Group Project Requirements Specification

Milestone 2

Many students just briefly scan this document, this inevitably leads to missed requirements and lost grades. Don't let this happen to you, or your group! Read this document carefully and ask questions if you have any concerns or questions. Assumptions are dangerous, do not make them, clarify!

Disclaimer

I will continue to make mistakes from time to time, and so *I continue to 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.*

Progress so far

We're getting into the hard part of the course now! You have learned several process-based practices that will help your team to achieve a high level of quality. You are using test-driven development to develop your code, so it comes well tested. Most groups have learned how to use continuous integration to ensure your build is healthy and stays healthy.

At this point your application and codebase should have the functionality described in milestone 1's requirements documentation.

If your project is missing a requirement for milestone 1 you must complete that work first. If your group has to put in extra work to make up for missed requirements, you'll need to do that work now, as this work is not considered in the effort required to implement the requirements for milestone 2.

Application Description

We continue to develop our hockey simulation system. We are striving to build a module that can be plugged into other systems that will perform the simulation of a season of hockey for a hockey league. Our module receives input (via imported JSON, and where necessary by prompting the user), initializes its "league" data model, and then performs simulation producing output (a persisted league in a database, and where necessary output to the console). We aren't going to the same level of detail as the real thing, but we're developing our system in a way that theoretically with more work we could plug your module into the real thing and have this mode added to a video game simply by creating an interface for it, and providing a new mechanism for persistence to replace the DB.

Expect and program for change

I want you to experience the devastating impact of change to poorly made systems. Already we see some change in milestone 2. You must do your best to adhere to the principles, processes and best practices taught in this course if you want to survive the changes that will come in milestone 3.

Milestone 1 Requirements

Note: One way to coordinate your efforts on this milestone is to first agree to the changes you will make to the classes already defined in your league data model. Then, each programmer should stub those changes out in their feature branches and merge those feature branches to develop. This will allow keep other programmers from being blocked as you go back to working in your feature branch to complete the functionality. This is not the only way, communicate with each other to determine how you resolve dependencies.

General requirements for the whole team:

- All code you write in this milestone, except for the classes below, will be in the **business layer** (to do with the business of simulating a season of hockey) and therefore must be implemented using test-driven development (TDD) such that they are 100% covered by unit tests.
    - **Exceptions:**
        - Small input/output classes in your **presentation layer** that deal with communicating with the user of the simulation system via the command line.
        - Small persistence classes in your **data layer** that implement a persistence interface for **business layer** classes to persist themselves to the database using your stored procedure class from milestone 1.
    - **Remember, you must write your own tests.** The easiest way to ensure this is to actually use test-driven development and write your tests first. Done properly, this way of working guarantees 100% on TDD for you because code won't exist without being covered by a test!
- Continue to monitor the health of your system and follow best practices:
    - **Use Gitflow**. Do not recklessly commit to master, we need master clean to evaluate your code this milestone.
    - **Maintain your CI/CD system** and keep your tests passing.
    - **Don't leave things to the last minute.** Work at a steady pace, 10 hours per week throughout the milestone so that your fellow group members can develop their work alongside you. Though you are marked individually for the most part you are not working in a vacuum, other group members will depend on the classes you write. Be a good group member. This milestone we will add a

professionalism element to the marking rubric to encourage you to maintain a steady pace (Principle 8 of Agile methodology).

- There is no one programmer assigned to database programming. *The database is now a detail.* We aren't that worried about it to be honest; the DB is just a means to an end. *Each programmer will write their own classes in the **data layer** when they need to persist their objects.* This means writing your own stored procedures to call as necessary. Do not assign this work to a single programmer or that programmer will unlikely be able to earn grades for TDD this milestone.
- You will notice the addition of a "gameplayConfig" section in the JSON (Refer to Appendix A). This section is designed to provide input into the system to configure AI decision making and eventually to control configurable business logic. Several programmers in the group will have tasks that depend on this section of the JSON. Coordinate with each other to develop the import process for this piece. Perhaps assign this task to the person with the least amount of work when you set up your sprint board.
- Any modifications to the JSON import should be validated as appropriate, continue to ensure the JSON you are importing is valid both syntactically and for the data values themselves.

