided you with a high level and very loose request. In previous semesters where we did web development the clients would often be non-technical, forcing students to experience the difficulty in interpreting requirements from non-technical customers. This semester we will experiment with working in a cross functional development team (where different aspects of the software are developed by different departments or sub teams). The lead architects are in charge of assembling all of the components into a final video game: rendering systems, sound and input, game logic and AI, front end and in-game overlays and finally your simulation team. They give you general direction for how your component needs to work but give you a great deal of freedom in the implementation details. This is a both a blessing and a curse. Freedom allows you to be creative and come up with designs you like, however it also means that there is an overall lack of high-level design. **This means that change is inevitable, and you should take the inevitability of this change into account.**

This milestone they have directed you to complete the following epics & tasks:

- **DevOps**:
  - Project setup & Configuration:
    - Set up your project and build system, provide your team with a simple "Hello world!" command line app set up in whatever IDE build system your team chooses. I recommend Maven or Gradle. The Canada Revenue Agency that hires a lot of our interns loves Maven experience.
    - Use your build system's mechanism to provide configuration to your code. Your code must be capable of changing things like the database server URL, db port, db username, db password and other configurable items that come along **without changing code.** This usually means things like .properties files or .ini files. You are free to use whatever mechanism imports or loads configuration you prefer.
    - Help your teammates configure their local environments so that they understand how to add configuration items, change configuration values, compile the app and run the app. Perhaps you want to make a document

you can add to your Teams channel that explains how to build the project, how to set up the IDE and your environment, and how to change config?

  - Establish standards:
    - Research best-practices for Java, then present some initial choices for your team to debate: what should your naming standard be? What about indentation? Tabs or spaces? Find a standard and then get your team to agree on the standards. Add these standards to your setup document.
  - Continuous Integration & Continuous Delivery (CI/CD):
    - Set up a continuous integration system using your gitlab project's CI/CD system that:
      - Detects a push to any branch in your project, pulls the code and builds the code, reporting success or failure.
      - Runs all unit tests from test driven development, and reports their success or failure (**Note: we will ask you to prove that individual test success/failure detection works as part of the submission for milestone 1**).
    - Set up continuous delivery:
      - Your final built product will be deployed to 2 different environments represented by 2 of your group's Timberlea accounts. The system will copy the final executable to the student's account ready for you to execute if you are subsequently SSH'd into that account.
      - Your CD system will configure the deployed executable to properly execute on the given environment:
        - Your develop branch's deployed executable must point to your **TEST** database.
        - Your release branch's deployed executable must point to your **PRODUCTION** database.
        - Local builds your developers do on their own machines without the CD system will point to the **DEVELOP** database of course.
- **Database Setup:** ***(Omit this epic if you are a team of 3, see note)***
  - Define a database schema that will support the "business" model (the classes that make up your team's opinion of how to represent all of the objects that will be needed to represent a hockey league and all of the various information in this project, or at least the information you have now…)
    - Think of your database as the game's "Save file", it will save the data we import for the simulation (the data about the league), player's choices for their team (and the ability to support multiple teams for the player), as well as the state of your league.
  - Create the tables, fields, indexes and relationships (keys) between tables on all three of your environments: develop, test and production. Develop is the database programmers will connect to while they are writing their code on their

    local machines. Test is the database that the deploy caused by changes to the develop branch will use. Production is the database that the deploy caused by changes to the release branch will use.
  - Define stored procedures in the database that can be called by the Java code to abstract the database details from the Java logic. Learn how to do this here: https://dev.mysql.com/doc/connector-net/en/connector-net-tutorials-stored-procedures.html
    - Essentially you create functions in the database that can be called that return rows or values when passed input parameters. This allows you to hide the tables, fields and relationships between tables from the Java logic. We will learn why this is important later on!
  - Write a re-usable class that allows anyone in the app to call a given (passed as parameter to the constructor) stored procedure:
    - Add the ability to supply parameters of types: String, int, long and double
    - The class should provide some mechanism for returning the results to the calling class (either the rows returned by the stored procedure call, or the data in the rows).
    - **All teams must ensure that you close any connection you make to the database.** Use mechanisms like try-with-resources (http://www.mastertheboss.com/jboss-server/jboss-datasource/using-try-with-resources-to-close-database-connections) or try..catch..finally blocks to ensure that all of your connections are closed. Any team that monopolizes the DB by not releasing connections may be subject to a grade decrement. (See rubric when released)
  - Assist the programmer assigned to the League Object Model epic in the development of database classes that load, save and perform operations for the model.
  - **Note:** If necessary, I will provide the schema and stored procedure class for teams with only 3 programmers, you will be required to run the scripts that deploy the schema on your databases.
- **League Object Model:** *(Assign this task to the person most interested in learning about hockey for now, eventually you will all need to know a little about hockey)*
  - In "business" apps we would call this the "Business Model", these are the classes that represent the various objects your team agrees represent the possible actors and actions in the hockey simulation. All logic in these models is business logic.
  - Take the first pass at defining the classes that will be required to represent the model required to simulate a hockey league, as an initial direction look at the JSON defined in **Appendix A**. This is the data you will import to seed the database at the beginning of the simulation. This data has to go somewhere!
    - Do **not** create anemic domain models: https://www.martinfowler.com/bliki/AnemicDomainModel.html
    - Remember at all times: the purpose of object-oriented programming is to combine data with the methods that operate on that data, pairing them

