Module Interface Specification for Farming Matters

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

January 18, 2023

1 Revision History

Date	Version	Notes
1/18/2023	1.0	Finished First Version

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 Market Inventory FarmGrid GameSettings CreateAccount Login TurnSummary
Software-Hiding Module	Avatar Consultant OtherAvatars FarmTile Seed Item DatabaseOperations ServerFirebase ClientFirebase AuthState Socket RedisClient Server MusicPlayer User AuthError

Table 1: Module Hierarchy

6 MIS of Avatar

6.1 Module

Avatar

6.2 Uses

AvatarMenu

6.3 Syntax

6.3.1 Exported Constants

None

6.3.2 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	

6.4 Semantics

6.4.1 State Variables

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

6.4.2 Environment Variables

None

6.4.3 Assumptions

None

6.4.4 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

7 MIS of AvatarMenu

7.1 Module

AvatarMenu

7.2 Uses

GameController, Avatar

7.3 Syntax

7.3.1 Exported Constants

None

7.3.2 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			

7.4 Semantics

7.4.1 State Variables

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

7.4.2 Environment Variables

None

7.4.3 Assumptions

None

7.4.4 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's 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

7.4.5 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

8 MIS of Consultant

8.1 Module

Consultant

8.2 Uses

GameController, Avatar

8.3 Syntax

8.3.1 Exported Constants

None

8.3.2 Exported Access Programs

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

8.4 Semantics

8.4.1 State Variables

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

8.4.2 Environment Variables

None

8.4.3 Assumptions

None

8.4.4 Access Routine Semantics

new Consultant(userDecisionType):

- transition: $decisionType, statement := userDecisionType, genereateStatement()$
- output: $out :=$ self
- exception: none

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

8.4.5 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.

observeSeason: GameController $\rightarrow$ seq of String
observeSeason(game) $\equiv$ Observe the current game state relevant to the current season and the season after. Consider which items are available to be purchased, the current item(s) the user has, the selling price of items that will yield the most profit, etc.

observeInventory: GameController $\rightarrow$ seq of String
observeSeason(game) $\equiv$ Observe the users inventory. Consider which items are still growing, what still needs to be planted, availability of space on the farm left to plant while also considering an optimal planting strategy, etc.

observeAllGameState: GameController $\rightarrow$ seq of String
observeAllGameState(game) $\equiv$ Observe the current game state relevant to the current season and the season after. It will use the local functions above that are related to observing a specific aspect of the users current game state. Consider which items are available to be purchased, the current item(s) the user has, the selling price of items that will yield the most profit, etc.

relatedStatements: String $\times$ GameController $\rightarrow$ seq of String
relatedStatements(statisticValue, game) $\equiv$ Taking the current state of the game, observe numerous aspects of the game that are relevant to the current users situtation (i.e considering the current season, current turn, current inventory, current shop market), set out a relevant prediction based on several parameters.

generateStatement: $\mathbb{Z} \times$ GameController $\rightarrow$ String
generateStatement(decisionType, game) $\equiv$ ($decisionType = PROBABLISTIC\_ID \Rightarrow$
$relatedStatements(generateProbablisticValue(), game)|$
$decisionType = DETERMINISTIC\_ID \Rightarrow$
$relatedStatements(generateDeterministicValue(), game)$)

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

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

8.4.6 Local Constants

PROBABLISTIC_ID = $\mathbb{Z}$
DETERMINISTIC_ID = $\mathbb{Z}$

9 MIS of OtherAvatar

9.1 Module

OtherAvatar

9.2 Uses

Avatar

9.3 Syntax

9.3.1 Exported Constants

None

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
newOtherAvatar		OtherAvatar	

9.4 Semantics

9.4.1 State Variables

statement : String

9.4.2 Environment Variables

None

9.4.3 Assumptions

None

9.4.4 Access Routine Semantics

new OtherAvatar():

- transition: *statement* := *genereateStatement*()
- output: *out* := self
- exception: none

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.

generateStatement: $\mathbb{Z} \rightarrow$ String
generateStatement(avatarID) $\equiv$
$DEFAULT\_STATEMENTS[randomIndex(DEFAULT\_STATEMENTS)]$
This function will randomly select a default statement from a seq of default statements that are associated with a specific OtherAvatar.

