Module Interface Specification for Farming Matters

Team #14, The Farmers
Brandon Duong
Andrew Balmakund
Mihail Serafimovski
Mohammad Harun
Namit Chopra

April 5, 2023

1 Revision History

Date	Version	Notes
18/1/2023	1.0	Finished First Version
1/4/2023	2.0	Finished Second Version
5/4/2023	2.0	Addressed GitHub issues

2 Symbols, Abbreviations and Acronyms

See SRS Documentation on github.

Contents

3 Introduction

The following document outlines the system design of Farming Matters. The project aims to conduct survey research through an interactive and engaging activity. This will further help understand genuine decisions from the users to help with the research of understanding risk-making decisions.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found on github.

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

The following table summarizes the primitive data types used by Farming Matters.

Data Type	Notation	Description
character	char	a single symbol or digit
string	String	a sequence of characters
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
boolean	$\mathbb{B}$	any true or false value

The specification of Farming Matters 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, Farming Matters 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	-
Behaviour-Hiding Module	GameController AvatarMenu – **done** Shop – **done** Inventory – **done** FarmGrid GameSettings – **done** CreateAccount Login SeasonalEvents GenerateStatistic
Software-Hiding Module	Avatar Consultant OtherAvatars FarmTile Seed Item – **done** DatabaseOperations – **done** ServerFirebase ClientFirebase AuthState Socket RedisClient Server MusicPlayer – **done** User AuthError

Table 1: Module Hierarchy

6 MIS of GenerateStatistic

6.1 Module

Generate Statistic

6.2 Uses

GameController

6.3 Syntax

6.3.1 Exported Constants

None

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
generateConsultantStatement	$\mathbb{Z}$, GameController	String	
generateGeneralStatement		String	
getEventHappening		$\mathbb{B}$	
getEventType		String	

6.4 Semantics

6.4.1 State Variables

pEventHappening : $\mathbb{Z}$
isEventHappening : $\mathbb{B}$
typeOfEvent : String

6.4.2 Environment Variables

None

6.4.3 Assumptions

None

6.4.4 Access Routine Semantics

generateConsultantStatement(decisionType, game):

- output: *out* := (decisionType = PROBABLISTIC_ID $\wedge$ (P_SEASONAL_LOW $\leq$ $randomProbability()$ $\leq$ P_SEASONAL_HIGH) $\Rightarrow$ $generateSeasonalEventStatements(generateProbablisticValue(), \text{game})$ | decisionType = PROBABLISTIC_ID $\wedge$ (P_MARKET_LOW $\leq$ $randomProbability()$ $\leq$ P_MARKET_HIGH) $\Rightarrow$ $generateMarketStatements(generateDeterministicValue(), \text{game})$ | decisionType = DETERMINISTIC_ID $\wedge$ (P_SEASONAL_LOW $\leq$ $randomProbability()$ $\leq$ P_SEASONAL_HIGH) $\Rightarrow$ $generateSeasonalEventStatements(generateProbablisticValue(), \text{game})$ | decisionType = DETERMINISTIC_ID $\wedge$ (P_MARKET_LOW $\leq$ $randomProbability()$ $\leq$ P_MARKET_HIGH) $\Rightarrow$ $generateMarketStatements(generateDeterministicValue(), \text{game})$)
- exception: none

generateGeneralStatement():

- output: *out* := generateOtherStatement()
- exception: none

getEventHappening():

- output: *out* := pEventHappening
- exception: none

getEventType():

- output: *out* := typeOfEvent
- exception: none

6.4.5 Local Functions

randomProbability: $\mathbb{R}$
This function produces a random real number between 0 and 1

generateSeasonalEventStatements: $String \times GameController \rightarrow$ seq of String
generateSeasonalEventStatements(generatedDecisionTypeValue, game) $\equiv$ Based on the current season, generate a seasonal event statement related to the next season. Choose a random event that can occur in the next season along with a probability stating of it occurring or not.

generateMarketStatements: $String \times GameController \rightarrow$ seq of String
generateMarketStatements(generatedDecisionTypeValue, game) $\equiv$ Based on the current season, generate a market statement related to the next season. Choose a random item along

with a probability stating whether the price will either increase or decrease and another probability of it occurring or not.

generateConsultantStatement: $\mathbb{Z} \times$ GameController $\rightarrow$ String
generateConsultantStatement(decisionType, game) $\equiv$ (decisionType = PROBABLISTIC_ID $\Rightarrow$ $relatedStatements(generateProbablisticValue(), \text{game})|$
decisionType = DETERMINISTIC_ID $\Rightarrow$ $relatedStatements(generateDeterministicValue(), \text{game})$)

realToString: $\mathbb{R} \rightarrow$ String
realToString(realValue) $\equiv$ Given a real value, convert this to a string.

booleanToString: $\mathbb{B} \rightarrow$ String
booleanToString(booleanValue) $\equiv$ Given a boolean value, convert this to a string.

generateProbablisticValue: void $\rightarrow$ String
generateProbablisticValue() $\equiv$ $realToString(\text{i} : \mathbb{R} | 0 \leq \text{i} \leq 1)$
Generating a random real number between 0 and 1.

generateDeterministicValue: void $\rightarrow$ String
generateDeterministicValue() $\equiv$ $booleanToString(\text{i} : \mathbb{B} | \text{i} \equiv true \vee \text{i} \equiv false)$
Generating a random boolean (true and false) value.

randomIndex: $\mathbb{Z} \rightarrow \mathbb{R}$
randomIndex(arrayLength) $\equiv i : \mathbb{R} | 0 \leq i \leq |arrayLength| - 1$
This function produces a random number between 0 and the length of the given array - 1.

generateOtherStatement: String
generateOtherStatement() $\equiv$
DEFAULT_STATEMENTS[$randomIndex$(DEFAULT_STATEMENTS)]
This function will randomly select a default statement from a seq of default statements that are associated with General Avatar Statements.