Your TA will assign each of you to one of the following epics. You are responsible for all tasks in your assigned epic.

- **Game Statistics:**
    - Epic goal: To add general manager, coach, and player statistics to the system to more closely tie the simulation to the real world. Teams with good general managers are more successful in acquiring advantageous trades. Teams with good coaches are more successful over the long term as the coaches develop their player's skills through practice. Finally, teams with good players win more games than teams with poor players.
    - Add player statistics to the league data model:
        - To distinguish good players from bad we will add 5 stats to the system for all players:
            - **Age:** Age catches up with us all, and hockey is arguably the most physical non-combat sport in the world. Unfortunately, the older a player is the more likely they are to become injured, and the less likely they are to improve with training. Eventually players have to retire.
            - **Skating:** This stat reflects the athlete's strength and speed at skating. The higher this stat, the faster and more agile the athlete is on the ice.
            - **Shooting:** This stat reflects the athlete's stick handling skills and ability to score goals. The higher this stat, the more likely a player's shot will score a goal.

- **Checking:** This stat reflects the athlete's ability to check other players. "Checking in ice hockey is any of a number of defensive techniques aimed at disrupting an opponent with possession of the puck or separating them from the puck entirely. Most types are not subject to penalty." The higher this stat, the more likely a player will be to disrupt a goal scoring attempt.
- **Saving:** This stat reflects the athlete's ability to prevent a goal. This stat is used only by goalies. The higher this stat, the more difficult it becomes to score on the goalie.

        - All player statistics have a value between 1 and 20.
        - Modify the JSON import process to support the addition of player statistics to the player and free agent objects. Refer to Appendix A.
    - Add coaching statistics to the league data model:
        - Coaches will share similar statistics as players.
        - Coaches improve athletes over time through training and practice. In our simulation each team only has one coach, yet in a real hockey team there are many coaches responsible for different aspects of improving the players. In our simulation coach statistics represent a training skill in that particular area. The higher the value, the better that coach will be at improving that particular skill for the players on the team.
        - Coach statistics have a value between 0 and 1 (float), representing a percentage of effectiveness at improving the associated skill. A coach with a stat value of 1 will always improve the player's skill in that area. A value of 0 will never improve the player's skill in that area.
        - Modify the JSON import process to adapt to the new format for defining coaches (Refer to Appendix A).
        - Modify the JSON import process to also import an array of coaches that can be hired by the player (Refer to Appendix A). Depending on the foresight of your database implementation this could mean separating the coach name from your team table in your database, and then creating a new table for coaches.
    - Implement aging:
        - Modify your system to add a method to your player class to age a player a given amount of time. Note, the caller of this method will likely be calling it a frequency of days vs. months or years of time elapsed.
            - The gameplayConfig section includes an "aging" object that defines the average retirement age, and maximum age of a player (Refer to Appendix A).
            - When this method is called determine whether the player retires. Create a decision-making algorithm that **slowly** increases the likelihood of retirement as a player approaches the average retirement age, and greatly increases the likelihood as a player passes the average retirement age.
            - If a player reaches the maximum age they automatically retire.

