SE 3XA3: Module Interface Specification (MIS) BigTwo

Team 06, Team Name: Aplus^3
Senni Tan, tans28
Manyi Cheng, chengm33
Jiaxin Tang, tangj63

March 19, 2021

Contents

List of Tables

Table 1: **Revision History**

Date	**Version**	**Notes**
Mar 16, 2021	0.0	Initial Draft

1 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 2. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Hardware-Hiding Module: Hardware-Hiding Module

M2: Behaviour-Hiding Module: Scene Module

M3: Behaviour-Hiding Module: Card Module

M4: Behaviour-Hiding Module: Player Module

M5: Behaviour-Hiding Module: PlayerBot Module

M6: Behaviour-Hiding Module: Rules Module

M7: Behaviour-Hiding Module: Game Module

M8: Behaviour-Hiding Module: GameplayField Module

Level 1	**Level 2**
Hardware-Hiding Module	Hardware-Hiding Module
Behaviour-Hiding Module	Scene Module Card Module Player Module PlayerBot Module Rules Module Game Module GameplayField Module
Software Decision Module	N/A9No generic type)

Table 2: Module Hierarchy

2 MIS of Scene Module

2.1 Uses

Game

2.2 Interface Syntax

2.2.1 Exported Types

Scene = ?

2.2.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
Scene	Game	Scene	InvalidInput
display	-	Screen	-

2.3 Interface Semantics

2.3.1 State Variables

game: Game

2.3.2 Environmental Variables

None

2.3.3 Assumptions

The constructor Scene is called for the object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

2.3.4 Access Program Semantics

Scene(*game*):

- transition: $game := game$
- output: $out := self$
- exception := exc := ((typeof(game) $\neq$ Game) $\Rightarrow$ InvalidInput)

display():

- output := output each component in the Game module with the expected image in the image folder to the screen.
- exception := None

3 MIS of Card Module

3.1 Interface Syntax

3.1.1 Exported Types

SuiteT = {Spade, Heart, Club, Diamond} Enum inside the Card module
NumT = {A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K}Enum inside the Card module
Card = ?

3.1.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
Card	SuiteT, NumT, String	Card	InvalidInput $\vee$ FileNotFound
rank	-	Z	-
image	-	String	-

3.2 Interface Semantics

3.2.1 State Variables

suit:SuiteT
num:NumT
image:String A string contains the location of the image file for the card

3.2.2 Environmental Variables

None

3.2.3 Assumptions

The constructor Card is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

3.2.4 Access Program Semantics

Card($suite, num, image$):

- transition: $suite, num, image := suite, num, image$
- output: $out := self$
- exception $:= exc :=$ ((typeof(suite) $\neq$ SuiteT) $\vee$ (typeof(num) $\neq$ NumT) lor (typeof(image) $\neq$ String) $\Rightarrow$ InvalidInput) $\wedge$ (can not find file at image location $\Rightarrow$ FileNotFound)

rank():

- output := out := ((num == A) ⇒ 2) ∨ ((num == 2) ⇒ 1) ∨ ((num == 3) ⇒ 13) ∨ ((num == 4) ⇒ 12) ∨ ((num == 5) ⇒ 11) ∨ ((num == 6) ⇒ 10) ∨ ((num == 7) ⇒ 9) ∨ ((num == 8) ⇒ 8) ∨ ((num == 9) ⇒ 7) ∨ ((num == 10) ⇒ 6) ∨ ((num == J) ⇒ 5) ∨ ((num == Q) ⇒ 4) ∨ ((num == K) ⇒ 3)
- exception := None

image()

- output := out := image
- exception := None

4 MIS of Player Module

4.1 Uses

Game

4.2 Interface Syntax

4.2.1 Exported Types

None

4.2.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
Player	Game	Player	InvaidInput
selectCard	mouse click on screen	-	-
selectPlay	mouse click on screen	-	InvalidCombination
selectPass	mouse click on screen	-	-
NumSort	mouse click on screen	-	-
SuiteSort	mouse click on screen	-	-
Restart	mouse click on screen	-	-
Exit	mouse click on screen	-	-

4.3 Interface Semantics

4.3.1 State Variables

game: Game
selectCards: seq of Card
cards: seq of Card

4.3.2 Environmental Variables

None

4.3.3 Assumptions

The constructor Player is called for the object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

4.3.4 Access Program Semantics

Player($game$):

- transition: $game, selectCards, cards := game, [], assignedcardsfromthegame$
- output: $out := self$
- exception: exc := ((typeof(game) $\neq$ Game) $\Rightarrow$ InvalidInput)

selectCard(screen):

- transition: $selectCards$:= add the card(s) that is(are) clicked on the screen
- exception: None

selectPlay(screen):