6.4.6 Local Constants

SEASONS: seq of String
SEASONS = [“Fall”, “Winter”, “Spring”, “Summer”]

PROBABILISTIC_ID: $\mathbb{Z}$
PROBABILISTIC_ID = 0

DETERMINISTIC_ID: $\mathbb{Z}$
DETERMINISTIC_ID = 1

P_SEASONAL_LOW: $\mathbb{R}$
P_SEASONAL_LOW = 0

P_SEASONAL_HIGH: $\mathbb{R}$
P_SEASONAL_HIGH = 0.2

P_MARKET_LOW: $\mathbb{R}$
P_SEASONAL_LOW = 0.2

P_MARKET_HIGH: $\mathbb{R}$
P_SEASONAL_HIGH = 1

OTHER_AVATAR_STATEMENTS: seq of String

7 MIS of Avatar

7.1 Module

Avatar

7.2 Uses

AvatarMenu

7.3 Syntax

7.3.1 Exported Constants

None

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
newAvatar	$\mathbb{Z}$,$\mathbb{Z}$, String, String, String, String	Avatar	
getID		$\mathbb{Z}$	
getType		$\mathbb{Z}$	
getName		String	
getRole		String	
getDescription		String	
getStatement		String	
setStatement	String		
renderDialog		String	

7.4 Semantics

7.4.1 State Variables

id : $\mathbb{Z}$
type : $\mathbb{Z}$
name : String
role : String
description : String
statement : String

7.4.2 Environment Variables

None

7.4.3 Assumptions

None

7.4.4 Access Routine Semantics

new Avatar(avatarID, avatarType, avatarName, avatarRole, avatarDescription, avatarStatement):

- transition: id, type, name, role, description, statement := avatarID, avatarType, avatarName, avatarRole, avatarDescription, avatarStatement
- output: *out* := self
- exception: none

getId():

- output: *out* := id
- exception: none

getType():

- output: *out* := type
- exception: none

getName():

- output: *out* := name
- exception: none

getRole():

- output: *out* := role
- exception: none

getDescription():

- output: *out* := description
- exception: none

getStatement():

- output: *out* := statement
- exception: none

setStatement(newStatement):

- transition: statement := newStatement
- exception: none

renderDialog():

- output: *out* := formats the styling of presenting a dialog to display all information about an avatar such as presenting the avatar's name, role, description and dialog statement.
- exception: none

8 MIS of AvatarMenu

8.1 Module

AvatarMenu

8.2 Uses

GameController, Avatar

8.3 Syntax

8.3.1 Exported Constants

None

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
newAvatarMenu	$\mathbb{Z}$	AvatarMenu	
getSelectedAvatar		$\mathbb{Z}$	
getDecisionType		$\mathbb{Z}$	
getIsOpened		$\mathbb{B}$	
onAvatarSelect	$\mathbb{Z}$		
onExitClick			
renderAvatarMenu		String	
generateAvatars			

8.4 Semantics

8.4.1 State Variables

selectedAvatar : $\mathbb{Z}$
decisionType : $\mathbb{Z}$
isOpened : $\mathbb{B}$
avatars : seq of Avatar

8.4.2 Environment Variables

None

8.4.3 Assumptions

None

8.4.4 Access Routine Semantics

new AvatarMenu(userDecisionType):

- transition: selectedAvatar, decisionType, isOpened := DEFAULT_AVATAR, userDecisionType, DEFAULT_OPEN_STATE
- output: *out* := self
- exception: none

getSelectedAvatar():

- out: *out* := selectedAvatar
- exception: none

getDecisionType():

- out: *out* := decisionType
- exception: none

getIsOpened():

- out: *out* := isOpened
- exception: none

onAvatarSelect(userSelectedAvatar):

- transition: isOpened, selectedAvatar := $true$, (a : $Avatar$|a $\in$ avatars : ($a.getId()$ = userSelectedAvatar))
- exception: none

onExitClick():

- transition: isOpened, selectedAvatar := DEFAULT_OPEN_STATE, DEFAULT_AVATAR
- exception: none

Upon requesting to exit out of the Avatar Menu, reset isOpened and selectedAvatar to their default state.

generateAvatars():

- transition: avatars := (index : $\mathbb{Z}$|0 $\leq$ index $\leq$ |AVATARS_ID| $-$ 1 : avatars||{$Avatar$(AVATARS_ID[i], AVATARS_TYPE[i], AVATARS_NAME[i], AVATARS_ROLE[i], AVATARS_DESCRIPTION[i], EMPTY_STRING)})

- exception: none

Generate a sequence of Avatar and concatenate to the state variable *avatars*.

renderAvatarMenu():

- out: formats the styling of presenting a menu with a list of in-game avatars to interact with.
- exception: none

8.4.5 Local Constants

EMPTY_STRING: String
DEFAULT_AVATAR: $\mathbb{Z}$
DEFAULT_OPEN_STATE: $\mathbb{B}$
AVATARS_ID: seq of $\mathbb{Z}$
AVATARS_TYPE: seq of $\mathbb{Z}$
AVATARS_NAME: seq of String
AVATARS_ROLE: seq of String
AVATARS_DESCRIPTION: seq of String

9 MIS of Consultant

9.1 Module

Consultant

9.2 Uses

GameController, GenerateStatistic

9.3 Syntax

9.3.1 Exported Constants

None

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
newConsultant	$\mathbb{Z}$	Consultant	
purchaseConsultant		$\mathbb{B}$	

9.4 Semantics

9.4.1 State Variables

decisionType : $\mathbb{Z}$
statement : string

9.4.2 Environment Variables

None

9.4.3 Assumptions

None

9.4.4 Access Routine Semantics

new Consultant(userDecisionType):

