

# Module Interface Specification for Software Engineering

Team 6, Pitch Perfect

Damien Cheung

Jad Haytaoglu

Derek Li

Temituoyo Ugborogho

Emma Wigglesworth

January 17, 2025

# 1 Revision History

Date	Version	Notes
Jan 3	1.0	Added MIS for TeamT, GameT, PlayerT, Backend
Jan 14	1.1	Added MIS for Season and Standing Record Modules
Jan 17	1.2	Added sections 2,3,4

## 2 Symbols, Abbreviations and Acronyms

See SRS Documentation at <https://github.com/dcheung11/team-6-capstone-project/blob/main/docs/SRS-Volere/SRS.pdf>

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
MIS	Module Interface Specification
OS	Operating System
R	Requirement
SRS	Software Requirements Specification
UC	Unlikely Change
RBAC	Role-Based Access Control
JSON	JavaScript Object Notation
API	Application Programming Interface

# Contents

<b>1</b>	<b>Revision History</b>	<b>i</b>
<b>2</b>	<b>Symbols, Abbreviations and Acronyms</b>	<b>ii</b>
<b>3</b>	<b>Introduction</b>	<b>1</b>
<b>4</b>	<b>Notation</b>	<b>1</b>
<b>5</b>	<b>Module Decomposition</b>	<b>2</b>
<b>6</b>	<b>MIS of User Interface Module (M1)</b>	<b>4</b>
6.1	Module . . . . .	4
6.2	Uses . . . . .	4
6.3	Syntax . . . . .	4
6.3.1	Exported Constants . . . . .	4
6.3.2	Exported Access Programs . . . . .	4
6.4	Semantics . . . . .	4
6.4.1	State Variables . . . . .	4
6.4.2	Environment Variables . . . . .	5
6.4.3	Assumptions . . . . .	5
6.4.4	Access Routine Semantics . . . . .	5
6.4.5	Local Functions . . . . .	5
<b>7</b>	<b>MIS of Scheduling Module</b>	<b>5</b>
7.1	Module . . . . .	5
7.2	Uses . . . . .	5
7.3	Syntax . . . . .	5
7.3.1	Exported Constants . . . . .	5
7.3.2	Exported Access Programs . . . . .	6
7.4	Semantics . . . . .	6
7.4.1	State Variables . . . . .	6
7.4.2	Environment Variables . . . . .	6
7.4.3	Assumptions . . . . .	6
7.4.4	Access Routine Semantics . . . . .	6
7.4.5	Local Functions . . . . .	7
<b>8</b>	<b>MIS of PlayerT Module</b>	<b>8</b>
8.1	Module . . . . .	8
8.2	Uses . . . . .	8
8.3	Syntax . . . . .	8
8.3.1	Exported Constants . . . . .	8
8.3.2	Exported Access Programs . . . . .	8

8.4	Semantics . . . . .	8
8.4.1	State Variables . . . . .	8
8.4.2	Environment Variables . . . . .	8
8.4.3	Assumptions . . . . .	9
8.4.4	Access Routine Semantics . . . . .	9
8.4.5	Local Functions . . . . .	9
<b>9</b>	<b>MIS of GameT Module</b>	<b>10</b>
9.1	Module . . . . .	10
9.2	Uses . . . . .	10
9.3	Syntax . . . . .	10
9.3.1	Exported Constants . . . . .	10
9.3.2	Exported Access Programs . . . . .	10
9.4	Semantics . . . . .	10
9.4.1	State Variables . . . . .	10
9.4.2	Environment Variables . . . . .	10
9.4.3	Assumptions . . . . .	10
9.4.4	Access Routine Semantics . . . . .	11
9.4.5	Local Functions . . . . .	12
<b>10</b>	<b>MIS of TeamT Module</b>	<b>12</b>
10.1	Module . . . . .	12
10.2	Uses . . . . .	12
10.3	Syntax . . . . .	12
10.3.1	Exported Constants . . . . .	12
10.3.2	Exported Access Programs . . . . .	12
10.4	Semantics . . . . .	13
10.4.1	State Variables . . . . .	13
10.4.2	Environment Variables . . . . .	13
10.4.3	Assumptions . . . . .	13
10.4.4	Access Routine Semantics . . . . .	13
10.4.5	Local Functions . . . . .	14
<b>11</b>	<b>MIS of Backend Module</b>	<b>14</b>
11.1	Module . . . . .	14
11.2	Uses . . . . .	14
11.3	Syntax . . . . .	15
11.3.1	Exported Constants . . . . .	15
11.3.2	Exported Access Programs . . . . .	15
11.4	Semantics . . . . .	15
11.4.1	State Variables . . . . .	15
11.4.2	Environment Variables . . . . .	15
11.4.3	Assumptions . . . . .	16

11.4.4	Access Routine Semantics	16
11.4.5	Local Functions	17
11.4.6	ERD for Database	18
<b>12</b>	<b>MIS of Waiver Module</b>	<b>18</b>
12.1	Module	18
12.2	Uses	19
12.3	Syntax	19
12.3.1	Exported Constants	19
12.3.2	Exported Access Programs	19
12.4	Semantics	19
12.4.1	State Variables	19
12.4.2	Environment Variables	19
12.4.3	Assumptions	19
12.4.4	Access Routine Semantics	19
12.4.5	Local Functions	20
<b>13</b>	<b>MIS of Notification Module</b>	<b>20</b>
13.1	Module	20
13.2	Uses	20
13.3	Syntax	20
13.3.1	Exported Constants	20
13.3.2	Exported Access Programs	21
13.4	Semantics	21
13.4.1	State Variables	21
13.4.2	Environment Variables	21
13.4.3	Assumptions	21
13.4.4	Access Routine Semantics	21
13.4.5	Local Functions	22
<b>14</b>	<b>MIS of Authentication Module</b>	<b>23</b>
14.1	Module	23
14.2	Uses	23
14.3	Syntax	23
14.3.1	Exported Constants	23
14.3.2	Exported Access Programs	23
14.4	Semantics	23
14.4.1	State Variables	23
14.4.2	Environment Variables	23
14.4.3	Assumptions	24
14.4.4	Access Routine Semantics	24
14.4.5	Local Functions	24