- transition: $selectCards, cards$:= [], remove cards in $selectCards$ from $cards$[ ]
- exception: exc := $\neg$ (checkSingle($selectCards$) $\vee$ checkPair($selectCards$) $\vee$ checkFive($selectCards$) $\vee$ checkThree($selectCards$) $\vee$ checkFour($selectCards$)) $\Rightarrow$ InvalidCombination

selectPass(screen):

- transition: pass the turn for the player in the game
- exception: None

NumSort(screen):

- transition: sort $cards$[ ] in the number rank order.
- exception: None

SuiteSort(screen):

- transition: sort $cards$[ ] in the Suite rank order.
- exception: None

Restart():

- transition: restart the game
- exception: None

Exit():

- transition: exit the game and return to main menu
- exception: None

4.3.5 Private Methods

checkSingle($selectCards$):

- output: out := (len($selectCards$) == 1) $\Rightarrow$ True | False

checkPair($selectCards$):

- output: out := ((len($selectCards$) == 2) $\wedge$ ($selectCards$[0] == $selectCards$[1])) $\Rightarrow$ True | False

checkFive($selectCards$):

- output: out := (len($selectCards$) == 5) $\wedge$ (isStraight($selectCards$) $\vee$ isFlush($selectCards$) $\vee$ isFullHouse($selectCards$) $\vee$ isFullHouse2($selectCards$) $\vee$ isStraightFlush($selectCards$)) $\Rightarrow$ True | False

checkThree($selectCards$):

- output: out := (len($selectCards$) == 3) $\wedge$ ($selectCards$[0] == $selectCards$[1]) $\wedge$ ($selectCards$[1] == $selectCards$[2])) $\Rightarrow$ True | False

checkFour($selectCards$):

- output: out := (len($selectCards$) == 3) $\wedge$ ($selectCards$[0] == $selectCards$[1]) $\wedge$ ($selectCards$[1] == $selectCards$[2]) $\wedge$ ($selectCards$[2] == $selectCards$[3])) $\Rightarrow$ True | False

isStraight($selectCards$):

- output: out := ((NumSort(selectCards)) $\wedge$ ((selectCards[0].rank() < selectCards[1].rank()) $\wedge$ (selectCards[1].rank() < selectCards[2].rank()) $\wedge$ (selectCards[2].rank() < selectCards[3].rank()) $\wedge$ (selectCards[3].rank() < selectCards[4].rank())) $\Rightarrow$ True | False

isFlush($selectCards$):

- output: *out* := (selectCards[0].suite == selectCards[1].suite) ∧ (selectCards[1].suite == selectCards[2].suite) ∧ (selectCards[2].suite == selectCards[3].suite) ∧ (selectCards[3].suite == selectCards[4].suite) ⇒ True | False

isFullHouse($selectCards$):

- output: *out* := (NumSort(selectCards) ∧ (checkThree([selectCards[0], selectCards[1], selectCards[2]]) ∧ checkPair([selectCards[3], selectCards[4]]) ∨ (checkThree([selectCards[2], selectCards[3], selectCards[4]]) ∧ checkPair([selectCards[0], selectCards[1]])) ⇒ True | False

isFullHouse2($selectCards$):

- output: *out* := (NumSort(selectCards) ∧ (checkFour([selectCards[0], selectCards[1], selectCards[2], selectCards[3]]) ∧ checkSingle([selectCards[4]]) ∨ (checkFour([selectCards[1], selectCards[2], selectCards[3], selectCards[4]]) ∧ checkSingle([selectCards[0]])) ⇒ True | False

isStraightFlush($selectCards$):

- output: *out* := (isStraight(selectCards) ∧ isFlush(selectCards)) ⇒ True | False

5 MIS of PlayerBot Module

5.1 Uses

Card, Rules

5.2 Interface Syntax

5.2.1 Exported Types

PlayerBot = ?

5.2.2 Exported Access Programs

Name	In	Out	Exceptions
playerBot	Sequence of Card	PlayerBot	
playCards	Sequence of Card, Sequence of Card		
playInitTurn			
checkSingle	Sequence of Card, Sequence of Card	Card	
checkPair	Sequence of Card, Sequence of Card	Sequence of Card	
checkFive	Sequence of Card , Sequence of Card	Sequence of Card	
removeSet	Sequence of Card		
passTurn			

5.3 Interface Semantics

5.3.1 State Variables

cards: Sequence of Card // Contains all the cards owned by the current computer player

last: Sequence of Card// Contains all the cards played by last player

5.3.2 Environmental Variables

5.3.3 Assumptions

The constructor of playerBot is called for each instance before any access routine is called for that object. The constructor cannot be called on an existing object.