9.4.6 Local Constants

DEFAULT_STATEMENTS = seq of String

10 MIS of Item

10.1 Module

Item

10.2 Uses

None

10.3 Syntax

10.3.1 Exported Constants

None

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
newItem	String,$\mathbb{Z}$,$\mathbb{R}$,String	Item	
getName		String	
getCount		$\mathbb{Z}$	
getPrice		$\mathbb{R}$	
getType		String	
setCount	$\mathbb{Z}$		Illegal Argument Exception
setPrice	$\mathbb{R}$		Illegal Argument Exception

10.4 Semantics

10.4.1 State Variables

name : String
count : $\mathbb{Z}$
price : $\mathbb{R}$
type : String

10.4.2 Environment Variables

None

10.4.3 Assumptions

None

10.4.4 Access Routine Semantics

new Item($itemName$, $itemCount$, $itemPrice$, $itemType$):

- transition: $name, count, price, type := itemName, itemCount, itemPrice, itemType$,
- output: $out :=$ self
- exception: none

getName():

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

getCount():

- output: $out := count$
- exception: none

getPrice():

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

getType():

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

setCount(quantity):

- transition: $count := quantity$
- exception: exc := $(quantity < 0) \Rightarrow$ IllegalArgumentException

setPrice(newPrice):

- transition: $price := newPrice$
- exception: exc := $(newPrice \leq 0) \Rightarrow$ IllegalArgumentException

11 MIS of Inventory

11.1 Module

Inventory

11.2 Uses

Item

11.3 Syntax

11.3.1 Exported Constants

None

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
newInventory		Inventory	
getItems		set of Item	
addItem	Item,$\mathbb{Z}$		
removeItem	Item,$\mathbb{Z}$		Illegal Argument Exception

11.4 Semantics

11.4.1 State Variables

inventory : set of Item

11.4.2 Environment Variables

None

11.4.3 Assumptions

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

11.4.4 Access Routine Semantics

new Inventory():

- transition: $inventory := \{\}$
- output: $out :=$ self
- exception: none

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

- output: $out := \{i : \text{Item} | i \in inventory : i\}$
- exception: none

addItem(item,quantity):

- transition: $inventory := \{i : \text{Item} | i \in inventory : (i.name = item) \Rightarrow (i.count = i.count + quantity) | \text{True} \Rightarrow inventory \cup \{item\} where\ item.count = quantity\}$
- exception: none

Cases:
1)The item exists in the inventory → change the count of the item by the quantity provided.
2) The item does not exist → add it to the inventory and set the count of the item to the quantity provided.
removeItem(item,quantity):

- transition: $inventory := \{i : \text{Item} | i \in inventory : (i.name = item) \wedge (i.count = quantity) \Rightarrow inventory - \{item\} | (i.name = item) \wedge (i.count > quantity) \Rightarrow inventory\ where\ (i.count = i.count - quantity)\}$
- exception: $exc := (quantity < 0) \Rightarrow$ IllegalArgumentException$|(item \notin inventory) \Rightarrow$ IllegalArgumentException$|(item \in inventory) \wedge (item.count < quantity) \Rightarrow$ IllegalArgumentException

Cases:
1) Item exists in inventory and quantity is less than item count.
2) Item exists in inventory and quantity equals item count
3) Item exists in inventory and quantity greater than item count(Exception)
4) Item does not exist in inventory(Exception)

12 MIS of Market

12.1 Module

Market

12.2 Uses

Items, Inventory

12.3 Syntax

12.3.1 Exported Constants

None

12.3.2 Exported Access Programs

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

12.4 Semantics

12.4.1 State Variables

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

12.4.2 Environment Variables

None

12.4.3 Assumptions

None

12.4.4 Access Routine Semantics

buyItems(item,amount):

- transition:
  $balance := (balance - amount \times item.price)\ \ where\ \ (item \in shopItems)$
  $inventory := inventory.addItem(item, amount)$

- exception: $exc := (quantity < 0) \vee (item \notin shopItems) \vee$
  $(balance < amount \times item.price) \Rightarrow$ IllegalArgumentException

sellItems(item,amount) :

- transition:
  $balance := (balance + amount \times fluctuatePrice(RandomNum(), item.price))\ \ where\ \ (item \in shopItems) \wedge (item \in inventory)$
  $inventory := inventory.removeItem(item, amount)$

- exception: $exc := (quantity < 0) \vee (item \notin shopItems) \Rightarrow$ IllegalArgumentException

purchaseInsurance(item,amount):

- transition:
  $balance := (balance - amount \times (item.price/4))\ \ where\ \ (item \in shopItems) \wedge (item \in inventory)$
  $inventory := \{i : \text{Item} | i \in inventory : (i.name = item) \Rightarrow i.price = max\ where\ max = (item.price \geq fluctuatePrice(RandomNum(), item.price) \Rightarrow item.price | item.price < item.price \Rightarrow fluctuatePrice(RandomNum(), item.price))\}$

- exception: none
  Purchasing insurance on an item will mean the farmer is guaranteed not to lose any money when selling the insured item. Since prices fluctuate it is possible the price that the item gets sold at is lower than the buy price. This is meant to ensure that the buyer will at least get back the amount paid for the item.