<b>15 MIS of Team Management Module</b>	<b>25</b>
15.1 Module . . . . .	25
15.2 Uses . . . . .	25
15.3 Syntax . . . . .	25
15.3.1 Exported Constants . . . . .	25
15.3.2 Exported Access Programs . . . . .	25
15.4 Semantics . . . . .	25
15.4.1 State Variables . . . . .	25
15.4.2 Environment Variables . . . . .	25
15.4.3 Assumptions . . . . .	26
15.4.4 Access Routine Semantics . . . . .	26
15.4.5 Local Functions . . . . .	27
<b>16 MIS of Announcements Module</b>	<b>27</b>
16.1 Module . . . . .	27
16.2 Uses . . . . .	27
16.3 Syntax . . . . .	27
16.3.1 Exported Constants . . . . .	27
16.3.2 Exported Access Programs . . . . .	27
16.4 Semantics . . . . .	27
16.4.1 State Variables . . . . .	27
16.4.2 Environment Variables . . . . .	28
16.4.3 Assumptions . . . . .	28
16.4.4 Access Routine Semantics . . . . .	28
16.4.5 Local Functions . . . . .	28
<b>17 MIS of Scheduling Algorithm Module</b>	<b>29</b>
17.1 Module . . . . .	29
17.2 Uses . . . . .	29
17.3 Syntax . . . . .	29
17.3.1 Exported Constants . . . . .	29
17.3.2 Exported Access Programs . . . . .	29
17.4 Semantics . . . . .	29
17.4.1 State Variables . . . . .	29
17.4.2 Environment Variables . . . . .	30
17.4.3 Assumptions . . . . .	30
17.4.4 Access Routine Semantics . . . . .	30
17.4.5 Local Functions . . . . .	30
<b>18 MIS of Game Management Module (M4)</b>	<b>31</b>
18.1 Module . . . . .	31
18.2 Uses . . . . .	31
18.3 Syntax . . . . .	31

18.3.1	Exported Constants	31
18.3.2	Exported Access Programs	31
18.4	Semantics	31
18.4.1	State Variables	31
18.4.2	Environment Variables	31
18.4.3	Assumptions	32
18.4.4	Access Routine Semantics	32
18.4.5	Local Functions	32
<b>19</b>	<b>MIS of Division Module</b>	<b>33</b>
19.1	Module	33
19.2	Uses	33
19.3	Syntax	33
19.3.1	Exported Constants	33
19.3.2	Exported Access Programs	33
19.4	Semantics	33
19.4.1	State Variables	33
19.4.2	Environment Variables	33
19.4.3	Assumptions	33
19.4.4	Access Routine Semantics	33
<b>20</b>	<b>MIS of GameslotT Module</b>	<b>35</b>
20.1	Module	35
20.2	Uses	35
20.3	Syntax	35
20.3.1	Exported Constants	35
20.3.2	Exported Access Programs	35
20.4	Semantics	35
20.4.1	State Variables	35
20.4.2	Environment Variables	35
20.4.3	Assumptions	35
20.4.4	Access Routine Semantics	35
<b>21</b>	<b>MIS of Reschedule Request Module</b>	<b>36</b>
21.1	Module	36
21.2	Uses	36
21.3	Syntax	36
21.3.1	Exported Constants	36
21.3.2	Exported Access Programs	36
21.4	Semantics	36
21.4.1	State Variables	36
21.4.2	Environment Variables	36
21.4.3	Assumptions	36



21.4.4	Access Routine Semantics . . . . .	36
<b>22</b>	<b>MIS of SeasonT Module</b>	<b>37</b>
22.1	Module . . . . .	37
22.2	Uses . . . . .	37
22.3	Syntax . . . . .	37
22.3.1	Exported Constants . . . . .	37
22.3.2	Exported Access Programs . . . . .	37
22.4	Semantics . . . . .	37
22.4.1	State Variables . . . . .	37
22.4.2	Environment Variables . . . . .	37
22.4.3	Assumptions . . . . .	37
22.4.4	Access Routine Semantics . . . . .	38
<b>23</b>	<b>MIS of StandingT Module</b>	<b>38</b>
23.1	Module . . . . .	38
23.2	Uses . . . . .	38
23.3	Syntax . . . . .	38
23.3.1	Exported Constants . . . . .	38
23.3.2	Exported Access Programs . . . . .	38
23.4	Semantics . . . . .	38
23.4.1	State Variables . . . . .	38
23.4.2	Environment Variables . . . . .	38
23.4.3	Assumptions . . . . .	38
23.4.4	Access Routine Semantics . . . . .	39
<b>24</b>	<b>Appendix</b>	<b>39</b>

### 3 Introduction

The following document details the Module Interface Specifications for Pitch Perfect McMaster GSA Softball League Platform, a web application for managing the McMaster GSA Softball League.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at <https://github.com/dccheung11/team-6-capstone-project>.

### 4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol  $:=$  is used for a multiple assignment statement and conditional rules follow the form  $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | \dots | c_n \Rightarrow r_n)$ .

The following table summarizes the primitive data types used by Software Engineering.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in $(-\infty, \infty)$
natural number	$\mathbb{N}$	a number without a fractional component in $[1, \infty)$
real	$\mathbb{R}$	any number in $(-\infty, \infty)$
multiple assignment	$:=$	multiple assignment statement
set	$\{ \dots \}$	collection of unique elements
not in set	$\backslash$	element not in set
union	$\cup$	union of two sets
intersection	$\cap$	intersection of two sets

The specification of Software Engineering uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Software Engineering uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

## 5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	Level 2
Hardware-Hiding Module	
	User Interface Module
	Authentication Module
	Team Management Module
Behaviour-Hiding Module	Game Management Module
	Announcements Module
	Scheduling Module
	Standings Module
	Waiver Module
	PlayerT Module
	TeamT Module
	GameT Module
Software Decision Module	Database Module
	Notification Module
	Scheduling Algorithm Module

Table 1: Module Hierarchy

## References

- Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.
- Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL <http://citeseer.ist.psu.edu/428727.html>.

## 6 MIS of User Interface Module (M1)

### 6.1 Module

User Interface Module

### 6.2 Uses

- Backend Module: To retrieve and send data for rendering views and processing user inputs.
- Authentication Module: For user login and role verification.
- Game Management Module: To display and update game information.
- Team Management Module: To display team information
- Scheduling Module: To display schedules
- Standings Module: To display league standings and updates.
- Announcements Module: To display Announcements
- Notification Module: To display interface for creating notifications

### 6.3 Syntax

#### 6.3.1 Exported Constants

- None

#### 6.3.2 Exported Access Programs

Name	In	Out	Exceptions
renderView	View	-	InvalidViewException

### 6.4 Semantics

#### 6.4.1 State Variables

- **currentView**: Stores the identifier for the currently displayed view.
- **userRole**: Stores the role of the currently logged-in user (e.g., player, captain, commissioner).

### 6.4.2 Environment Variables

- Browser environment: The module interacts with the user's browser, including DOM manipulation and event handling.
- Network connection: Required for fetching and sending data to the backend.