- transition: decisionType, statement := userDecisionType
- output: *out* := self
- exception: none

purchaseConsultant():

- transition: statement := ($canPurchaseConsultant$(game.money) $\Rightarrow$ $GenerateStatistics.genereateConsultantStatement$(decisionType, game))
- output: *out* := self
- exception: none

Upon requesting to exit out of the Avatar Menu, reset *isOpened* and *selectedAvatar* to their default state

9.4.5 Local Functions

randomIndex: $\mathbb{Z} \rightarrow \mathbb{R}$
randomIndex(arrayLength) $\equiv$ i : $\mathbb{R}|0 \leq$ i $\leq |$arrayLength$| - 1$
This function produces a random number between 0 and the length of the given array - 1.

canPurchaseConsultant: $\mathbb{R} \rightarrow \mathbb{B}$
canPurchaseConsultant(currentMoney) $\equiv$ (currentMoney $<$ DEFAULT_PURCHASE_PRICE $\Rightarrow$ $false|$currentMoney $>=$ DEFAULT_PURCHASE_PRICE $\Rightarrow$ $true$)
This function checks to see if the consultant's advice can be purchased given the money they have.

9.4.6 Local Constants

DEFAULT_PURCHASE_PRICE: $\mathbb{R}$
DEFAULT_PURCHASE_PRICE = 150.00

10 MIS of OtherAvatar

10.1 Module

OtherAvatar

10.2 Uses

Avatar

10.3 Syntax

10.3.1 Exported Constants

None

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
newOtherAvatar		OtherAvatar	

10.4 Semantics

10.4.1 State Variables

statement : String

10.4.2 Environment Variables

None

10.4.3 Assumptions

None

10.4.4 Access Routine Semantics

new OtherAvatar():

- transition: statement := *GenereateStatistcs.genereateGeneralStatement*()
- output: *out* := self
- exception: none

11 MIS of SeasonalEvents

11.1 Module

Seasonal Events

11.2 Uses

GameController, FarmGrid, FarmTile, Inventory

11.3 Syntax

11.3.1 Exported Constants

None

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
renderSeasonTransition	String	String	

11.4 Semantics

11.4.1 State Variables

None

11.4.2 Environment Variables

None

11.4.3 Assumptions

None

11.4.4 Access Routine Semantics

renderSeasonTransition(season):

- transition: game := all crops that are currently planted on the FarmGrid are removed and any crop that is insured is added to the Inventory
- output: *out* := formats the styling of displaying a seasonal event that occurs. This simulates the transition between seasons changing. There is a prompt description that displays an event statement that corresponds to that specific event in a given season. There is also a delay of SEASON_TRANSITION_DELAY seconds the season

transition prompt cannot be exited until the SEASON_TRANSITION_DELAY time has been completed.

- exception: none

11.4.5 Local Constants

SEASON_TRANSITION_DELAY: $\mathbb{Z}$
SEASON_TRANSITION_DELAY = 5

12 MIS of Item

12.1 Module

Item

12.2 Uses

None

12.3 Syntax

12.3.1 Exported Constants

None

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
newItem	String,$\mathbb{R}$,String	Item	
getName		String	
getFloorPrice		$\mathbb{R}$	
getType		String	

12.4 Semantics

12.4.1 State Variables

name : String
floorPrice : $\mathbb{R}$
type : String

12.4.2 Environment Variables

None

12.4.3 Assumptions

None

12.4.4 Access Routine Semantics

new Item(itemName, itemFloorPrice, itemType):

- transition: name, floorPrice, type := itemName, itemFloorPrice, itemType,

- output: *out* := self
- exception: none

getName():

- output: *out* := name
- exception: none

getFloorPrice():

- output: *out* := price
- exception: none

getType():

- output: *out* := type
- exception: none

13 MIS of Inventory

13.1 Module

Inventory

13.2 Uses

Item

13.3 Syntax

13.3.1 Exported Constants

None

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
newInventory		Inventory	
getItems		set of Item	
addItem	Item		
removeItem	Item		Illegal Argument Exception

13.4 Semantics

13.4.1 State Variables

inventory : Seq of Item

13.4.2 Environment Variables

None

13.4.3 Assumptions

An inventory will be created only at the beginning of the game or if the user wishes to delete their data and restart.

13.4.4 Access Routine Semantics

new Inventory():

- transition: inventory := []
- output: *out* := self
- exception: none

This constructor will create an empty sequence as the user will start with no items when the game begins.
getItems():

- output: *out* := {i : Item|i ∈ inventory : i}
- exception: none

addItem(item):

- transition: *inventory* := inventory ∪ {item}
- exception: none

removeItem(item):

- transition: *inventory* := inventory - {item}
- exception: *exc* := (item ∉ inventory) ⇒ IllegalArgumentException

14 MIS of Shop

14.1 Module

Shop

14.2 Uses

Items, Inventory

14.3 Syntax

14.3.1 Exported Constants

None

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
buyItems	Item,$\mathbb{Z}$,$\mathbb{R}$		IllegalArgumentException
sellItems	Item,$\mathbb{Z}$,$\mathbb{R}$		

14.4 Semantics

14.4.1 State Variables

inventory : set of Item
shopItems : set of Item
cropItems : set of Item
balance : $\mathbb{R}$

14.4.2 Environment Variables

None

14.4.3 Assumptions

None

14.4.4 Access Routine Semantics

buyItems(item,amount,floorPrice):

- transition:
  balance := (balance − amount × (item.price + floorPrice)) *where* (item ∈ shopItems)
  inventory := (∀x : $\mathbb{Z}|0 \leq x \leq$ amount : inventory.*addItem*(item))

- exception: *exc* := (balance < amount×(item.price + floorPrice)) ⇒ IllegalArgumentException
  Purchasing insurance on an item will mean the farmer is guaranteed to sell the item at a certain price(floorPrice). Since crop prices fluctuate, the price of the item they wish to sell may be lower than the buy price. This is meant to ensure the buyer does not lose any money when selling.