        - When a player retires the team must replace that player from the free agent list. Write a decision-making algorithm that chooses the best player to replace the retired player with. The retired player is removed from the league data model. The replacement player must share the same position as the retired player.
        - Aging slowly removes injuries (see below).
    - Implement a simple injury system:
        - After each game is simulated for a team, for each player on that team determine whether the player was injured in that game.
        - The gameplayConfig section includes an "injuries" object that defines a random injury chance, and a range of days that a player should be injured for. When a player is checked for an injury after a game use the random injury chance value, and if an injury is warranted determine a length of time for the player to be considered injured by a random value between the low and high injury date range.
        - As the player is aged this injured status should be removed when the number of injury days for the player have elapsed.
        - We will assume that an injured player cannot be reinjured.
    - Calculate team strength:
        - There is a strong correlation between quality of players and points scored by NHL teams. It seems in hockey having the star players matters.
        - Add a method to the appropriate class in your system to calculate a team's overall strength. Use the following logic:
            - A player's strength depends on their position:
                - **Forward:** Skating + Shooting + (Checking / 2)
                - **Defense:** Skating + Checking + (Shooting / 2)
                - **Goalie:** Skating + Saving
            - A team's strength is the total of all player's strengths on the team. **Injured** players contribute only 50% of their strength to the team.
- **Enhanced Team Creation & Team Logic:**
    - Epic goal: To add depth to the simulation by giving the user more control over their team. We allow the user to hire a general manager and coaches from a list, and then to populate their initial roster from the free agent list. Coaches will train their players and increase their statistics over time. Additionally, this programmer will add a mechanism to output the state of the league to JSON on request.
    - Modify the JSON import process to remove the "captain" field from the "freeAgents" list (Refer to Appendix A). This field makes no sense for free agents that are not attached to a team.
    - Expand user choices when the user is creating their team:
        - Continue to prompt for the team name and the conference and division to put the team in.
        - Display a list of potential general managers from the new list of available starting general managers in the import JSON (Refer to Appendix A).

        - Allow the user to pick from the list of general managers.
        - Display a list of coaches to fill the head coach position on the team. Ensure you display the coaching stats for the coach so the user can choose the stats they most want to improve.
        - Allow the user to pick a head coach from the list of coaches.
        - Use your **presentation layer** to display a list of free agents to the user. Include all player details in an organized fashion. Make the list easy to scan for the user to identify good players from bad.
        - Allow the user to choose the players for their team from the list of free agents.
            - A team must have 18 skaters (forwards and defense) and 2 goalies. Do not allow the user to proceed until they have this condition met.
    - Implement training:
        - The gameplayConfig section of the import JSON includes a "training" object that defines the number of days until the chance of players on teams having their stats increased through practice and training (Refer to Appendix A).
        - Work with the league simulation programmer to implement a system that is triggered after the number of days until a stat increase check have passed. When triggered:
            - For every player on the team:
                - For every statistic:
                    - Generate a random value between 0 and 1.
                    - If the random value is < head coach's training stat for this area the player's stat increases by 1.
                    - If the random value is > head coach's training stat for this area the player risks an injury from overtraining, use the same mechanism used by the game simulation to trigger an injury check for the player.
    - Serialize the league data model to a JSON file:
        - When requested by the simulation system, output the entire state of the league data model to a text file on the filesystem in JSON format. The format should have a similar structure to that of the import JSON, except with your various fields / model objects represented appropriately. It does not have to match. It should be such that were you to **de**serialize this file you would then have a perfect in memory version of the league.
- **League Simulation:**
    - Epic goal: To implement the internal state machine for one season of a hockey league. We introduce the concept of time to the system, and the state of the league changes over time. The simulation directs the overall flow of the app and drives other components in the system.
    - Introduce the concept of time:

        - When leagues are first created in memory, set the current date to September 30th of the current year.
        - From then on, as the simulation progresses, ensure that you track how much time has elapsed. The smallest unit of time we will concern ourselves with in this system is 1 day. The system should theoretically allow the player to simulate their league forever.
        - When the league data model is persisted to the DB you should save the current date. When simulation resumes in further execution it will proceed from the last date simulated.
        - Remember, time passes separately for every team/league combination the user has created in the database. Only one league is simulated at a time.
    - Implement a team standings system:
        - Teams earn points by winning games. Winning a game earns a team 2 points.
        - Track all team wins and losses throughout simulated seasons. When teams win, increase their points.
        - Your team standings system should be able to determine team rankings in various formats (ranking across entire league, ranking in conference, ranking in division).
    - Implement a scheduling system:
        - At the start of a season the scheduling system determines the dates for most of the major events in a season of hockey in a major league like the NHL:
            - **October 1st:** Regular season starts (the first games of the regular season can be scheduled on this date).
            - **Last Monday in February:** Trade deadline. No trading is allowed after this date.
            - **First Saturday in April:** End of the regular season, after all games are played on this date determine the round 1 seedings for the playoffs. We will assume that all leagues use the Stanley Cup playoff format.
            - **Second Wednesday in April:** Playoffs begin.
            - **June 1st:** Last possible day for the Stanley Cup final.
        - Also at the start of a season the scheduling system determines the dates for all regular season games between teams:
            - Teams play 82 games per regular season.
            - ~1/3rd of their games should be against teams in their division.
            - ~1/3rd of their games should be against teams in their conference, but not in their division.
            - ~1/3rd of their games should be against teams outside of their conference.

        - Ensure that every team plays every other team at least once in the regular season.
    - **Modify your state machine to alter existing states and add new states defined in the state diagram in Appendix B**. Appendix B describes the entry, do and exit actions for each state. Be sure you understand everything about this diagram, if you do not, clarify your understanding with your TA or Rob.
- **Trading: (Omit for groups of 3)**
    - Epic goal: To implement the trading system in the simulation. AI controlled teams will initiate trades between each other's teams. AI will initiate trade offers to the user, which the user can accept or reject. The user can initiate a trade offer to an AI controlled team, which the AI will either accept or reject.
    - Implement player trading between teams:
        - Two teams can trade players
        - Any number of players can be exchanged up to the "maxPlayersPerTrade" value defined in the gameplayConfig's "trading" object (Refer to Appendix A).
        - Teams cannot exceed 20 players. After any trade negotiation is resolved both teams must drop excess players to the free agent list:
            - AI-controlled teams will drop their weakest players
            - When the user ends up with more than 20 players they must be prompted to request which players to drop to the free agent list.
            - All teams must retain 18 skaters and 2 goalies.
        - Teams cannot have fewer than 20 players. After any trade is resolved both teams must ensure they are at 20 players:
            - AI-controlled teams will hire the best skaters or goalies from the free agent list to reach 18 skaters and 2 goalies.
            - The user must be shown the free agent list and select replacement players. They too must ensure 18 skaters and 2 goalies.
    - Implement AI trade offers:
        - The gameplayConfig section of the import JSON (Refer to Appendix A) has a "trading" object that defines config values that tune the trading system:
            - lossPoint: This value determines the minimum number of game losses an AI-controlled team must suffer before considering a trade. Once a trade is made, reset the team's count before considering a further trade.
            - randomTradeOfferChance: This value determines the likelihood of a trade offer once a team reaches the loss point.
            - maxPlayersPerTrade: AI-controlled teams will not generate trade offers with more players than this value.
            - randomAcceptanceChance: This is a value you can use in the trading algorithm you create to introduce an element of randomness. Use this value to randomly accept some trades, no

matter whether they are advantageous for the AI-controlled team.

        - Once per day during the regular season (before the trade deadline) the simulation should call a method to generate trade offers:
            - For every AI-controlled team in the league calculate their loss point.
            - If a team has reached their loss point generate a random value between 0 and 1. If the random value is less than the random trade offer chance, generate a trade offer for that team.
    - Design and implement a trade accept/reject algorithm. AI-controlled teams should attempt to improve the overall strength of their team by trading their weakest players for stronger players. AI-controlled teams should attempt to prevent the reduction of their team strength by rejecting trades that reduce their overall team strength. An element of randomness should be introduced as per the randomAcceptanceChance value.
    - If an AI-controlled team makes a trade offer to the user-controlled team, use your presentation layer to display the details of the trade (players offered and their statistics, players requested and their statistics) and then prompt the user whether to accept or reject the trade. If they accept the trade, they must adjust their team roster up or down to the 20 player (18 skaters, 2 goalies) limit. Dropped players are sent to the free agent list. The user can add players using the same mechanism used when creating their team to hire from the free agent list.