### 6.4.3 Assumptions

- Users will have a modern web browser that supports the required JavaScript features.
- A stable network connection is available during interactions requiring backend communication.

### 6.4.4 Access Routine Semantics

`renderView(view):`

- transition: `currentView := view`.
- exception: Throws `InvalidViewException` if `view` is unsupported.

### 6.4.5 Local Functions

- None

## 7 MIS of Scheduling Module

### 7.1 Module

**Scheduling Module:** Behaviour-Hiding Module

### 7.2 Uses

- Scheduling Algorithm Module
- TeamT Module
- GameT Module

### 7.3 Syntax

#### 7.3.1 Exported Constants

- **DEFAULT\_WEEK\_COUNT:** Integer  
Default number of weeks in the season.

### 7.3.2 Exported Access Programs

Name	In	Out	Exceptions
create schedule	TeamT[], Slot Object[], Integer	Schedule Object	-
addGame	GameT	Boolean	InvalidInput
removeGame	String	Boolean	GameNotFound
updateGameSlot	String, GameslotT	Boolean	GameNotFound
rescheduleGame	String, GameT, GameT	Boolean	GameNotFound, SlotNotFound

## 7.4 Semantics

### 7.4.1 State Variables

$$\text{schedule} \subseteq \mathcal{T} \times \mathcal{S} \times \mathcal{G}$$

where:

- $\mathcal{T}$ : Set of all teams.
- $\mathcal{S}$ : Set of available slots.
- $\mathcal{G}$ : Set of all game identifiers.

### 7.4.2 Environment Variables

None.

### 7.4.3 Assumptions

- $\forall t \in \mathcal{T}, \exists s \in \mathcal{S}, t$  is scheduled exactly once per week.
- Valid team ( $t \in \mathcal{T}$ ) and slot ( $s \in \mathcal{S}$ ) data are provided for schedule creation.

### 7.4.4 Access Routine Semantics

**createSchedule(teams, slots, seasonLength):**

transition:  $\text{schedule} := \text{generateSchedule}(\text{teams}, \text{slots}, \text{seasonLength})$

output:  $\text{out} := \text{schedule}$

constraint:  $\forall t \in \mathcal{T}, \exists s \in \mathcal{S}, t$  plays once per week.

exception: None.

**addGame(details):**

transition:  $\text{schedule} := \text{schedule} \cup \{(t_1, t_2, s) \mid t_1, t_2 \in \mathcal{T}, t_1 \neq t_2, s \in \mathcal{S}\}$

output:  $\text{out} := \text{true}$  if details are valid, false otherwise.

exception: `InvalidInput` if details are incomplete or invalid.

**removeGame(gameId):**

transition:  $\text{schedule} := \text{schedule} \setminus \{g \mid g \in \mathcal{G}, g.\text{id} = \text{gameId}\}$

output:  $\text{out} := \text{true}$  if gameId exists in schedule, false otherwise.

exception: `GameNotFound` if gameId does not exist.

**updateGameSlot(gameId, newSlot):**

transition:  $\text{schedule} := \text{schedule} \setminus \{g \mid g \in \mathcal{G}, g.\text{id} = \text{gameId}\} \cup \{(t_1, t_2, \text{newSlot})\}$

output:  $\text{out} := \text{true}$  if the update is successful.

exception: `GameNotFound` if gameId does not exist.

**rescheduleGame(gameId, newSlot, oldSlot):**

transition:  $\text{schedule} := \text{schedule} \setminus \{(t_1, t_2, \text{oldSlot}) \mid t_1, t_2 \in \mathcal{T}\} \cup \{(t_1, t_2, \text{newSlot})\}$

output:  $\text{out} := \text{true}$  if the reschedule is successful.

exceptions:  $\begin{cases} \text{GameNotFound} & \text{if gameId does not exist.} \\ \text{SlotNotFound} & \text{if oldSlot or newSlot does not exist.} \end{cases}$

**7.4.5 Local Functions**

None.



## 8 MIS of PlayerT Module

### 8.1 Module

**PlayerT**: Abstract Player Module.

### 8.2 Uses

TeamT

- **GameT**: The Player module interacts with the Game module to track player participation in games.
- **TeamT**: The Player module is connected to the Team module, as players are assigned to teams.

### 8.3 Syntax

#### 8.3.1 Exported Constants

#### 8.3.2 Exported Access Programs

Name	In	Out	Exceptions
PlayerT	String, String, String, String, Bool, String	-	-
getPlayerId		String	-
getName		String	-
getEmail		String	-
getWaiverStatus		Bool	-
getTeam		Bool	-
setWaiverStatus	Bool	-	-
setTeam	String	-	-

### 8.4 Semantics

#### 8.4.1 State Variables

#### 8.4.2 Environment Variables

// TODO: some UUID auth

### 8.4.3 Assumptions

### 8.4.4 Access Routine Semantics

PlayerT(id, n, e, p, w, t):

- transition:  $playerId, name, email, password, waiverStatus, team := id, n, e, p, w, t$
- output:  $out := self$
- exception: None

getPlayerId():

- output:  $out := playerId$
- exception: None

getName():

- output:  $out := name$
- exception: None

getEmail():

- output:  $out := email$
- exception: None

getWaiverStatus():

- output:  $out := waiverStatus$
- exception: None

getTeam():

- output:  $out := team$
- exception: None

setTeam(t):

- transition:  $team := t$
- exception: None

setWaiverStatus(w):

- transition:  $waiverStatus := w$
- exception: None

### 8.4.5 Local Functions

## 9 MIS of GameT Module

### 9.1 Module

**GameT**: Abstract Game Type

### 9.2 Uses

TeamT

### 9.3 Syntax

#### 9.3.1 Exported Constants

#### 9.3.2 Exported Access Programs

Name	In	Out	Exceptions
GameT	String, String, Date, String, String, Integer, Integer, String	-	-
getGameId		String	-
getTeamsInGame		TeamT[]	GameNotFound
getGameDetails		Object	GameNotFound
getStatus		String	GameNotFound
getField		String	GameNotFound
setStatus	String	-	InvalidStatus
setField	String	-	InvalidField
setScore	Integer, Integer	-	InvalidScore

### 9.4 Semantics

#### 9.4.1 State Variables

#### 9.4.2 Environment Variables

None

#### 9.4.3 Assumptions

- A game must have two teams assigned to it.
- A game must have a field assigned to it.

- A game's score can only be updated after the game is completed.
- The game status must be updated to reflect its current state (e.g., scheduled, completed).