sellItems(item,amount,floorPrice) :

- transition:
  balance := (balance + amount × item.price) *where* (item ∈ cropItems) ∧ (item ∈ inventory) ∧ (item.floorPrice = floorPrice)
  inventory := (∀x : $\mathbb{Z}|0 \leq x \leq$ amount : inventory.*removeItem*(item))

- exception: *exc* := None

15 MIS of DatabaseOperations

15.1 Module

DatabaseOperations

15.2 Uses

AuthState, GameController

15.3 Syntax

15.3.1 Exported Constants

None

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
createConnection		$\mathbb{B}$	
stopConnection		$\mathbb{B}$	
createUserTable	$\mathbb{Z}$		
deleteUserTable	$\mathbb{Z}$		
logData	$\mathbb{Z}$, set of ActionDetails		IllegalArgument Exception
saveGame	$\mathbb{Z}$, set of MapGameStates		IllegalArgument Exception
loadGame	$\mathbb{Z}$	set of MapGameStates	IllegalArguement Exception

15.4 Semantics

15.4.1 State Variables

isConnected: $\mathbb{B}$

15.4.2 Environment Variables

None

15.4.3 Assumptions

The database server is running allowing for connection with valid incoming requests.

15.4.4 Access Routine Semantics

createConnection():

- output: *out* := If the user is logged in and not connected to the database server, send a request to using CREDENTIALS for access. Return True if the request is successful, False otherwise. If the user is not logged in or there is already a connection to the database, return False.
- transition: isConnected := True if request successful or there is already a connection, False otherwise
- exception: none

stopConnection():

- output: *out* := If the user is logged in and there is a connection to the database server, send a request to stop the connection. If the request is successful, return True and False otherwise. Return False, if the user is not logged in or there is no connection to the database.
- transition: isConnected := False if request successful or there is already not a connection, True otherwise
- exception: none

createUserTable(userId):

- output: *out* := If the user is logged in, the database is connected, and there is not a table corresponding to the userId, create a table corresponding using the userId.
- exception: None

deleteUserTable(userId):

- output: *out* := If the user is logged in, the database is connected, and there is a table corresponding to the userId, delete the table corresponding using the userId.
- exception: None

logData(userId, action):

- transition: If the user is logged in and the database is connected, add an entry in the database table corresponding to the userId.
- exception: If there is not a table corresponding to the userId, raise IllegalArguementException.

saveGame(userId, gameStateData):

- transition: If the user is logged in and the database is connected, modify the entry in the game state table corresponding to the userId with the gameStateData.
- exception: If there is not a table corresponding to the userId, raise IllegalArguementException.

loadGame(userId):

- output: *out* := If the user is logged in and the database is connected, return the entry in the game state table corresponding to the userId.
- exception: If there is not a table corresponding to the userId, raise IllegalArguementException.

15.4.5 Local Constants

CREDENTIALS: set of MapAuthInfo #mapping authentication secret keys and values

16 MIS of MusicPlayer

16.1 Module

MusicPlayer

16.2 Uses

None

16.3 Syntax

16.3.1 Exported Constants

None

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
loadMusic	String		InvalidFile
startMusic	$\mathbb{R}$		IllegalArguement
changeVolume	$\mathbb{R}$		IllegalArguement

16.4 Semantics

16.4.1 State Variables

audioFile: Audio
isPlaying: $\mathbb{B}$
volume: $\mathbb{R}$

16.4.2 Environment Variables

None

16.4.3 Assumptions

None

16.4.4 Access Routine Semantics

loadMusic(audioFileName):

- transition: audioFile := audioFileName

- exception: If the given audio file name does not exist in the directory or the provided file name is of invalid type, raise invalidFile exception.

startMusic(volume):

- transition: audioFile, volume, isPlaying := If isPlaying is False, start playing the audio file in an infinite loop, otherwise continue playing the music, volume, True

- exception: (volume $< 0 \Rightarrow$ IllegalArguement)

changeVolume(volume):

- transition: volume := volume
- exception: (volume $< 0 \Rightarrow$ IllegalArguement)

16.4.5 Local Functions

None

17 MIS of GameSettings

17.1 Module

GameSettings

17.2 Uses

DatabaseOperations, MusicPlayer

17.3 Syntax

17.3.1 Exported Constants

None

17.3.2 Exported Access Programs

Name	In	Out	Exceptions
setVolumeBackgroun-dMusic	$\mathbb{R}$		IllegalArguement Exception
setVolumeSoundEffects	$\mathbb{R}$		IllegalArguement Exception
withdrawFromStudy	String		
withdrawFromGame	String		

17.4 Semantics

17.4.1 State Variables

backgroundMusicVolume: $\mathbb{R}$ soundEffectsVolume: $\mathbb{R}$

17.4.2 Environment Variables

None

17.4.3 Assumptions

None

17.4.4 Access Routine Semantics

setVolumeBackgroundMusic(volume):

- transition: backgroundMusicVolume := volume

- exception: (volume $< 0 \Rightarrow$ IllegalArguementException)

setVolumeSoundEffects(volume):

- transition: currentVolume := newVolume
- exception: (volume $< 0 \Rightarrow$ IllegalArguementException)

withdrawFromStudy(userId):

- output: *out* := Deletes the user table using DatabaseOperations.deleteUserTable and also removes the entry corresponding to the userId in the game state table. The user information is also removed from Firebase.
- exception: None

withdrawFromGame(userId):

- output: *out* := The user information is removed from Firebase.
- exception: None

18 MIS of AuthState

18.1 Module

AuthState

18.2 Uses

User, AuthError, Socket, ClientFirebase, DatabaseOperations

18.3 Syntax

18.3.1 Exported Constants

None

18.3.2 Exported Access Programs

Name	In	Out	Exceptions
getUser		{User} ∪ {null}	
getUserId		{String} ∪ {null}	
getIsLoggedIn		$\mathbb{B}$	
getIsActiveSession		$\mathbb{B}$	
getIsDenied		$\mathbb{B}$	
getAuthError		AuthError	
signIn	String, String		
signOut			
createAccount	String, String, String		

Note: There are no exceptions present because instead of throwing exceptions, auth errors in all access programs are handled internally using the authError state variable.