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 2 Marking Rubric

Professionalism Grade Cap
This milestone the grade you can achieve **as an individual** is capped by the level of professionalism you demonstrate in your group. To earn 100% in this milestone you must deliver all of the tasks you are assigned at a steady pace (some work done each week). Leaving tasks incomplete or leaving the work until the last minute results in a reduced maximum grade for your work.

Grade Cap	Criteria
100	Completed **all tasks** at a reasonable pace as determined by your TA (33% each of the 3 weeks of the milestone).
90	Completed all tasks by the milestone deadline (11:59PM!)
70	Did not complete all tasks.
50	Completed 50% or less of assigned tasks.

Monitoring the health of your system, and maintaining process best-practices (10%)

- **CI/CD: 5%**
  - Your team has a working CI/CD system throughout the milestone. Your TA will evaluate this periodically. If they click "Run pipeline" and the pipeline fails you will lose points.
- **Gitflow: 5%**
  - Your team adheres to the Gitflow git workflow.

Assessment (90%)
The remaining 90% of your milestone 2 grade is determined by the results of an assessment of your code by your TA.

This means you must focus on learning, understanding and applying the best practices taught to you before and during this milestone.

Make sure your classes do not violate S.O.L.I.D. principles. Use test-driven development. Evaluate and adjust your packaging to ensure your packages adhere to the principles of cohesion and coupling. During the cohesion and coupling lecture you will learn that this means you want to base your packaging decisions on a 50/50 split between the Common Closure Principle and the Common Reuse Principle. You should understand what I mean by that sentence when we have covered this material. Finally, you should ensure that your code does not violate any layer boundaries or clean code practices.

Remember, you can use the assessment spreadsheet as a check to ensure you are submitting quality work! Assess yourself! Since you will need to do this, you might as well practice.

Refer to the milestone 2 assessment spreadsheet for details.

Appendix A – JSON Import Data

The following JSON defines the structure of the import file:


```
{
  "leagueName":"Dalhousie Hockey League",
  "gameplayConfig":{
    "aging":{
      "averageRetirementAge":35,
      "maximumAge":50
    },
    "gameResolver":{
      "randomWinChance":0.1
    },
    "injuries":{
      "randomInjuryChance":0.05,
      "injuryDaysLow":1,
      "injuryDaysHigh":260
    },
    "training":{
      "daysUntilStatIncreaseCheck":100
    },
    "trading":{
      "lossPoint":8,
      "randomTradeOfferChance":0.05,
      "maxPlayersPerTrade":2,
      "randomAcceptanceChance":0.05
    }
  },
  "conferences":[
    {
      "conferenceName":"Eastern Conference",
      "divisions":[
        {
          "divisionName":"Atlantic",
          "teams":[
            {
              "teamName":"Boston",
              "generalManager":"Mister Fred",
              "headCoach":{
                "name":"Mary Smith",
                "skating":0.5,
                "shooting":0.8,
```

```
            "checking":0.3,
            "saving":0.5
          },
          "players":[
            {
              "playerName":"Player One",
              "position":"forward",
              "captain":true,
              "age":27,
              "skating":15,
              "shooting":18,
              "checking":12,
              "saving":0
            },
            {
              "playerName":"Player Two",
              "position":"defense",
              "captain":false,
              "age":20,
              "skating":10,
              "shooting":10,
              "checking":10,
              "saving":0
            },
            {
              "playerName":"Player Three",
              "position":"goalie",
              "captain":false,
              "age":33,
              "skating":10,
              "shooting":4,
              "checking":9,
              "saving":18
            }
          ]
        }
      ]
    }
  ]
],
"freeAgents":[
  {
    "playerName":"Agent One",
```