#### 9.4.4 Access Routine Semantics

GameT(id, t1, t2, d, t, s1, s2, f):

- transition:  $gameId, team1Id, team2Id, gameDate, gameTime, scoreTeam1, scoreTeam2, field := id, t1, t2, d, t, s1, s2, f$
- output:  $out := self$
- exception: None

getGameId():

- output:  $out := gameId$
- exception: None

getGameDetails():

- output:  $out :=$  Object containing game details:  $gameId, team1Id, team2Id, gameDate, gameTime,$
- exception: Game not found if the game ID does not exist.

getTeamsInGame():

- output:  $out :=$  Array of Teams participating in the game
- exception: Game not found if the game ID does not exist.

getStatus():

- output:  $out := status$
- exception: Game not found if the game ID does not exist.

getField():

- output:  $out := field$
- exception: Game not found if the game ID does not exist.

setStatus(status):

- transition:  $status := status$
- exception: Invalid status if the provided status is not valid.

setField(field):

- transition:  $field := field$
- exception: Invalid field if the provided field is not valid.

setScore(score1, score2):

- transition:  $scoreTeam1 := score1, scoreTeam2 := score2$
- exception: Invalid score if the provided scores are not valid integers.

#### 9.4.5 Local Functions

- **validateGameDetails()**: A function to validate the input details when creating a new game.

## 10 MIS of TeamT Module

### 10.1 Module

**TeamT Module:** Abstract Team Module

### 10.2 Uses

PlayerT, GameT

### 10.3 Syntax

#### 10.3.1 Exported Constants

#### 10.3.2 Exported Access Programs

Name	In	Out	Exceptions
TeamT	String, String, String, PlayerT[], PlayerT	-	-
getTeamId		String	-
getTeamName		String	-
getDivision		String	-
getRoster		PlayerT[]	-
getCaptain		PlayerT	-
addPlayer	PlayerT	-	InvalidPlayer
removePlayer	PlayerT	-	PlayerNotFound

## 10.4 Semantics

### 10.4.1 State Variables

### 10.4.2 Environment Variables

None

### 10.4.3 Assumptions

- Each team must have a unique team ID.
- A team can belong to one division at a time.
- The team roster must be an array or list of players (with unique player identifiers).

### 10.4.4 Access Routine Semantics

TeamT(id, n, d, r, c):

- transition:  $teamId, teamName, division, roster, captain := id, n, d, r, c$
- output:  $out := self$
- exception: None

getTeamId():

- output:  $out := teamId$
- exception: None

getTeamName():

- output:  $out := teamName$
- exception: None

getDivision():

- output:  $out := division$
- exception: None

getRoster():

- output:  $out := roster$
- exception: None

getCaptain():

- output:  $out := captain$
- exception: None

addPlayer(player):

- transition:  $roster := roster + player$
- exception: Invalid player if the player is invalid or already in the roster.

removePlayer(player):

- transition:  $roster := roster - player$
- exception: Player not found if the player is not in the roster.

#### 10.4.5 Local Functions

None

## 11 MIS of Backend Module

### 11.1 Module

Backend/Database

### 11.2 Uses

PlayerT, GameT, TeamT

## 11.3 Syntax

### 11.3.1 Exported Constants

### 11.3.2 Exported Access Programs

Name	In	Out	Exceptions
createPlayer	String, String, String, String, Bool, String	-	PlayerCreationError
getPlayer	String	PlayerT	PlayerNotFound
updatePlayer	String, String, String, String, Bool, String	-	PlayerNotFound
deletePlayer	String	-	PlayerNotFound
createTeam	String, String, String, PlayerT[], PlayerT	-	TeamCreationError
getTeam	String	TeamT	TeamNotFound
updateTeam	String, String, String, PlayerT[], PlayerT	-	TeamCreationError
deleteTeam	String	-	TeamNotFound
createGame	String, String, Date, String, String, Integer, Integer, String	-	GameCreationError
getGame	String	GameT	GameNotFound
updateGame	String, String, Date, String, String, Integer, Integer, String	-	GameCreationError
deleteGame	String	-	GameNotFound
getAllPlayersForTeam	String	PlayerT[]	TeamNotFound
getAllGamesForTeam	String	GameT[]	TeamNotFound

## 11.4 Semantics

### 11.4.1 State Variables

None

### 11.4.2 Environment Variables

None



### 11.4.3 Assumptions

- The backend is connected to a database
- Each database operation (CRUD) will be encapsulated in a backend method to ensure separation of concerns
- Data consistency and integrity are maintained by the backend during each operation.

### 11.4.4 Access Routine Semantics

createPlayer(name, email, password, waiverStatus, team):

- transition:  $playerId, name, email, password, waiverStatus, team := name, email, password, waiverStatus, team$
- exception: "Player Creation Error" if player cannot be created

getPlayer(playerId):

- output:  $out := player$  (retrieves player object based on playerId)
- exception: "Player Not Found" if player does not exist

updatePlayer(playerId, name, email, password, waiverStatus, team):

- transition:  $name, email, password, waiverStatus, team := name, email, password, waiverStatus, team$  (updates player's details)
- exception: "Player Not Found" if player does not exist

deletePlayer(playerId):

- transition:  $player := null$  (deletes the player from the database)
- exception: "Player Not Found" if player does not exist

createTeam(teamName, division, captain, roster):

- transition:  $teamId, teamName, division, captain, roster := teamName, division, captain, roster$
- exception: "Team Creation Error" if team cannot be created

getTeam(teamId):

- output:  $out := team$  (retrieves team object based on teamId)
- exception: "Team Not Found" if team does not exist

updateTeam(teamId, teamName, division, roster):

- transition:  $teamName, division, roster := teamName, division, roster$  (updates team's details)
- exception: "Team Not Found" if team does not exist

deleteTeam(teamId):

- transition:  $team := null$  (deletes the team from the database)
- exception: "Team Not Found" if team does not exist

createGame(teams, date, time, field, score):

- transition:  $gameId, teams, date, time, field, score := teams, date, time, field, score$  (creates a new game)
- exception: "Game Creation Error" if game cannot be created (e.g., scheduling conflict)

getGame(gameId):

- output:  $out := game$  (retrieves game object based on gameId)
- exception: "Game Not Found" if game does not exist

updateGame(gameId, updates):

- transition:  $gameId, updates := gameId, updates$  (updates the game's details)
- exception: "Game Not Found" if game does not exist

deleteGame(gameId):

- transition:  $game := null$  (deletes the game from the database)
- exception: "Game Not Found" if game does not exist

getAllPlayersForTeam(teamId):

- output:  $out := players$  (retrieves all players associated with the team)
- exception: "Team Not Found" if team does not exist