18.4 Semantics

18.4.1 State Variables

user: User
isLoggedIn: $\mathbb{B}$
isActiveSession: $\mathbb{B}$
isDenied: $\mathbb{B}$
authError: AuthError
socket: Socket

18.4.2 Environment Variables

None

18.4.3 Assumptions

This module is the 'source of truth' for anything auth-related. This is an assumption guaranteed by this module.

18.4.4 Access Routine Semantics

getUser():

- output: *out* := ((isLoggedIn $\wedge$ $\neg$isDenied) $\Rightarrow$ user) | (True $\Rightarrow$ null)

getUserId():

- output: *out* := ((isLoggedIn $\wedge$ $\neg$isDenied) $\Rightarrow$ user.uid) | (True $\Rightarrow$ null)

getIsLoggedIn():

- output: *out* := isLoggedIn

getIsActiveSession():

- output: *out* := isActiveSession

getIsDenied():

- output: *out* := isDenied

getAuthError():

- output: *out* := authError

signIn(email, password):

- transition:
  Note that userCredential is a temporary variable in this method used to store the results of calling the Firebase API. It is not a state variable.

  userCredential := $ClientFirebase.signInWithEmailAndPassword$(email, password)

  user := (userCrediential.user $\neq$ null $\Rightarrow$ userCredential.user) | (userCrediential.user = null $\Rightarrow$ null)

  isLoggedIn := (userCrediential.user $\neq$ null)

isActiveSession :=
(userCrediential.user ≠ null)
⇒ socket.$checkIsOnlySession$(userCredential.user.uid, userCrediential.user.accessToken)
| (userCrediential.user = null) ⇒ false

isDenied := (userCrediential.error ≠ null)

authError := isDenied ⇒ userCredential.error | ¬isDenied ⇒ null

signOut():

- transition:
  user := null

  isLoggedIn := false

  isActiveSession := false

  isDenied := false

  authError := null

- side effect:
  Because the app needs to keep track of which users have a unique session active, the frontend needs to notify the server when a user signs out so their active status can be removed.

  socket.$userSignedOut(getUserId(), getUser().accessToken)$

createAccount(displayName, email, password):

- transition:
  Note that userCredential is a temporary variable in this method used to store the results of calling the Firebase API. It is not a state variable.

  userCredential := $ClientFirebase.createUserWithEmailAndPassword$(email, password)

  user := (userCrediential.user ≠ null ⇒ userCredential.user) | (userCrediential.user = null ⇒ null)

  isLoggedIn := (userCrediential.user ≠ null)

  isActiveSession := (userCrediential.user ≠ null)

  isDenied := (userCrediential.error ≠ null)

authError := isDenied ⇒ userCredential.error | ¬isDenied ⇒ null

- side effect: Calls the createUserTable method from the DatabaseOperations module on successful account creation.

  (userCrediential.user ≠ null) ⇒
  $DatabaseOperations.createUserTable$(userCredential.user.uid)

  Also notifies the socket that a new user is connected.

  socket.$checkIsOnlySession$(userCredential.user.uid, userCrediential.user.accessToken)

19 MIS of Socket

19.1 Module

Socket (inherits Socket from socket-io).

The Socket module defines a few methods using the underlying communication infrastructure of the socket-io package. This is one of the modules used to communicate between client and server, the other being DatabaseOperations.

19.2 Uses

Server

19.3 Syntax

19.3.1 Exported Constants

19.3.2 Exported Access Programs

Name	In	Out	Exceptions
checkIsOnlySession	String, String	$\mathbb{B}$	InvalidUserIdException
userSignedOut	String, String	$\mathbb{B}$	InvalidUserIdException

19.4 Semantics

19.4.1 State Variables

None.

19.4.2 Environment Variables

isClientDisconnected : $\mathbb{B}$
socket-io features a built-in disconnect event which fires when the client disconnects from the socket. This environment variable represents the event firing.

19.4.3 Assumptions

The socket-io package behaves as expected and the disconnect event fires correctly whenever a client ends their connection with the socket.

The server is always on and connected to the socket.

19.4.4 Access Routine Semantics

checkIsOnlySession(userId, token):

- output: $out := Server.userExists$(userId)
- side effect:
  If there is no active user then the socket notifies the server that a new unique session is created.
  ($Server.userExists$(userId) = false) $\Rightarrow$ $Server.addUser$(userId, token)
- exception: $exception := isValidUUID(userId) \Rightarrow$ InvalidUserIdException

userSignedOut(userId, token):

- side effect:
  $Server.deleteUser$(userId, token)
- exception: $exception := isValidUUID$(userId) $\Rightarrow$ InvalidUserIdException

The following access routine is called upon a client disconnecting (see socket-io docs).
userDisconnected(userId, token):

- side effect:
  isClientDisconnected $\Rightarrow$ $Server.deleteUser$(userId, token)
- exception: $exception := isValidUUID$(userId) $\Rightarrow$ InvalidUserIdException

19.4.5 Local Functions

isValidUUID: String $\rightarrow \mathbb{B}$
isValidUUID(userId) $\equiv$ userId $\in$ {[0-9a-fA-F]{8}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{12}}
This function makes sure that the format of the userId is a valid UUID (Universally Unique Identifier). The regular expression above represents the set of all allowable strings.