together into a class that can be passed around and manipulated. An intelligent set of data.

    - Present your design to the team and seek feedback.
    - Implement the models:
        - Write all of the classes and tests backing the classes.
        - Objects should include validation, ensure that the data entered into the model is valid (for example, player names should not be empty). Use your best judgement here.
        - Each object needs to be able to persist itself (save itself somewhere, load from somewhere). Work with the database programmer to define an interface between your model and the DB classes that will perform persistence logic to the database. Plan for change. Maybe your boss will tell you the system has to save to NoSQL or AWS next sprint.
        - First, write mock objects that stand in for the DB persistence classes to test your model.
        - Then, work with the DB programmer to implement all persistence classes that fulfill the other side of the model -> persistence interface.
            - If you are a team of 3 the whole team will need to help with this work.
- **Simulation State Machine:**
    - Write a class that will import the JSON defined in **Appendix A** and instantiate the models defined by the League Object Model programmer(s). This will recreate the league defined in the JSON, but in memory using your model objects. Once the model is recreated in memory, persist the model (save it) to the database using the DB classes and stored procedures written by the DB programmer and League Object Model programmer(s).
    - Write a state machine system capable of:
        - Persisting (saving and loading) itself to a database
        - Nesting state machines inside states (in other words a state encapsulates an internal state machine and is not complete until the internal state machine is complete).
    - Implement the state machine defined in **Appendix B**, the behaviour of each state is defined in the appendix.
    - Implement any input from the command-line required for each state.
    - Implement any output required for each state.

You have a very busy first milestone and I encourage your group not to delay in getting to work immediately. Anyone who finishes early will likely need to assist the programmer working on the League Data Model. Work as a team to accomplish your tasks, but also ensure that work goes into the epic you are responsible for has quality. Be professional, cordial and co-operative.

Additional System / Process Requirements

In addition to the functional requirements above your group project must always adhere to the following system and process requirements:

- Your project is built with Java
- You do not include 3rd party packages/components without permission from the instructor. Tag your instructor in your Teams channel with a request to use a technology and wait for a response before using it.
- Your group uses git, through the repository we create for your group on Dalhousie's Gitlab server (git.cs.dal.ca), and **each individual in the group performs their own git operations using their own account**
- Your group follows the Gitflow branching workflow.
- Your group uses the MySQL variant database specifically assigned to this course (details will be given to your group when it is formed).
- **All students write their own tests for their own code. When a TA determines that code should be covered by unit tests, they will only consider tests to exist and cover code when they were written or updated by the author of that code.**

Milestone 1 Marking Rubric

Milestone 1 is marked differently than milestones 2 and 3. We are still learning the process-based best practices in the course. This milestone is primarily an assessment of your adherence to these process-based best practices.

In this milestone you will be assessed on:

- Your adherence to Agile principles:
  - This is assessed in various ways across the semester, but in this milestone we're focused on the health of the system we're building by working on CI/CD and by writing the software with test-driven development.
- Test-driven development:
  - Your TAs will assess code written for this milestone and assign each of you a grade for test-driven development.
  - 100% of your business logic must be covered by unit tests, achieve this result easily using test-driven development.
- Continuous Integration / Continuous Deployment:
  - Your CI/CD programmer and database programmer will both work to build a system capable of being built and deployed to multiple environments targeting multiple databases.
  - The diligence required to achieve this robustness will be the foundation of your application's quality.