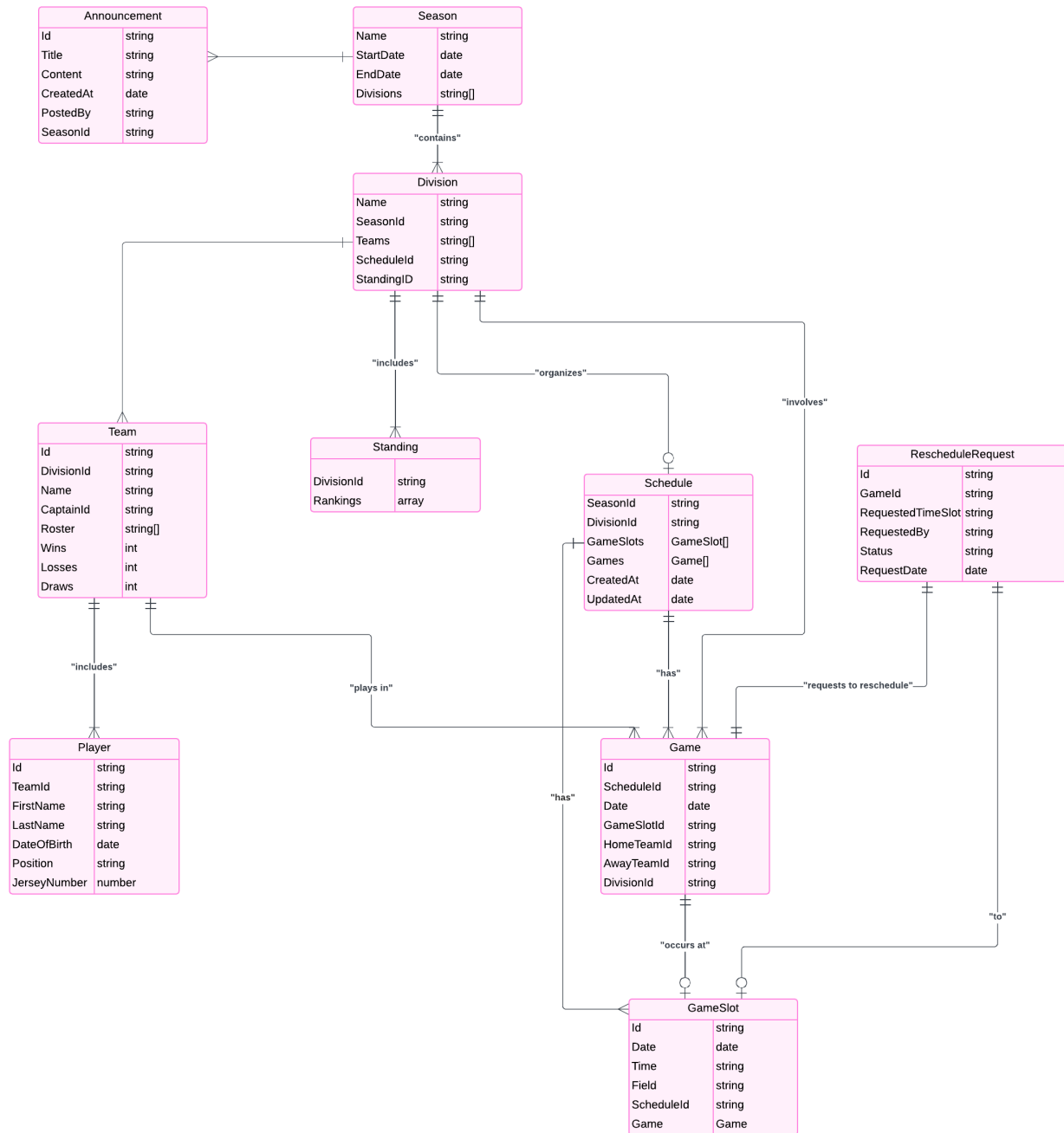
getAllGamesForTeam(teamId):

- output:  $out := games$  (retrieves all games associated with the team)
- exception: "Team Not Found" if team does not exist

#### 11.4.5 Local Functions

None

## 11.4.6 ERD for Database



## 12 MIS of Waiver Module

### 12.1 Module

Waiver

## 12.2 Uses

- None

## 12.3 Syntax

### 12.3.1 Exported Constants

- None

### 12.3.2 Exported Access Programs

Name	In	Out	Exceptions
<code>createWaiver</code>	Waiver details	Waiver ID	<code>InvalidInputException</code>
<code>getWaiver</code>	Waiver ID	Waiver details	<code>WaiverNotFoundException</code>
<code>signWaiver</code>	User ID, Waiver ID	Boolean	<code>WaiverNotFoundException</code> , <code>AlreadySignedException</code>
<code>listWaivers</code>	User ID	List of waivers	-

## 12.4 Semantics

### 12.4.1 State Variables

- `waiverList`: A collection of all waivers, including details and user signatures.

### 12.4.2 Environment Variables

- Database: For storage of waiver details and signatures.

### 12.4.3 Assumptions

- All users accessing the waiver module are authenticated.
- Waiver text complies with the legal requirements of the university softball league.

### 12.4.4 Access Routine Semantics

`createWaiver(details)`:

- transition: Adds a new waiver to `waiverList`.
- output: Returns a unique identifier for the created waiver.
- exception: Raises `InvalidInputException` if the waiver details are incomplete.

`getWaiver(waiverID):`

- output: Retrieves details of the specified waiver.
- exception: Raises `WaiverNotFoundException` if the waiver does not exist.

`signWaiver(playerID, waiverID):`

- transition: Updates `waiverList` to record the user's signature for the specified waiver.
- output: Returns `true` if the signature is successful.
- exception: Raises `WaiverNotFoundException` if the waiver does not exist or `AlreadySignedException` if the user has already signed the waiver.

`listWaivers(userID):`

- output: Returns all waivers associated with the user.

#### 12.4.5 Local Functions

- None

## 13 MIS of Notification Module

### 13.1 Module

Notification

### 13.2 Uses

- None

### 13.3 Syntax

#### 13.3.1 Exported Constants

- `MAX_NOTIFICATION_LENGTH`: Maximum allowed characters in a notification message.
- `NOTIFICATION_RETRY_LIMIT`: Maximum retry attempts for failed notifications.

### 13.3.2 Exported Access Programs

Name	In	Out	Exceptions
<code>send</code>	NotificationID, playerID	Boolean	InvalidPlayerID, SendFailure
<code>schedule</code>	NotificationID, DateTime	Boolean	InvalidDateTime
<code>status</code>	NotificationID	Status	InvalidNotificationID

## 13.4 Semantics

### 13.4.1 State Variables

- `pendingNotifications`: List of notifications yet to be delivered.
- `deliveredNotifications`: List of successfully delivered notifications.