20 MIS of ClientFirebase

20.1 Module

ClientFirebase inherits firebase.auth

This module uses API credentials to give access to Firebase. For details of all syntax and semantics of exported constants and access programs, see the firebase auth package documentation.

20.2 Uses

None

20.3 Syntax

20.3.1 Exported Constants

See the firebase auth package documentation.

20.3.2 Exported Access Programs

See the firebase auth package documentation.

20.4 Semantics

20.4.1 State Variables

None.

20.4.2 Environment Variables

None.

20.4.3 Assumptions

The underlying Firebase instance is always working properly. Credentials are correct and never need to be updated.

20.4.4 Access Routine Semantics

new ClientFirebase():

- output: $out := self$
- exception: $exception :=$ InvalidCredentialException if the FIREBASE_API_KEY is invalid.

20.4.5 Local Constants

FIREBASE_API_KEY: String
The API key to access the Firebase instance.

21 MIS of User

21.1 Module

firebase.auth.User

This module is defined by Firebase. See the linked documentation for all exported constants and access routines.

22 MIS of AuthError

22.1 AuthError

firebase.auth.AuthError

This module is defined by Firebase. See the linked documentation for all exported constants.

23 MIS of CreateAccount

23.1 Module

CreateAccount

23.2 Uses

AuthState

23.3 Syntax

23.3.1 Exported Constants

None.

23.3.2 Exported Access Programs

Name	In	Out	Exceptions
setDisplayName	String		InvalidDisplayNameException
setEmail	String		InvalidEmailException
setPassword	String		InvalidPasswordException
setConfirmPassword	String		
clickCreateAccount			IncorrectCredentialsException

23.4 Semantics

23.4.1 State Variables

displayName: String
email: String
password: String
confirmPassword: String

23.4.2 Environment Variables

These environment variables capture input from the keyboard. They each correspond to their input section where a user may type.

inputDisplayName: String
inputEmail: String
inputPassword: String
inputConfirmPassword: String

23.4.3 Assumptions

An important assumption for security is that all client-server requests will use HTTPS, allowing for a secure connection when dealing with sensitive data.

It is clear to the users which section to type in concerning the environment variables (eg. The section capturing keyboard input for displayName should be labelled with 'Display Name').

23.4.4 Access Routine Semantics

setDisplayName(inputDisplayName):

- transition: displayName := inputDisplayName
- exception:
  $exception := isInvalidDisplayName(\text{inputDisplayName}) \Rightarrow \text{InvalidDisplayNameException}$

setEmail(inputEmail):

- transition: email := inputEmail
- exception: $exception := isInvalidEmail(\text{inputEmail}) \Rightarrow \text{InvalidEmailException}$

setPassword(inputPassword):

- transition: email := inputEmail
- exception: $exception := isInvalidEmail(\text{inputEmail}) \Rightarrow \text{InvalidEmailException}$

setConfirmPassword(inputConfirmPassword):

- transition: confirmPassword := inputConfirmPassword

clickCreateAccount():

- side effect:
  $(\text{password} = \text{confirmPassword}) \wedge (humanVerification() = \text{true}) \Rightarrow AuthState.createAccount(\text{displa}$
- exception: $exception := (AuthState.getAuthError() \neq \text{null} \vee \text{password} \neq \text{confirmPassword})$
  $\Rightarrow \text{IncorrectCredentialsException}$

23.4.5 Local Functions

isInvalidDisplayName: String $\rightarrow \mathbb{B}$
isInvalidDisplayName(displayName) $\equiv$ displayName $\notin$ {/(.*[[a-zA-Z0-9]]){3}/i}

This function validates the displayName input. Valid display names are at least 3 characters long and only consist of alphanumeric characters.

isInvalidEmail: String $\rightarrow \mathbb{B}$
isInvalidEmail(email) $\equiv$ email $\notin$ {/([a-zA-Z0-9.!#%&'*+/=?^_`{—} -]+@[a-zA-Z0-9-]+(?:[a-zA-Z0-9-]+)*/}

This function validates the email input. Valid emails contain only valid characters and contain an '@' in the middle.

isInvalidPassword: String $\rightarrow \mathbb{B}$
isInvalidPassword(password) $\equiv$ password $\notin$ {/.{6,}/}

This function validates the password input. Valid passwords are at least 6 characters long.

humanVerification: $\mathbb{B}$
humanVerification() $\equiv$ This function validates if the individual creating an account is human by using a CAPTCHA system. This is to prevent individuals from using automation scripts to spam the server with several bot accounts.

24 MIS of Login

24.1 Module

Login

24.2 Uses

AuthState

24.3 Syntax

24.3.1 Exported Constants

None.

24.3.2 Exported Access Programs

Name	In	Out	Exceptions
setEmail	String		InvalidEmailException
setPassword	String		InvalidPasswordException
clickLogin			IncorrectCredentialsException, InvalidSessionException

24.4 Semantics

24.4.1 State Variables

email: String
password: String

24.4.2 Environment Variables

These environment variables capture input from the keyboard. They each correspond to their input section where a user may type.

inputEmail: String
inputPassword: String

24.4.3 Assumptions

An important assumption for security is that all client-server requests will use HTTPS, allowing for a secure connection when dealing with sensitive data.

It is clear to the users which section to type in concerning the environment variables (eg. The section capturing keyboard input for the email should be labelled with 'Email').