To assess the above, we will calculate your milestone 1 grade as follows:

- **Group portion (33%), the app meets milestone 1's functional requirements (33%):**

    - The executable is deployed to an account on Timberlea.
    - A video you create demonstrates the app running from your designated production account, we see you interacting with the app via command-line and a sample JSON file we provide for input.
        - We will ask you to demonstrate the DB populated with the imported data and created team in a video you record for your milestone submission.
    - The app validates the input and reports any errors it finds.
    - The app prompts the user to create or load a team.
    - The app successfully creates (by prompting the user) or loads the team (verified by looking at your code / database).
    - The app prompts for a number of seasons to simulate.
    - The app enters the simulation state and simulates the given number of seasons (in this milestone this is a no-op, for now just output "Simulated season X for <team name>…"
- **Individual portion (66%), each programmer on the team earns these points in a different way, depending on the epic they have been assigned for the sprint:**
    - **CI/CD Programmer:**
        - (50%) Group has a working continuous integration system that:
            - Builds and tests **any** new branch or commit
                - Your CI/CD must do more than simply execute tests, it must react to test failures and stop the build. We will have you test this and prove that it works when you submit your work for milestone 1.
            - Deploys the executable from successfully built and tested commits to the **develop** branch to the group's designated **test** Timberlea account
            - Deploys the executable from successfully built and tested commits to the **release** branch to the group's designated **production** Timberlea account
            - Appropriately configures the deployed executable for the **test** account such that the app connects to the test database.
            - Appropriately configures the deployed executable for the **production** account such that the app connects to the production database.
        - (16%) The TA and your team is provided with a document or resource that easily tells them how to configure and compile your application on their machine.
    - **Database Programmer:**
        - Environment (33%):
            - Each of your 3 environments is populated with a schema capable of persisting the data model designed by your team.

            - Each CI/CD deploy properly connects the app to the appropriate DB environment **through configuration, not hardcoded values in your Java**
        - Programming (33%):
            - Queries made to the database are done via stored procedures to abstract the database from the Java code.
            - Your team is provided with a class that makes it convenient to call stored procedures and that ensures you close all database connections that are opened:
                - I will be running a script regularly throughout the day on a scheduled job that attempts to connect to the database. If the script cannot make a connection it will then query all of the open connections on the database server. Groups identified as monopolizing database connections will be warned. **If your group is warned the database programmer loses 10% of their milestone 1 grade to a maximum of 30%.** Take this seriously. In the past students have ignored this and denied access to this resource for other students, this is unacceptable for something that can be easily prevented by defensive programming.
    - **League Data Model Programmer:**
        - Functionality (33%):
            - Your model assists in the JSON import process by **validating the JSON values**, in coordination with the Simulation State Machine Programmer the end result is an in-memory representation of the league via your data model classes.
            - You correctly persist the state of your in-memory model to the database through a set of database classes, separated by an interface and used by the model objects through dependency injection.
        - Test-driven development (33%):
            - Your logic is covered 100% by unit tests
            - Anything that needs to be mocked to properly to write tests is mocked (database classes, etc.). Mock objects are well written and appropriate.
    - **Simulation State Machine Programmer:**
        - Functionality (33%):
            - Your logic correctly parses the input JSON and reports errors in the JSON structure. Your import process uses the classes created by the League Data Model Programmer to create an in-memory representation of the league. Your logic drives the import validation process (uses the methods written by the model programmer) and performs output / control flow.

    - Your state machine has all of the states in the state diagram and moves through them appropriately.
    - Your state machine supports nested internal state machines in any state.
    - Your app performs command-line input and output appropriately, you are responsible for the harness and main() of this program that kicks off your state machine. Each state should perform input and output in a way that abstracts it from the fact that it is communicating with the command-line. You will learn about this in the TDD lecture.
- Test-driven development (33%):
    - Your logic is covered 100% by unit tests.
    - Anything that needs to be mocked to properly to write tests is mocked (database classes, etc.). Mock objects are well written and appropriate.

Appendix A – JSON Import Data

Rob's handy tip for new programmers: Any time you are told to import some kind of JSON or XML data **PLAN FOR CHANGE**. Nobody gets these things right the first time. It usually takes any system like this several iterations before the final state of the input file is determined. This will most definitely be the case here. We will begin the simulation with a simplistic representation of the league and evolve the complexity in further milestones. This has far reaching implications for your project's design, it means almost everything in the project will change over time.