### 13.4.2 Environment Variables

- Email Gateway: For sending email notifications.
- SMS Gateway: For sending SMS notifications.

### 13.4.3 Assumptions

- Users have valid email addresses or phone numbers stored in the database.
- Gateway APIs are operational and accessible.

### 13.4.4 Access Routine Semantics

`send(NotificationID, playerID)`:

- `transition`: Moves the notification from `pendingNotifications` to `deliveredNotifications` if successfully sent.
- `output`: `true` if the notification is successfully sent; `false` otherwise.
- `exception`: `InvalidPlayerID` if the `playerID` is not found; `SendFailure` if the notification fails to send after retries.

`schedule(NotificationID, DateTime)`:

- `transition`: Adds the notification to the `pendingNotifications` queue with the scheduled delivery time.

- output: `true` if the scheduling is successful; `false` otherwise.
- exception: `InvalidDateTime` if the `DateTime` is in the past or improperly formatted.

`status(NotificationID):`

- transition: `None`.
- output: The status of the notification (e.g., `Pending`, `Delivered`, `Failed`).
- exception: `InvalidNotificationID` if the `NotificationID` is not found.

#### 13.4.5 Local Functions

- `validateNotification(NotificationID):` Ensures the notification exists and is properly formatted.
- `retryFailedNotifications():` Attempts to resend notifications marked as failed.

## 14 MIS of Authentication Module

### 14.1 Module

**Auth** (M2) - Abstract object for handling user authentication and session management.

### 14.2 Uses

- **User Interface Module** (M1) - For collecting user credentials (username, password) and displaying authentication feedback.
- **Backend Module** (M13) - For verifying credentials, managing tokens, and storing authentication data.

### 14.3 Syntax

#### 14.3.1 Exported Constants

- **SESSION\_TIMEOUT** - Integer representing session timeout in minutes (default: 30).

#### 14.3.2 Exported Access Programs

Name	In	Out	Exceptions
login	String, String	Boolean	InvalidCredentials
logout	String	Void	SessionNotFound
registerUser	String, String, String	Boolean	DuplicateUserError
verifyToken	String	Boolean	TokenExpiredError
generateToken	String	String (Token)	UserNotFound

### 14.4 Semantics

#### 14.4.1 State Variables

- **userSessions**: Map of active session tokens to user IDs.
- **userData**: Map of user IDs to credentials and roles.

#### 14.4.2 Environment Variables

- **Database Connection**: Used for storing and retrieving user authentication data.
- **SSL/TLS Connection**: Required for secure communication between client and server.



### 14.4.3 Assumptions

- All passwords are stored as securely hashed values.
- Token expiration is managed based on `SESSION_TIMEOUT`.

### 14.4.4 Access Routine Semantics

- `login(username, password)`
  - **Transition:** If the username and password match, generate a session token and add it to `userSessions`.
  - **Output:** Returns `true` if successful, otherwise throws `InvalidCredentials`.
  - **Exception:** Throws `InvalidCredentials` if the username or password is incorrect.
- `logout(token)`
  - **Transition:** Removes the token from `userSessions`.
  - **Output:** None.
  - **Exception:** Throws `SessionNotFound` if the token is invalid or expired.
- `registerUser(username, password, role)`
  - **Transition:** Adds a new entry to `userData` with hashed password and role.
  - **Output:** Returns `true` if registration is successful.
  - **Exception:** Throws `DuplicateUserError` if the username already exists.
- `verifyToken(token)`
  - **Output:** Returns `true` if the token is valid, otherwise throws `TokenExpiredError`.
  - **Exception:** Throws `TokenExpiredError` if the token is expired.
- `generateToken(userID)`
  - **Output:** Generates a unique token linked to the `userID`.
  - **Exception:** Throws `UserNotFound` if the user ID does not exist.

### 14.4.5 Local Functions

- `hashPassword(password)`: Converts a plaintext password into a securely hashed value.
- `validatePassword(inputPassword, storedHash)`: Compares an input password to the stored hashed password.
- `generateUniqueToken(userID)`: Generates a cryptographically secure token linked to a user ID.

## 15 MIS of Team Management Module

### 15.1 Module

Team Management

### 15.2 Uses

- Waiver Module
- Notification Module

### 15.3 Syntax

#### 15.3.1 Exported Constants

- None

#### 15.3.2 Exported Access Programs

Name	In	Out	Exceptions
createTeam	Team details	Boolean	InvalidInput
getTeamDetails	Team ID	String	TeamNotFound
updateTeam	Team ID, Updates	Boolean	TeamNotFound
deleteTeam	Team ID	Boolean	TeamNotFound
listTeams	League ID	List	-
createPlayer	Team ID, Player ID	Boolean	TeamNotFound
deletePlayer	Team ID, Player ID	Boolean	TeamNotFound, PlayerNotFound

### 15.4 Semantics

#### 15.4.1 State Variables

- `teamList`: A collection of all teams, their details, and associated players.
- `leagueTeams`: A mapping between leagues and their associated teams.

#### 15.4.2 Environment Variables

- None

### 15.4.3 Assumptions

- All team operations are initiated by authorized users.
- Team details such as names and IDs are unique within a league

### 15.4.4 Access Routine Semantics

`createTeam(details):`

- transition: Adds a new team to `teamList`.
- output: Returns a unique identifier for the created team.
- exception: Raises `InvalidInput` if the team details are incomplete or violate constraints (e.g., duplicate team name).

`getTeamDetails(teamID):`

- output: Retrieves the details of the specified team.
- exception: Raises `TeamNotFound` if the team does not exist.

`updateTeam(teamID, updates):`

- transition: Updates the details of the specified team in `teamList`.
- output: Returns `true` if the update is successful.
- exception: Raises `TeamNotFound` if the team does not exist.

`deleteTeam(teamID):`

- transition: Removes the specified team from `teamList`.
- output: Returns `true` if the deletion is successful.
- exception: Raises `TeamNotFound` if the team does not exist.

`listTeams(leagueID):`

- output: Returns a list of all teams associated with the specified league.