24.4.4 Access Routine Semantics

setEmail(inputEmail):

- transition: email := inputEmail
- exception: *exception* := *isInvalidEmail*(inputEmail) ⇒ InvalidEmailException

setPassword(inputPassword):

- transition: email := inputEmail
- exception: *exception* := *isInvalidEmail*(inputEmail) ⇒ InvalidEmailException

clickLogin():

- side effect:
  AuthState.signIn(email, password)
- exception: *exception* :=
  (*AuthState.getAuthError*() ≠ null ∨ *AuthState.getIsDenied*() = true)
  ⇒ IncorrectCredentialsException
  | (*AuthState.getIsActiveSession*() = false) ⇒ InvalidSessionException

24.4.5 Local Functions

isInvalidEmail: String → 𝔹
isInvalidEmail(email) ≡ email ∉ {/([a-zA-Z0-9.!#%&'*+/=?^_`{—} -]+@[a-zA-Z0-9-]+(?:[a-zA-Z0-9-]+)*/}

This function validates the email input. Valid emails contain only valid characters and contain an '@' in the middle.

isInvalidPassword: String → 𝔹
isInvalidPassword(password) ≡ password ∉ {/÷{6,}/}

This function validates the password input. Valid passwords are at least 6 characters long.

25 MIS of ServerFirebase

25.1 Module

ServerFirebase inherits firebase-admin.auth

This module uses API credentials to give access to the server-side instance of Firebase, which can be used for privileged operations. For details of all syntax and semantics of exported constants and access programs, see the Firebase auth package documentation. Note that the firebase-admin Auth class further inherits the BaseAuth class. This should be referenced for exported access routines.

25.2 Uses

None

25.3 Syntax

25.3.1 Exported Constants

See the Firebase BaseAuth package documentation.

25.3.2 Exported Access Programs

See the Firebase BaseAuth package documentation.

25.4 Semantics

25.4.1 State Variables

None.

25.4.2 Environment Variables

None.

25.4.3 Assumptions

The underlying Firebase instance is always working properly. Credentials are correct and never need to be updated.

25.4.4 Access Routine Semantics

new ServerFirebase():

- output: $out := self$

- exception: *exception* := InvalidCredentialException if the FIREBASE-ADMIN_API_KEY is invalid.

25.4.5 Local Constants

FIREBASE-ADMIN_API_KEY: String
The API key to access the firebase-admin instance.

26 MIS of RedisClient

26.1 Module

RedisClient inherits ioredis.Redis

This module uses API credentials to give access to the Redis database hosted on the cloud (Redis Labs) specifically. Redis is typically used as a key-value store but in this case, we will use it like a set (which is natively supported).

For details of all syntax and semantics of exported constants and access programs, see the ioredis.Redis documentation.

26.2 Uses

None

26.3 Syntax

26.3.1 Exported Constants

See the ioredis.Redis documentation.

26.3.2 Exported Access Programs

See the ioredis.Redis documentation.

26.4 Semantics

26.4.1 State Variables

None.

26.4.2 Environment Variables

None.

26.4.3 Assumptions

The underlying Redis instance is always working properly. Credentials are correct and never need to be updated.

26.4.4 Access Routine Semantics

new RedisClient():

- output: $out := self$
- exception: $exception :=$ InvalidCredentialException if the REDIS_API_KEY is invalid.

26.4.5 Local Constants

REDIS_API_KEY: String
The API key to access the Redis instance.

27 MIS of Server

27.1 Module

Server

27.2 Uses

RedisClient, ServerFirebase

27.3 Syntax

27.3.1 Exported Constants

27.3.2 Exported Access Programs

Name	In	Out	Exceptions
userExists	String	$\mathbb{B}$	
addUser	String, String		
deleteUser	String, String		
checkToken		$\mathbb{B}$	InvalidTokenException

27.4 Semantics

27.4.1 State Variables

redis: RedisClient

27.4.2 Environment Variables

None.

27.4.3 Assumptions

The hardware running the server is always functional while the application is being used.

27.4.4 Access Routine Semantics

userExists(userId):

- output: $out := redis.ismember(\text{userId}) = \text{true}$

addUser(userId, token):

- side effect: $checkToken(\text{token}) \Rightarrow redis.sadd(\text{userId})$

deleteUser(userId, token):

- output: $checkToken$(token) $\Rightarrow$ $redis.srem$(userId)

checkToken(token):

- output: $out := ServerFirebase.verifyIdToken$(token)

27.4.5 Local Functions

None.

28 MIS of Seed

28.1 Module inherits Item

Seed

28.2 Uses

None

28.3 Syntax

28.3.1 Exported Constants

None

28.3.2 Exported Access Programs

Name	In	Out	Exceptions
Seed	$String, \mathbb{N}, \mathbb{R}$	-	-
getGrowthLength		$\mathbb{N}$	-
getPlantableSeasons		set of String	-
getSellValueRanges		seq of $\mathbb{N}$	-

28.4 Semantics

28.4.1 State Variables

growthLength : $\mathbb{N}$
plantableSeasons : set of String
sellValueRanges : seq of $\mathbb{N}$

28.4.2 Environment Variables

None

28.4.3 Assumptions

None

28.4.4 Access Routine Semantics

new Seed(itemName, itemCount, itemPrice, growthLength, plantableSeasons, sellValueRanges):

- transition: name, count, price, type, growthLength, plantableSeasons, sellValueRanges := itemName, itemCount, itemPrice, 'Seed', growthLength, plantableSeasons, sellValueRanges
- output: *out* := self
- exception: none

getGrowthLength():

- output: *out* := growthLength
- exception: none

getPlantableSeasons():

- output: *out* := plantableSeasons
- exception: none

getSellValueRanges():

- output: *out* := sellValueRanges
- exception: none

29 MIS of FarmTile

29.1 Module

FarmTile

29.2 Uses

Seed, Inventory, GameController

29.3 Syntax

29.3.1 Exported Constants

None

29.3.2 Exported Access Programs

Name	In	Out	Exceptions
FarmTile	$\mathbb{Z}$, $\mathbb{Z}$	FarmTile	
plantSeed	Seed, $\mathbb{N}$	-	AlreadyPlantedSeed, NoSeed, NotInSeason
getPlantedSeed	-	Seed	NoPlantedSeed
getTurnPlanted	-	$\mathbb{N}$	NoPlantedSeed
harvestCrop	-	-	InvalidHarvest, NoPlantedSeed

