Module Interface Specification for Software Engineering

Team 6, Pitch Perfect
Damien Cheung
Jad Haytaoglu
Derek Li
Temituoyo Ugborogho
Emma Wigglesworth

January 14, 2025

1 Revision History

Date	Version	Notes
Jan 3	1.0	Added MIS for TeamT, GameT, PlayerT, Backend
Jan 14	1.x	Added MIS for Season and Standing Record Modules

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at [give url —SS]

[Also add any additional symbols, abbreviations or acronyms —SS]

Contents

3 Introduction

The following document details the Module Interface Specifications for [Fill in your project name and description —SS]

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at .... [provide the url for your repo —SS]

4 Notation

[You should describe your notation. You can use what is below as a starting point. —SS]

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 $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

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 $(-\infty, \infty)$

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	
Behaviour-Hiding	Input Parameters Output Format Output Verification Temperature ODEs Energy Equations Control Module Specification Parameters Module
Software Decision	Sequence Data Structure ODE Solver Plotting

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

- **schedule: Object**
  The current schedule object containing all games, gameslots, and team assignments.

7.4.2 Environment Variables

- None

7.4.3 Assumptions

- Valid team and gameslot data are provided for schedule creation.

7.4.4 Access Routine Semantics

createSchedule(teams, slots, seasonLength):

- transition: $schedule := generateSchedule(teams, slots, seasonLength)$
- output: $out := schedule$
- exception: None

addGame(details):

- transition: Adds a new game to the schedule using the provided details.
- output: $out := true$ if the game is successfully added.
- exception: Raises InvalidInput if details are incomplete or invalid.

removeGame(gameId):

- transition: Removes the game with the given gameId from the schedule.

- output: $out := true$ if the game is successfully removed.
- exception: Raises GameNotFound if the game does not exist.

updateGameSlot(gameId, newSlot):

- transition: Updates the slot for the game with gameId in the schedule to newSlot. Removes the game from the old slot.
- output: $out := true$ if the update is successful.
- exception: Raises GameNotFound if the game does not exist.

rescheduleGame(gameId, newSlot, oldSlot):

- transition: Removes the game with gameId from oldSlot and assigns it to newSlot in the schedule.
- output: $out := true$ if the reschedule is successful.
- exception: Raises GameNotFound if the game does not exist or SlotNotFound if either slot does not exist.

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

[Not all modules will have state variables. State variables give the module a memory. —SS]

11.4.2 Environment Variables

[This section is not necessary for all modules. Its purpose is to capture when the module has external interaction with the environment, such as for a device driver, screen interface, keyboard, file, etc. —SS]

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: $playerId, name, email, password, waiverStatus, team := name, email, password, waiver$
- 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: $name, email, password, waiverStatus, team := name, email, password, waiverStatus, te$ (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

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**
`createWaiver`	Waiver details	Waiver ID	InvalidInputException
`getWaiver`	Waiver ID	Waiver details	WaiverNotFoundException
`signWaiver`	User ID, Waiver ID	Boolean	WaiverNotFoundException, AlreadySignedException
`listWaivers`	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 `AlreadySignedExceptio` 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
`send`	NotificationID, playerID	Boolean	InvalidPlayerID, SendFailure
`schedule`	NotificationID, DateTime	Boolean	InvalidDateTime
`status`	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**
`sendAnnouncement`	Announcement details (text, type)	Announcement ID	InvalidInput
`updateAnnouncement`	Announcement ID	Boolean	AnnouncementNotFound, InvalidUpdate
`deleteAnnouncement`	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 stasuses

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

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

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?
2. What pain points did you experience during this deliverable, and how did you resolve them?
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?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?
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)
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)