`createPlayer(teamID, playerID):`

- transition: Adds a player to the specified team in `teamList`.
- output: Returns `true` if the player is successfully added.
- exception: Raises `TeamNotFound` if the team does not exist

`deletePlayer(teamID, playerID):`

- transition: Removes a player from the specified team in `teamList`.
- output: Returns `true` if the player is successfully removed.
- exception: Raises `TeamNotFound` if the team does not exist or `PlayerNotFound` if the player is not part of the team

#### 15.4.5 Local Functions

- None

## 16 MIS of Announcements Module

### 16.1 Module

Announcements

### 16.2 Uses

- Notification Module
- Backend Module

### 16.3 Syntax

#### 16.3.1 Exported Constants

- `ANNOUNCEMENT_TYPE_INFO`: Constant for an informational announcement type.

#### 16.3.2 Exported Access Programs

Name	In	Out	Exceptions
<code>sendAnnouncement</code>	Announcement details (text, type)	Announcement ID	InvalidInput
<code>updateAnnouncement</code>	Announcement ID	Boolean	AnnouncementNotFound, InvalidUpdate
<code>deleteAnnouncement</code>	Announcement ID	Boolean	AnnouncementNotFound

### 16.4 Semantics

#### 16.4.1 State Variables

- None

### 16.4.2 Environment Variables

- None

### 16.4.3 Assumptions

- All announcement operations (create, update, delete) are performed by authorized users only.
- Users will be notified of new or updated announcements based on their notification preferences.

### 16.4.4 Access Routine Semantics

`sendAnnouncement(announcementID):`

- transition: Sends a notification about the specified announcement to subscribed users using `send()`
- output: Returns `true` if the notification was sent successfully.
- exception: Raises `AnnouncementNotFoundException` if the announcement does not exist.

`updateAnnouncement(announcementID):`

- transition: Updates the details of the specified announcement
- output: Returns `true` if the update is successful.
- exception: Raises `AnnouncementNotFound` if the announcement does not exist

`deleteAnnouncement(announcementID):`

- transition: Removes the specified announcement
- output: Returns `true` if the deletion is successful.
- exception: Raises `AnnouncementNotFound` if the announcementID does not exist.

### 16.4.5 Local Functions

- None

## 17 MIS of Scheduling Algorithm Module

### 17.1 Module

SchedulingAlgorithm: Software Decision Module.

### 17.2 Uses

- Scheduling Module
- TeamT
- GameslotT
- GameT

### 17.3 Syntax

#### 17.3.1 Exported Constants

- DEFAULT\_WEEK\_COUNT: Integer  
Default number of weeks for scheduling.
- MAX\_GAMES\_PER\_TEAM: Integer  
Maximum number of games per team.
- MIN\_GAMES\_PER\_TEAM: Integer  
Minimum number of games per team.

#### 17.3.2 Exported Access Programs

Name	In	Out	Exceptions
generateSchedule	TeamT[], GameslotT[]	Schedule object	InvalidInput
resolveConflicts	Schedule Object	Schedule Object	ConflictResolutionFailure
optimizeSchedule	Schedule Object	Schedule Object	OptimizationFailure

### 17.4 Semantics

#### 17.4.1 State Variables

None

### 17.4.2 Environment Variables

None

### 17.4.3 Assumptions

- Inputs, such as the list of teams, slots, and week count, are valid and non-empty.
- Teams have all required properties, such as constraints and preferences.
- Slots are pre-validated and adhere to game capacity limits.
- Conflicts are identifiable based on provided rules (e.g., double bookings, unavailable fields).

### 17.4.4 Access Routine Semantics

`generateSchedule(teams, slots, weekCount):`

- output: `out := schedule`  
A schedule object generated based on team constraints, slot availability, and the given week count.
- exception: Raises `InvalidInput` if the input is incomplete or invalid.

`resolveConflicts(schedule):`

- transition: Resolves scheduling conflicts.
- output: `out := updated_schedule`.
- exception: Raises `ConflictResolutionFailure` if the conflicts cannot be resolved.

`optimizeSchedule(schedule):`

- transition: Refines the input schedule to improve adherence to the specified metrics.
- output: `out := optimized_schedule`.
- exception: Raises `OptimizationFailure` if optimization fails due to constraints or conflicts.

### 17.4.5 Local Functions

- `calculateFairness(schedule):` Computes a fairness metric for the schedule, ensuring balance across teams.
- `detectConflicts(schedule):` Identifies conflicts, such as double bookings or unavailable slots, in the schedule.
- `applyOptimization(schedule, metrics):` Applies optimization techniques to enhance the schedule.

## 18 MIS of Game Management Module (M4)

### 18.1 Module

Game Management Module

### 18.2 Uses

- GameT Module
- TeamT Module
- Scheduling Module
- Database Module

### 18.3 Syntax

#### 18.3.1 Exported Constants

None

#### 18.3.2 Exported Access Programs

Name	In	Out	Exceptions
reportScore	int, int, int	-	InvalidGameID, InvalidScore
updateGameStatus	int, string	-	InvalidGameID, InvalidStatus
getGameDetails	int	GameT	InvalidGameID

### 18.4 Semantics

#### 18.4.1 State Variables

- **games**: A collection of all GameT objects which contain game details including game ID, participating teams, scores, and statuses

#### 18.4.2 Environment Variables

- Database: For storing and retrieving game records.
- User Interface Module: For allowing users to report scores and update statuses.



### 18.4.3 Assumptions

- Game IDs are unique and valid.
- Scores are reported accurately and honestly
- Game statuses are reported or updated accurately by only captains or administrators

### 18.4.4 Access Routine Semantics

`reportScore(gameID, team1Score, team2Score):`

- transition: `game.score1, game.score2 := team1Score, team2Score`
- exception: Throws `InvalidGameIDException` if the game does not exist. Throws `InvalidScoreException` if the scores are invalid.

`updateGameStatus(gameID, newStatus):`

- transition: `game.status := newStatus`
- exception: Throws `InvalidGameIDException` if the game does not exist. Throws `InvalidStatusException` if the status is invalid.

`getGameResults(gameID):`

- output: `out := game` (retrieves `GameT` object based on `gameId`)
- exception: Throws `InvalidGameIDException` if the game does not exist.