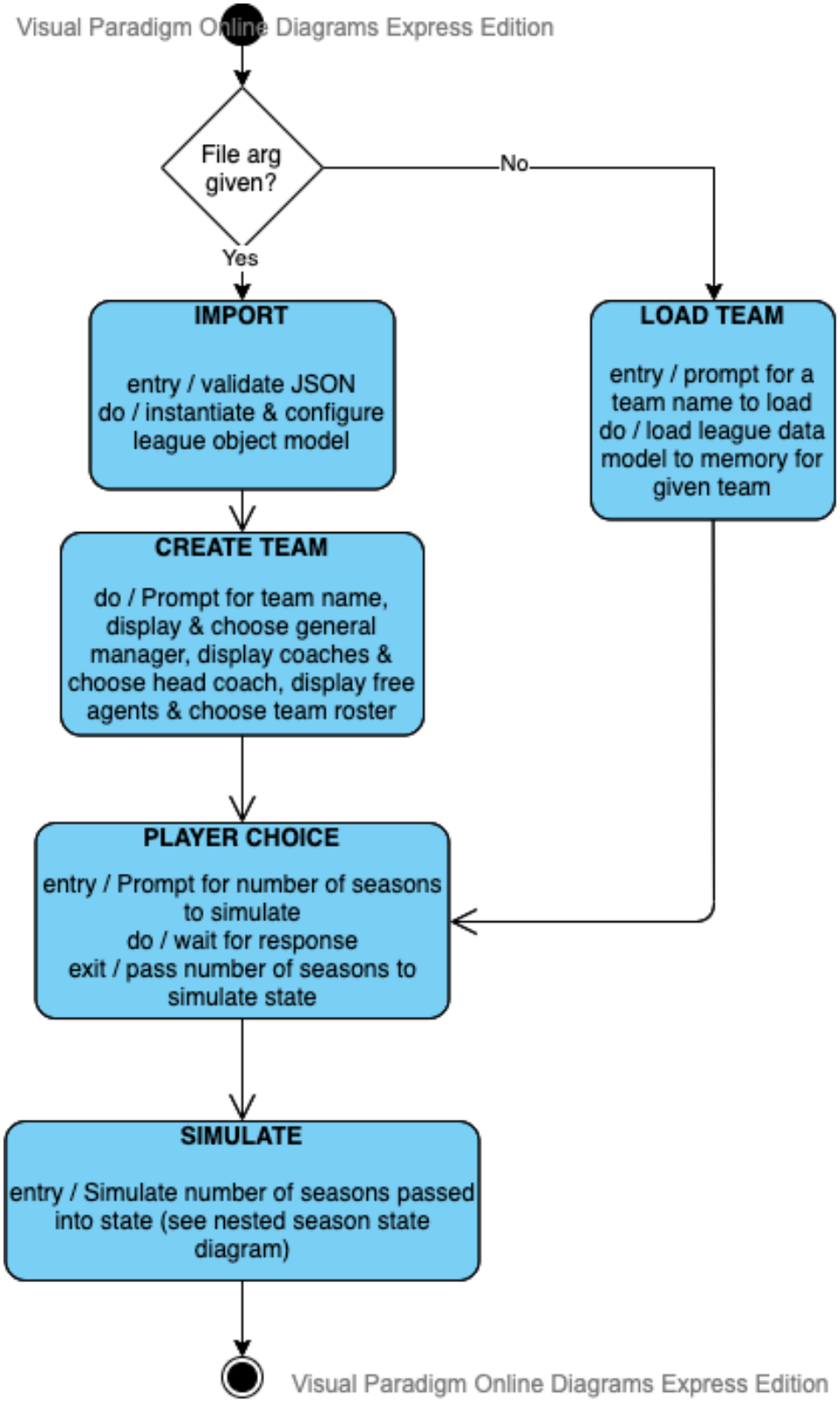
```
      "position":"forward",
      "age":25,
      "skating":10,
      "shooting":10,
      "checking":10,
      "saving":0
    },
    {
      "playerName":"Agent Two",
      "position":"defense",
      "age":25,
      "skating":10,
      "shooting":10,
      "checking":10,
      "saving":0
    },
    {
      "playerName":"Agent Three",
      "position":"goalie",
      "age":25,
      "skating":10,
      "shooting":5,
      "checking":5,
      "saving":10
    }
  ],
  "coaches":[
    {
      "name":"Joe Smith",
      "skating":0.5,
      "shooting":0.8,
      "checking":0.3,
      "saving":1.0
    },
    {
      "name":"Frank Smith",
      "skating":0.5,
      "shooting":0.8,
      "checking":0.3,
      "saving":0.5
    },
    {
      "name":"Samantha Smith",
      "skating":1.0,
```