5.3.4 Access Program Semantics

playerBot(s):
Input: A list of cards owned by the current playerBot.
Transition: Initialize the state variables.
cards := s
last := []
Output: out := self
Exceptions: None

playCards(s, l):
Input: A list of cards owned by the current playerBot. A list of cards played by the last player.
Transition: Check if the current player is the intial player, if it is then calls playInitTurn(), else checks the length of l.
If l.length == 1, calls checkSingle().
If l.length == 2, calls checkPair().
If l.length == 5, calls checkFive.
Let validSet := checkSingle()/checkPair()/checkFive(). If validSet == null, calls passTurn(), else
cards := removeSet(validSet)
last := validSet
Output: None
Exceptions: None

playInitTurn()
Input: None
Transition: Removes diamond 3 from the state variable cards and updates the state variable last.
cards := cards.remove(c) where c.suite == 'Diamond' $\wedge$ c.num == '3'

last := c where c.suite == ’Diamond’ $\wedge$ c.num == ’3’
Output: None
Exceptions: None

checkSingle(s, l):
Input: A list of cards owned by the current playerBot. A list of cards played by the last player.
Transition: None
Output: Valid Card to be played.
Exceptions: None

checkPair(s, l):
Input: A list of cards owned by the current playerBot. A list of cards played by the last player.
Transition: None
Output: Valid pair of Cards to be played.
Exceptions: None

checkFive(s, l):
Input: A list of cards owned by the current playerBot. A list of cards played by the last player.
Transition: None
Output: Valid combination of Cards to be played.
Exceptions: None

removeSet(validPlay):
Input: A list of cards owned by the current playerBot. A list of valid combination of cards.
Transition: cards := $\forall(c : Card | c \in validPlay : cards.remove(c))$
Output: None
Exceptions: None

passTurn()
Input: None
Transition: Goes to next player
Output: None
Exceptions: None

6 MIS of Rules Module

6.1 Uses

Card

6.2 Interface Syntax

6.2.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
rules			
newDeck			
shuffle			
setPlayerCards			
NumSort	Sequence of Card		
SuiteSort	Sequence of Card		
isInitPlayer	Sequence of Card	boolean	
isVaildPair	Sequence of Card	boolean	
isVaildStraight	Sequence of Card	boolean	
isVaildFlush	Sequence of Card	boolean	
isValidFullHouse	Sequence of Card	boolean	
isValidFourOfaKind	Sequence of Card	boolean	
isStrongerPlay	Sequence of Card	boolean	
isStrongerSingle	Card	boolean	
isStrongerPair	Sequence of Card	boolean	
isStrongerFive	Sequence of Card	boolean	
palyCards	Sequence of Card		

6.3 Interface Semantics

6.3.1 State Constants

Suite = {Spade, Heart, Club, Diamond}
Num = {A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K}

6.3.2 State Variables

Deck: Sequence of Card
Player1 : Sequence of Card
Player2 : Sequence of Card
Player3 : Sequence of Card
Player4 : Sequence of Card
last: Sequence of Card // Contains all the cards played by the last player.

6.3.3 Assumptions

The constructor of rules is called for each instance before any access routine is called for that object.

shuffle() must be called before setPlayerCards().

6.3.4 Access Program Semantics

rules():
Input: None
Transition: Deck := [ ]
Player1 := [ ]
Player2 := [ ]
Player3 := [ ]
Player4 := [ ]
last := [ ]
Output: out := self
Exceptions: None

newDeck():
Input: None
Transition : Deck := ($\forall$ i, j : $\mathbf{N}$, c : Card | i $\in$ [0..3], j $\in$ [0..12] : Deck.add(c) where c.suite := Suite[i] $\wedge$ c.num := Num[j]).
Output: None
Exceptions: None

shuffle():
Input: None
Transition : Updates Deck by rearranging the order of Cards in Deck randomly.
Output: None
Exceptions: None

setPalyerCards():
Input: None
Transition: Player1 := $\forall$ i : $\mathbf{N}$| i $\in$ [0..12] : Player1.add(Deck[i])
Player2 := $\forall$ i : $\mathbf{N}$| i $\in$ [13..25] : Player2.add(Deck[i])
Player3 := $\forall$ i : $\mathbf{N}$| i $\in$ [26..38] : Player3.add(Deck[i])
Player4 := $\forall$ i : $\mathbf{N}$| i $\in$ [39..51] : Player4.add(Deck[i])
Output: None
Exceptions: None

NumSort(s):

Input: A list of Cards
Transition: None
Output : out := a list of Cards from s sorted in the number rank order
Exceptions: None