### 18.4.5 Local Functions

None

## 19 MIS of Division Module

### 19.1 Module

Division Module

### 19.2 Uses

TeamT Module

### 19.3 Syntax

#### 19.3.1 Exported Constants

None

#### 19.3.2 Exported Access Programs

Name	In	Out	Exceptions
getTeams	-	TeamT[]	-

### 19.4 Semantics

#### 19.4.1 State Variables

teams (array of all teams in division)

#### 19.4.2 Environment Variables

None

#### 19.4.3 Assumptions

Teams will be evenly distributed among divisions

#### 19.4.4 Access Routine Semantics

addTeam(team):

- transition: teams := teams + team (append team to list of teams)
- exception: TeamNotFound

removeTeam(team):

- transition: teams := teams - team (remove team from list of teams)

- exception: TeamNotFound

getTeams():

- output: teams
- exception: EmptyDivision

## **20 MIS of GameslotT Module**

### **20.1 Module**

Gameslot Record Module

### **20.2 Uses**

- GameT
- Scheduling Module

### **20.3 Syntax**

#### **20.3.1 Exported Constants**

None

#### **20.3.2 Exported Access Programs**

None

### **20.4 Semantics**

#### **20.4.1 State Variables**

- Time
- Field
- Game

#### **20.4.2 Environment Variables**

None

#### **20.4.3 Assumptions**

Timeslots may or may not contain a game

#### **20.4.4 Access Routine Semantics**

None

## **21 MIS of Reschedule Request Module**

### **21.1 Module**

Reschedule Request Record Module

### **21.2 Uses**

- GameT
- Scheduling Module

### **21.3 Syntax**

#### **21.3.1 Exported Constants**

None

#### **21.3.2 Exported Access Programs**

None

### **21.4 Semantics**

#### **21.4.1 State Variables**

- Requests

#### **21.4.2 Environment Variables**

None

#### **21.4.3 Assumptions**

Requests are created and seen in a timely manner

#### **21.4.4 Access Routine Semantics**

None

## **22 MIS of SeasonT Module**

### **22.1 Module**

Season Record Module

### **22.2 Uses**

- TeamT
- DivisionT
- StandingT
- Scheduling Module

### **22.3 Syntax**

#### **22.3.1 Exported Constants**

None

#### **22.3.2 Exported Access Programs**

None

### **22.4 Semantics**

#### **22.4.1 State Variables**

- Season ID
- Start Date
- End Date
- Divisions

#### **22.4.2 Environment Variables**

None

#### **22.4.3 Assumptions**

None

#### **22.4.4 Access Routine Semantics**

None

## **23 MIS of StandingT Module**

### **23.1 Module**

Standing Record Module

### **23.2 Uses**

- TeamT
- DivisionT
- SeasonT

### **23.3 Syntax**

#### **23.3.1 Exported Constants**

None

#### **23.3.2 Exported Access Programs**

None

### **23.4 Semantics**

#### **23.4.1 State Variables**

- Rankings
- DivisionID

#### **23.4.2 Environment Variables**

None

#### **23.4.3 Assumptions**

Standings are updated based on the outcome of games.

#### **23.4.4 Access Routine Semantics**

None

## **24 Appendix**



## Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

We understood the objectives of this design document quite well since we are more passionate about topics related to code writing processes, and the creation of each module includes brainstorming functions, variables, etc. Our capstone idea of a sports platform is straightforward, so many of the requirements were intuitive and discussed already with our supervisor, providing a solid foundation for our design decisions throughout this document. Thanks to the time we were given to write this document, we collaborated better compared to the previous documents. We had more meetings together as a group and discussed more through text messages, which kept everyone on track on what they were supposed to do.

2. What pain points did you experience during this deliverable, and how did you resolve them?

Although we discussed and assigned work better, we struggled with overall cohesiveness as we worked on each module independently from one another. Hence, although one person worked on the announcements module which depends on the notifications module, they may have no idea what the notifications module contains. Since we did not schedule specific times to work on it, there was randomness in which each module was completed.

Our solution to resolve this problem was to have several iterations of the design documentation, in which everyone took time to review modules, especially if they relied on other modules. This way, the necessary updates can be made continuously in case of new changes done to modules. There were also many merge conflicts due to these random completion times, as we were all making changes to the same document. We just had one person fix all the merge conflicts, and reviewed the final product to check if all the details were there and complete.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

Most of our design decisions came directly from discussions with our supervisor (client), as this project was a pre-approved capstone. The rest of our design decisions came directly from our own user experiences with sports league platforms, as we were heavy users of platforms such as IMLeague for McMaster Intramurals. For example, the implementation for the game-scheduling algorithm came directly from our supervisor, as he wanted to keep the same algorithm that the league has always used.

Similarly, we were asked to keep the user interface simple, emphasizing the importance of functionality over complexity for less tech-savvy users. Some architecture decisions, such as the backend database and choice of development framework were entirely decided based on the technical expertise of our team, which we also agreed were aligned with the industry's best practices for robustness and scalability.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), if any, needed to be changed, and why?

Some sections of our requirements document needed to be tweaked to consider all the modules that we've created, as some of the modules were based on intuition and preference, such as wanting to use MongoDB for our database. Not all database programs function the same way, so these requirements will be a continuous process as we become more familiar with this particular database platform. We also talked with our supervisor since then to discuss new details that we've possibly missed in our requirements document.

The hazard analysis also needed some additional revisions, as our design process may incorporate vulnerabilities related to user authentication or data security issues. We ran into a bunch of exceptions to consider when making the functionality of each module, which could potentially be a vulnerability that we forgot to add to our hazard analysis document. These changes are made to ensure that we have consistency throughout our project documentation as we continue to develop narrower goals.

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO\_ProbSolutions)

Athletes love stats. Our current solution lacks advanced analytics and reporting features due to resource constraints, which would be the amount of time for us to figure that out. It also would not be feasible as officials would have to record data continuously on a sheet of paper, which is a flawed solution because of carelessness and lack of attention coming from our own experiences, such as intramural basketball. Also, it is an advanced feature that our client did not ask for as the main functionality is prioritized.

Given unlimited resources, machine learning algorithms to analyze player and game data for performance insights, such as determining an MVP would be interesting. Other

improvements could be made to improve our project's scalability to accommodate new teams, but again, that would be extremely unnecessary given the scope of the project.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO\_Explores)

When considering design solutions, the main discussion was between time versus control. We wanted to maintain autonomy while considering the time constraints of the project. Autonomy was a big deal because we want our client to be able to use the project independently long-term without needing to update libraries or troubleshoot issues with third-party providers. If one of our external libraries stopped updating its source, that could potentially lead to an issue that our client didn't ask for.

For example, instead of using normal CSS, we chose to use Material UI as it is a library that is continuously up to date and won't be disappearing any time soon. By using this react component library, we save time and efficiency by not needing to reinvent the wheel every time we want to style a component. We wanted to create the other design solutions from scratch, such as the game scheduling algorithm, as that would be the best possible solution to prevent raised concerns about the long-term sustainability of the project.

Other design solutions will be created with an open mind so that the platform can evolve according to specific league requirements. For example, we could choose to hard code the minimum or maximum number of players on the team, in case the client wants to easily change it in the future by replacing the value with another value. It is something that a new developer can easily understand and acknowledge, instead of needing to dive into reading documentation or in the worst case, the entire code base.