If you don't know what JSON is: https://www.w3schools.com/js/js_json_syntax.asp

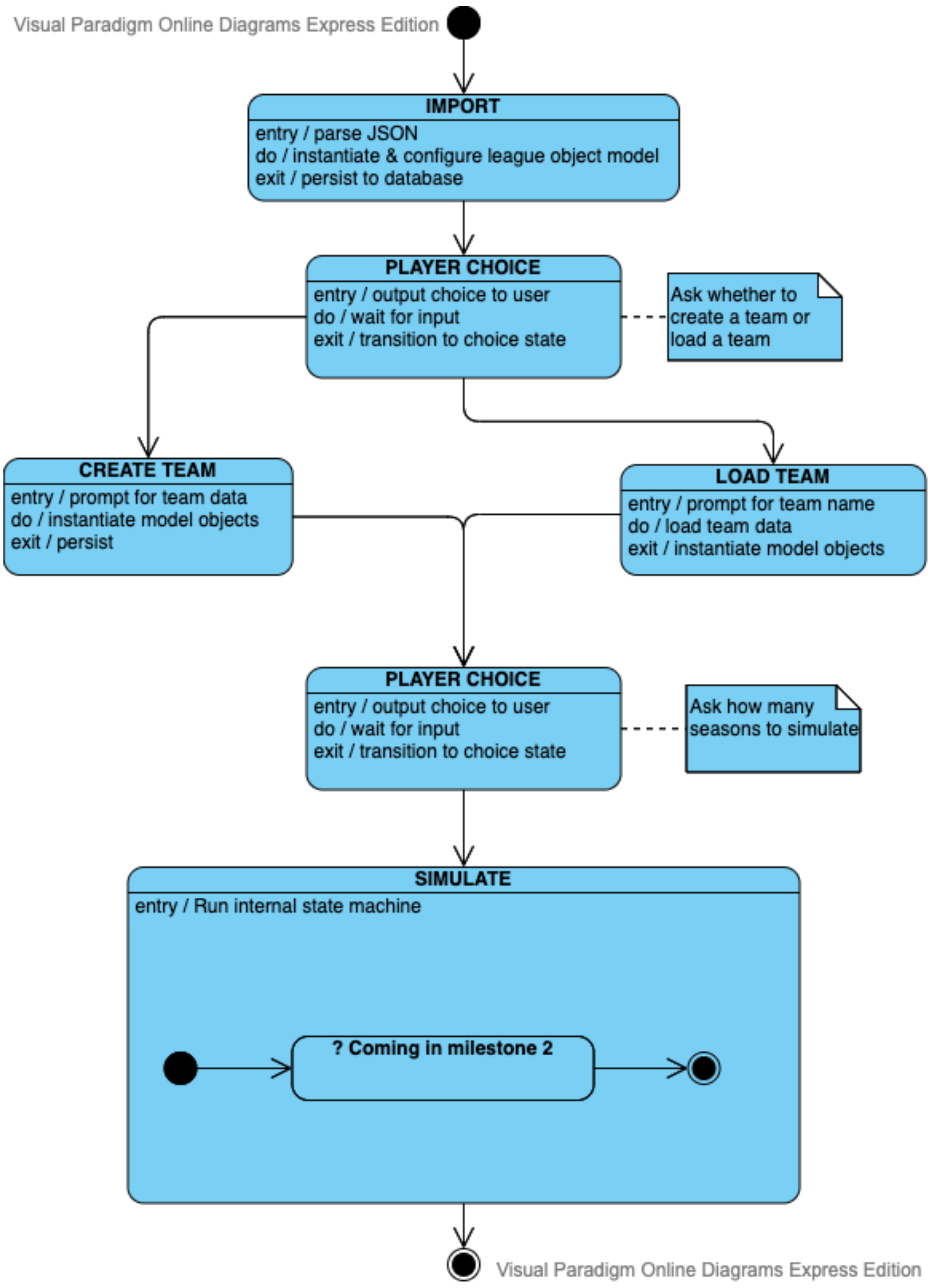
The following JSON defines the structure of the import file:

```
{
  "leagueName":"Dalhousie Hockey League",
  "conferences":[
    {
      "conferenceName":"Eastern Conference",
      "divisions":[
        {
          "divisionName":"Atlantic",
          "teams":[
            {
              "teamName":"Boston",
              "generalManager":"Mister Fred",
              "headCoach":"Mary Smith",
              "players":[
                {
                  "playerName":"Player One",
                  "position":"forward",
                  "captain":true
                },
                {
                  "playerName":"Player Two",
                  "position":"defense",
                  "captain":false
                },
                {
                  "playerName":"Player Three",
                  "position":"goalie",
                  "captain":false
                }
              ]
            }
          ]
        }
      ]
    }
  ],
  "freeAgents":[
    {
      "playerName":"Agent One",
      "position":"forward",
      "captain":false
    },
    {
      "playerName":"Agent Two",
      "position":"defense",
      "captain":false
    },
    {
      "playerName":"Agent Three",
      "position":"goalie",
      "captain":false
    }
  ]
}
```