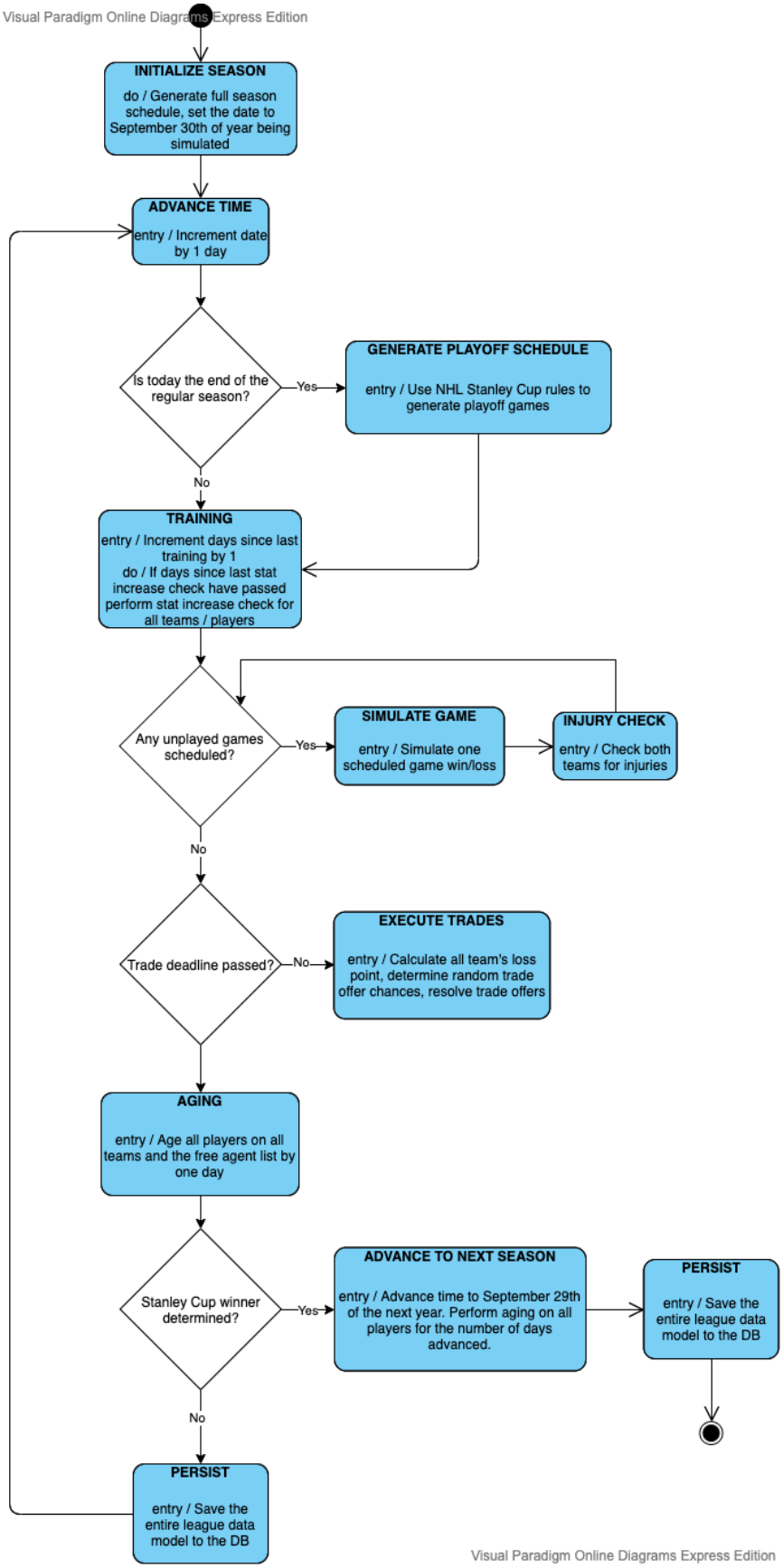
```
      "shooting":0.5,
      "checking":0.5,
      "saving":0.0
    }
  ],
  "generalManagers":[
    "Karen Potam",
    "Joseph Squidly",
    "Tom Spaghetti"
  ]
}
```