12.4.5 Local Functions

randomNum: void $\rightarrow \mathbb{R}$
randomNum() $\equiv i : \mathbb{R} | -1 \leq i \leq 1$
This function produces a random number between -1 and 1.
fluctuatePrice: $\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$
fluctuatePrice(randomFactor,basePrice) $\equiv basePrice + randomFactor \times (basePrice/3)$ This function will allow the user to sell crops at fluctuating prices as mentioned in FR8 of the SRS.

13 MIS of DatabaseOperations

13.1 Module

DatabaseOperations

13.2 Uses

AuthState, GameController

13.3 Syntax

13.3.1 Exported Constants

None

13.3.2 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		IllegalArgumentException
saveGame	$\mathbb{Z}$, set of MapGameStates		IllegalArgumentException
loadGame	$\mathbb{Z}$	set of MapGameStates	IllegalArguementException

13.4 Semantics

13.4.1 State Variables

isConnected: $\mathbb{B}$

13.4.2 Environment Variables

None

13.4.3 Assumptions

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

13.4.4 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 request to stop the connection. If the request is successful, return True and False otherwise. Return False, if the user is not logged or there is not a 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 the user is logged, 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 the user is logged, 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 the user is logged 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 the user is logged 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 the user is logged 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.

13.4.5 Local Constants

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

14 MIS of MusicPlayer

14.1 Module

MusicPlayer

14.2 Uses

None

14.3 Syntax

14.3.1 Exported Constants

None

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
loadMusic	String		InvalidFile
startMusic			
stopMusic			

14.4 Semantics

14.4.1 State Variables

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

14.4.2 Environment Variables

None

14.4.3 Assumptions

None

14.4.4 Access Routine Semantics

loadMusic(audioFileName):

- transition: $turn, money := turn, money$

- 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():

- transition: $audioFile :=$ If isPlaying is False, start playing the audio file in a infinte loop, otherwise continue playing the music.
  $isPlaying := True$
- exception: None

stopMusic():

- transition: $audioFile :=$ If isPlaying is False, stops playing the audio file, otherwise do nothing.
  $isPlaying := False$
- exception: none

14.4.5 Local Functions

None

15 MIS of GameSettings

15.1 Module

GameSettings

15.2 Uses

DatabaseOperations, MusicPlayer

15.3 Syntax

15.3.1 Exported Constants

None

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
getVolume		$\mathbb{N}$	
setVolume	$\mathbb{N}$		IllegalArguementException
changeVolumne	$\mathbb{Z}$		IllegalArguementException
optOutOfStudy	$\mathbb{Z}$		

15.4 Semantics

15.4.1 State Variables

currentVolume: $\mathbb{N}$

15.4.2 Environment Variables

None

15.4.3 Assumptions

None

15.4.4 Access Routine Semantics

getVolume():

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

setVolume(newVolume):

- transition: $currentVolumne := newVolume$
- exception: $(newVolume < 0 \lor newVolumne > 100) \Rightarrow IllegalArgumentException$

changeVolume(relativeChangeVolume):

- transition: $currentVolumne := currentVolumne + newVolume$
- exception: $((currentVolumne+newVolume) < 0 \lor (currentVolumne+newVolumne) > 100) \Rightarrow IllegalArgumentException$

optOutOfStudy(userId):

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

16 MIS of AuthState

16.1 Module

AuthState

16.2 Uses

User, AuthError, Socket, ClientFirebase, DatabaseOperations

16.3 Syntax

16.3.1 Exported Constants

None

16.3.2 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.

16.4 Semantics

16.4.1 State Variables

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

16.4.2 Environment Variables

None

16.4.3 Assumptions

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

16.4.4 Access Routine Semantics

getUser():

- output: $out := ((isLoggedIn \land \neg isDenied) \Rightarrow user) \mid (\text{True} \Rightarrow \text{null})$

getUserId():

- output: $out := ((isLoggedIn \land \neg isDenied) \Rightarrow user.uid) \mid (\text{True} \Rightarrow \text{null})$

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.