Note the following:

- []'s denote arrays. JSON does not specify array dimensions. You should assume that a league can have any even number of conferences, each conference has an even number of divisions, and each division can have any number of teams. Teams have 20 players.
- Where you see a string value, assume that it should have validation to ensure it is not an empty string.
- "position" items should have values: "goalie", "forward" or "defense" only
- "captain" must be true or false, only one player can be captain on a team
- "freeAgents" is a list of players not assigned to a team, we will need those in milestone 2 when the user hires their initial players.

Appendix B – State Machine


Visual Paradigm Online Diagrams Express Edition
IMPORT
entry / parse JSON
do / instantiate & configure league object model
exit / persist to database
PLAYER CHOICE
entry / output choice to user
do / wait for input
exit / transition to choice state
Ask whether to create a team or load a team
CREATE TEAM
entry / prompt for team data
do / instantiate model objects
exit / persist
LOAD TEAM
entry / prompt for team name
do / load team data
exit / instantiate model objects
PLAYER CHOICE
entry / output choice to user
do / wait for input
exit / transition to choice state
Ask how many seasons to simulate
SIMULATE
entry / Run internal state machine
? Coming in milestone 2
Visual Paradigm Online Diagrams Express Edition

The UML state diagram above represents the state machine for the simulation for milestone 1. Every state lists actions that occur when the state is entered (entry), the action performed in the state (do) and the actions that occur when the state is exited (exit).

IMPORT

This state is responsible for importing the JSON from a file. The state will parse the JSON and instantiate all of the game objects required to represent the league defined in the JSON in memory.

The import process must use the league domain model objects to validate the data that comes out of the JSON. **If the data is invalid the simulation should abort and inform the user of the problem.** The better you do this (the easier you make it to debug problems with JSON, for example outputting the line or specific item that has a problem, the more points you will earn.

PLAYER CHOICE

This state should be programmed for reuse to present a choice to a user on the command line, wait for user input on the command line, and then act on that input (transition to a single state and pass the input as a parameter, or chose between multiple transition states based on the input value).

CREATE TEAM

This state should walk the user through a series of prompts to populate all of the required data for a team. Prompt the user, perform validation using the league domain model objects. If input is invalid continue prompting until valid data is entered.

When you have collected all necessary information the state should instantiate the necessary domain model objects required to begin simulation.

On exit of this state the state of the league in memory (including the new team that has just been created and inserted into the league) should be saved.

Information you must prompt for: (Refer to this link on the NHL to understand what conferences and divisions are)

- Conference the team belongs in (must be one of the existing conferences defined in the import)
- Division the team belongs in (must be one of the existing divisions in the given conference defined in the import)
- Name of the team (not empty)
- Name of the general manager
- Name of the head coach

Teams are created without players, the first thing we will do in Milestone 2 is provide a way for a new team to hire and trade for players.

LOAD TEAM

This state should perform the same instantiation of domain model objects performed in the CREATE TEAM state, however rather than using input to generate the player's team this state should prompt the user for the name of an existing team to load from the save.

If no team with the given name exists, output an error message and end the simulation. The prompt should not be case sensitive.

SIMULATE

This state will make use of your nested state implementation to perform the actual simulation in later milestones. On entry it should run the internal state machine and wait until the machine has completed. You will want to design this so that you can embed state machines to any depth. Do not make this multithreaded, stick to a single thread.

For milestone 1 the simulation is a no-op, however it should demonstrate that it is capable of performing nested states by nesting some fake states that simply do some output to show off and then terminate.

On exit you should persist (save) the state of the league.