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 that can be hired to fill holes on teams caused by team creation, retirement or trading.

Appendix B – State Machines





Notes:

The UML state diagrams above represent the state machines for the simulation for milestone 2. 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).

The first state diagram shows the state machine for the entire system. This has changed little from milestone 1 other than to factor in the changes we made midway through milestone 1.

The second state diagram shows the internal state machine you must nest inside the **SIMULATE** state.

INITIALIZE SEASON

This state initializes the schedule for the regular season. It sets up whatever classes you have designed to track time and scheduled games. It also sets the current date to September 30th so that when the simulation begins we will begin on October 1st, the first day of the regular season.

ADVANCE TIME

This state advances time by 1 day.

GENERATE PLAYOFF SCHEDULE

This state generates the playoff schedule of games at the end of the regular season. It determines the league rankings and then seeds the playoff spots as per the rules of the Stanley Cup.
https://en.wikipedia.org/wiki/Stanley_Cup_playoffs

TRAINING

This state tracks the time elapsed since the system last performed a training check on all of the players. Remember that free agents are not trained because they are not on a team. Only players assigned to a team when training is calculated can potentially receive stat increases.

SIMULATE GAME

This state simulates the results of a game. For now we will use a simplistic formula to determine the winner of a game:

- Calculate team strength for team A
- Calculate team strength for team B
- The winner is the team with the greater total strength
- Use the "gameResolver" object defined in the gameplayConfig to randomly override the team strength check to allow for random upsets. Generate a random value between 0 and 1, if it is < the random win chance then reverse the results of the game.

INJURY CHECK

Check all players involved in the game for injuries.

EXECUTE TRADES

Evaluate loss points and generate trade offers.

AGING

All players in the league age by 1 day. This will slowly heal injuries and progress players towards retirement.

ADVANCE TO NEXT SEASON

Once the Stanley Cup is resolved we can simulate directly to September 29th, one day before the beginning system date. We need to do this in order to correctly age the players outside of the regular season.

PERSIST

To avoid loss of data we should persist the state of the league after each day of simulation and at the end of simulating a season of play. This state instructs the entire league data model to persist itself through whatever mechanism you create.

Resources

National Hockey League:
https://en.wikipedia.org/wiki/National_Hockey_League

This video shows the NHL'06 Dynasty Mode in action, if you are still uncertain what exactly we're building here this will give you a good idea of what a fully functioning version of what we are building would do and look like. We are not building the interface for this system, but the back end logic that drives it.
https://www.youtube.com/watch?v=lJPppnVbTpU

Stanley Cup Playoff Format
https://en.wikipedia.org/wiki/Stanley_Cup_playoffs