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

  $user := (userCrediential.user \neq \text{null} \Rightarrow userCredential.user) \mid (userCrediential.user = \text{null} \Rightarrow \text{null})$

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

$isActiveSession :=$
$(userCrediential.user \neq \text{null})$
$\Rightarrow socket.checkIsOnlySession(userCredential.user.uid, userCrediential.user.accessToken)$
$\mid (userCrediential.user = \text{null}) \Rightarrow \text{false}$

$isDenied := (userCrediential.error \neq \text{null})$

$authError := isDenied \Rightarrow userCredential.error \mid \neg isDenied \Rightarrow \text{null}$

signOut():

- transition:
  $user := \text{null}$
  $isLoggedIn := \text{false}$
  $isActiveSession := \text{false}$
  $isDenied := \text{false}$
  $authError := \text{null}$
- 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.

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

  $user := (userCrediential.user \neq \text{null} \Rightarrow userCredential.user) \mid (userCrediential.user = \text{null} \Rightarrow \text{null})$

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

  $isActiveSession := (userCrediential.user \neq \text{null})$

  $isDenied := (userCrediential.error \neq \text{null})$

$authError := isDenied \Rightarrow userCredential.error \mid \neg isDenied \Rightarrow \text{null}$

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

  $(userCrediential.user \neq \text{null}) \Rightarrow$
  $DatabaseOperations.createUserTable(userCredential.user.uid)$

  Also notifies the socket that a new user is connected.

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

17 MIS of Socket

17.1 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.

17.2 Uses

Server

17.3 Syntax

17.3.1 Exported Constants

17.3.2 Exported Access Programs

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

17.4 Semantics

17.4.1 State Variables

None.

17.4.2 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.

17.4.3 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.

17.4.4 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.
  $(Server.userExists(userId) = \text{false}) \Rightarrow Server.addUser(userId, token)$
- 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

17.4.5 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.

18 MIS of ClientFirebase

18.1 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.

18.2 Uses

None

18.3 Syntax

18.3.1 Exported Constants

See the firebase auth package documentation.

18.3.2 Exported Access Programs

See the firebase auth package documentation.

18.4 Semantics

18.4.1 State Variables

None.

18.4.2 Environment Variables

None.

18.4.3 Assumptions

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

18.4.4 Access Routine Semantics

new ClientFirebase():

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

18.4.5 Local Constants

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

19 MIS of User

19.1 Module

firebase.auth.User

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

20 MIS of AuthError

20.1 AuthError

firebase.auth.AuthError

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

21 MIS of CreateAccount

21.1 Module

CreateAccount

21.2 Uses

AuthState

21.3 Syntax

21.3.1 Exported Constants

None.

21.3.2 Exported Access Programs

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

21.4 Semantics

21.4.1 State Variables

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

21.4.2 Environment Variables

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

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

21.4.3 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 sections to type in with respect to the environment variables (eg. The section capturing keyboard input for displayName should be labelled with 'Display Name').

21.4.4 Access Routine Semantics

setDisplayName(inputDisplayName):

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

setEmail(inputEmail):

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

setPassword(inputPassword):

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

setConfirmPassword(inputConfirmPassword):

- transition: $confirmPassword := inputConfirmPassword$

clickCreateAccount():

- side effect:
  $(password = confirmPassword) \Rightarrow AuthState.createAccount(displayName, email, password)$
- exception: $exception := (AuthState.getAuthError() \neq \text{null} \vee password \neq confirmPassword)$
  $\Rightarrow IncorrectCredentialsException$

21.4.5 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 consists 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.

22 MIS of Login

22.1 Module

Login

22.2 Uses

AuthState

22.3 Syntax

22.3.1 Exported Constants

None.

22.3.2 Exported Access Programs

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

22.4 Semantics

22.4.1 State Variables

email: String
password: String

22.4.2 Environment Variables

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

inputEmail: String
inputPassword: String

22.4.3 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 sections to type in with respect to the environment variables (eg. The section capturing keyboard input for email should be labelled with 'Email').

22.4.4 Access Routine Semantics

setEmail(inputEmail):

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

setPassword(inputPassword):

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

clickLogin():

- side effect:
  $AuthState.signIn(email, password)$
- exception: $exception :=$
  $(AuthState.getAuthError() \neq \text{null} \ \vee \ AuthState.getIsDenied() = \text{true})$
  $\Rightarrow IncorrectCredentialsException$
  $|\ (AuthState.getIsActiveSession() = \text{false}) \Rightarrow InvalidSessionException$

22.4.5 Local Functions

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.