29.4 Semantics

29.4.1 State Variables

x : $\mathbb{Z}$
y : $\mathbb{Z}$
plantedSeed : Seed
turnPlanted : $\mathbb{N}$

29.4.2 Environment Variables

None

29.4.3 Assumptions

None

29.4.4 Access Routine Semantics

new FarmTile(x, y):

- transition: x, y := x, y
- output: *out* := self
- exception: none

plantSeed(seed, turn):

- transition: plantedSeed, turnPlanted := seed, turn
- exception: exc := plantedSeed $\neq$ null $\implies$ AlreadyPlantedSeed | $\neg\exists$(i : Item|i $\in$ Inventory.$getItems()$: i.$getName()$ = seed.$getName()$) $\implies$ NoSeed | GameController.$getSeason()$ $\notin$ seed.$getPlantableSeasons()$ $\implies$ NotInSeason

getPlantedSeed():

- output: *out* := plantedSeed
- exception: exc := (plantedSeed = null) $\implies$ NoPlantedSeed

getTurnPlanted():

- output: *out* := turnPlanted
- exception: exc := (plantedSeed = null) $\implies$ NoPlantedSeed

harvestPlant():

- transition: plantedSeed, turnPlanted := null, null
- exception: exc := plantedSeed = null $\implies$ NoPlantedSeed | $\neg$(GameController.turn $-$ turnPlanted $\geq$ plantedSeed.$getGrowthLength()$) $\implies$ InvalidHarvest

30 MIS of FarmGrid

30.1 Module

FarmGrid

30.2 Uses

FarmTile

30.3 Syntax

30.3.1 Exported Constants

None

30.3.2 Exported Access Programs

Name	In	Out	Exceptions
FarmGrid	set of FarmTile	FarmGrid	-
getTiles		set of FarmTile	-
addTile	FarmTile	-	-

30.4 Semantics

30.4.1 State Variables

tiles : set of FarmTile

30.4.2 Environment Variables

None

30.4.3 Assumptions

None

30.4.4 Access Routine Semantics

new FarmGrid(tiles):

- transition: tiles := tiles
- output: *out* := self
- exception: none

getTiles():

- output: *out* := tiles
- exception: none

addTile(tile):

- transition: tiles := tiles $\cup$ {tile}
- exception: none

31 MIS of GameController

31.1 Module

GameController

31.2 Uses

None

31.3 Syntax

31.3.1 Exported Constants

SEASONS = ['Winter', 'Spring', 'Summer', 'Fall']

31.3.2 Exported Access Programs

Name	In	Out	Exceptions
GameController	$\mathbb{N}$, $\mathbb{N}$	GameController	-
getTurn	-	$\mathbb{N}$	-
setTurn	$\mathbb{N}$	-	-
getMoney	-	$\mathbb{N}$	-
setMoney	$\mathbb{N}$	-	-
getSeason	-	String	-
getDecisionType	-	$\mathbb{Z}$	-
getMarketItems	-	set of Item	-
getAccessToConsultant	-	$\mathbb{B}$	-
setAccessToConsultant	$\mathbb{B}$	-	-
getEventHappening	-	$\mathbb{B}$	-
setEventHappening	$\mathbb{B}$	-	-
getBackgroundVolume	-	$\mathbb{R}$	-
setBackgroundVolume	$\mathbb{R}$	-	-
getSoundEffectsVolume	-	$\mathbb{R}$	-
setSoundEffectsVolume	$\mathbb{R}$	-	-
getTypeOfEvent	-	String	-
setTypeOfEvent	String	-	-
getGrid	-	set of FarmTile	-
setGrid	set of FarmTile	-	-

31.4 Semantics

31.4.1 State Variables

turn : $\mathbb{N}$
money : $\mathbb{R}$
decisionType : $\mathbb{Z}$
marketItems : set of Item
accessToConsultant : $\mathbb{B}$
isEventHappening : $\mathbb{B}$
backgroundMusicVolume : $\mathbb{R}$
soundEffectsVolume : $\mathbb{R}$
typeOfEvent : String
grid : set of FarmTile

31.4.2 Environment Variables

None

31.4.3 Assumptions

None

31.4.4 Access Routine Semantics

new GameController(turn, money):

- transition: turn, money := turn, money
- output: *out* := self
- exception: None

getTurn():

- output: *out* := turn
- exception: None

setTurn(turn):

- transition: turn := turn
- exception: None

getMoney():

- output: *out* := money
- exception: None

setMoney(money):

- transition: money := money
- exception: None

getSeason():

- output: *out* := SEASONS[⌊(turn/4)⌋ % 4]
- exception: None

getDecisionType():

- output: *out* := decisionType
- exception: None

getMarketItems():

- output: *out* := marketItems
- exception: None

getAccessToConsultant():

- output: *out* := accessToConsultant
- exception: None

setAccessToConsultant(access):

- transition: accessToConsultant := access
- exception: None

getEventHappening():

- output: *out* := isEventHappening
- exception: None

setEventHappening(isNextEventHappening):

- output: isEventHappening := isNextEventHappening
- exception: None

getBackgroundVolume():

- output: *out* := backgroundMusicVolume
- exception: None

setBackgroundVolume(changedVolume):

- output: backgroundMusicVolume := changedVolume
- exception: None

getSoundEffectsVolume():

- output: *out* := soundEffectsVolume
- exception: None

setSoundEffectsVolume(changedVolume):

- output: soundEffectsVolume := changedVolume
- exception: None

getTypeOfEvent():

- output: *out* := typeOfEvent
- exception: None

setTypeOfEvent(eventType):

- output: typeOfEvent := eventType
- exception: None

getGrid():

- output: *out* := grid
- exception: None

setGrid(updatedGrid):

- output: grid := updatedGrid
- exception: None

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.

32 Appendix

N/A