23 MIS of ServerFirebase

23.1 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.

23.2 Uses

None

23.3 Syntax

23.3.1 Exported Constants

See the Firebase BaseAuth package documentation.

23.3.2 Exported Access Programs

See the Firebase BaseAuth package documentation.

23.4 Semantics

23.4.1 State Variables

None.

23.4.2 Environment Variables

None.

23.4.3 Assumptions

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

23.4.4 Access Routine Semantics

new ServerFirebase():

- output: $out := self$

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

23.4.5 Local Constants

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

24 MIS of RedisClient

24.1 Module

RedisClient inherits ioredis.Redis

This module uses API credentials to give access to the a 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.

24.2 Uses

None

24.3 Syntax

24.3.1 Exported Constants

See the ioredis.Redis documentation.

24.3.2 Exported Access Programs

See the ioredis.Redis documentation.

24.4 Semantics

24.4.1 State Variables

None.

24.4.2 Environment Variables

None.

24.4.3 Assumptions

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

24.4.4 Access Routine Semantics

new RedisClient():

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

24.4.5 Local Constants

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

25 MIS of Server

25.1 Module

Server

25.2 Uses

RedisClient, ServerFirebase

25.3 Syntax

25.3.1 Exported Constants

25.3.2 Exported Access Programs

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

25.4 Semantics

25.4.1 State Variables

redis: RedisClient

25.4.2 Environment Variables

None.

25.4.3 Assumptions

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

25.4.4 Access Routine Semantics

userExists(userId):

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

addUser(userId, token):

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

deleteUser(userId, token):

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

checkToken(token):

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

25.4.5 Local Functions

None.

26 MIS of Seed

26.1 Module inherits Item

Seed

26.2 Uses

None

26.3 Syntax

26.3.1 Exported Constants

None

26.3.2 Exported Access Programs

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

26.4 Semantics

26.4.1 State Variables

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

26.4.2 Environment Variables

None

26.4.3 Assumptions

None

26.4.4 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

26.4.5 Local Functions

None

27 MIS of FarmTile

27.1 Module

FarmTile

27.2 Uses

Seed, Inventory, GameController

27.3 Syntax

27.3.1 Exported Constants

None

27.3.2 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

27.4 Semantics

27.4.1 State Variables

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

27.4.2 Environment Variables

None

27.4.3 Assumptions

None

27.4.4 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: $exc := plantedSeed \neq null \implies AlreadyPlantedSeed \mid$
  $\neg\exists(i : Item | i \in Inventory.getItems() : i.getName() = seed.getName()) \implies NoSeed \mid$
  $GameController.getSeason() \notin seed.getPlantableSeasons() \implies NotInSeason$

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

27.4.5 Local Functions

None

28 MIS of FarmGrid

28.1 Module

FarmGrid

28.2 Uses

FarmTile

28.3 Syntax

28.3.1 Exported Constants

None

28.3.2 Exported Access Programs

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

28.4 Semantics

28.4.1 State Variables

tiles : set of FarmTile

28.4.2 Environment Variables

None

28.4.3 Assumptions

None

28.4.4 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

28.4.5 Local Functions

None

29 MIS of GameController

29.1 Module

GameController

29.2 Uses

None

29.3 Syntax

29.3.1 Exported Constants

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

29.3.2 Exported Access Programs

Name	In	Out	Exceptions
GameController	$\mathbb{N}$, $\mathbb{N}$	GameController	-
getTurn	-	$\mathbb{N}$	-
getMoney	-	$\mathbb{N}$	-
getSeason	-	String	-
setTurn	$\mathbb{N}$	-	-
setMoney	$\mathbb{N}$	-	-

29.4 Semantics

29.4.1 State Variables

$turn : \mathbb{N}$
$money : \mathbb{N}$

29.4.2 Environment Variables

None

29.4.3 Assumptions

None

29.4.4 Access Routine Semantics

new GameController(turn, money):

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

getTurn():

- output: $out := turn$
- exception: none

getMoney():

- output: $out := money$
- exception: none

getSeason():

- output: $out := SEASONS[\lfloor (turn/4) \rfloor \ \% \ 4]$
- exception: none

setTurn(turn):

- transition: $turn := turn$
- exception: none

setMoney(money):

- transition: $money := money$
- exception: none

29.4.5 Local Functions

None

30 Appendix

N/A