SuiteSort(s):
Input: A list of Cards
Transition: None
Output : out := a list of Cards from s sorted in the suite rank order
Exceptions: None

isInitPlayer(s):
Input: A list of the Cards owned by the current player.
Transition: None
Output : out := ($\exists$ i : $\mathbf{N}$| i $\in$ [0..s.length] : s[i].suite == 'Diamond' $\wedge$ s[i].num == '3') $\Rightarrow$ true Exceptions: None

isValidPair(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := s.length == 2 $\wedge$ s[0].num == s[1].num
Exceptions: None

isValidStraight(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := s.length == 5 $\wedge$ ($\forall$ i : $\mathbf{N}$| i $\in$ [0..3] : NumSort(s)[i+1].num == NumSort(s)[i].num + 1)
Exceptions: None

isValidFlush(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := s.length == 5 $\wedge$ ($\forall$ i : $\mathbf{N}$| i $\in$ [1..4] : s[i].suite == s[0].suite)
Exceptions: None

isValidFullHouse(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := s.length == 5 $\wedge$ (NumSort(s)[0] == NumSort(s)[1] $\wedge$ NumSort(s)[3] == NumSort(s)[4] $\wedge$ (NumSort(s)[2] == NumSort(s)[1] $\vee$ NumSort(s)[2] == NumSort(s)[3]))
Exceptions: None

isValidFourOfaKind(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := s.length == 5 $\wedge$ ((NumSort(s)[0] == NumSort(s)[1] $\wedge$ NumSort(s)[0] == NumSort(s)[2] $\wedge$ NumSort(s)[0] == NumSort(s)[3]) $\vee$ (NumSort(s)[4] == NumSort(s)[1] $\wedge$ NumSort(s)[4] == NumSort(s)[2] $\wedge$ NumSort(s)[4] == NumSort(s)[3]))
Exceptions: None

isStrongerPlay(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : $\neg$ (s.length == last.length) $\Rightarrow$ false
s.length == 1 $\wedge$ isStrongerSingle(s[0], last[0]) $\Rightarrow$ true
s.length == 2 $\wedge$ isStrongerPair(s, last) $\Rightarrow$ true
s.length == 5 $\wedge$ isStrongerFive(s, last) $\Rightarrow$ true
$\neg$ (s.length $\in$ [1, 2, 5]) $\Rightarrow$ false
Exceptions: None

isStrongerSingle(s):
Input: A Card selected by the current player.
Transition: None
Output : out := s.num > last[0].num $\vee$ (s.suite > last[0].suite $\wedge$ s.num == last[0].num)
Exceptions: None

isStrongerPair(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := isValidPair(s) $\wedge$ s[0].num ¿ last[0].num $\vee$ (SuiteSort(s)[1].suite > SuiteSort(last)[1].suite $\wedge$ s[0].num == last[0].num) Exceptions: None

isStrongerFive(s):
Input: A list of Cards selected by the current player.
Transition: None
Output : out := isValidFourOfaKind(s) $\wedge$ isValidFullhouse(last) $\Rightarrow$ true
isValidFourOfaKind(s) $\wedge$ isValidFlush(last) $\Rightarrow$ true
isValidFourOfaKind(s) $\wedge$ isValidStraight(last) $\Rightarrow$ true
isValidFullHouse(s) $\wedge$ isValidStraight(last) $\Rightarrow$ true
isValidFullHouse(s) $\wedge$ isValidFlush(last) $\Rightarrow$ true
isValidFullFlush(s) $\wedge$ isValidStraight(last) $\Rightarrow$ true
isValidStraight(s) $\wedge$ isValidStraight(last) $\wedge$ NumSort(s)[4].num > NumSort(last)[4].num $\Rightarrow$ true

isValidFlush(s) ∧ isValidFlush(last) ∧ s[0].suite > last[0].suite ⇒ true
isValidFullHouse(s) ∧ isValidFullHouse(last) ∧ (NumSort(s)[3].num > NumSort(last)[3].num) ⇒ true
isValidFourOfaKind(s) ∧ isValidFourOfaKind(last) ∧ (NumSort(s)[4].num > NumSort(last)[4].num ∨ NumSort(s)[0].num > NumSort(last)[0].num) ⇒ true
Exceptions: None

playCards(s):
Input: A list of Cards selected by the current player.
Transition : isStrongerPlay(s) == ture ⇒ last := s
Output: None
Exceptions: None

7 MIS of Game Module

7.1 Interface Syntax

7.1.1 Exported Types

Game = ?

7.1.2 Exported Access Programs

Name	In	Out	Exceptions
Game			
resetGame			
playerPlayCards	sequence of Card		
AIplayCards			
getCardsforTurn	sequence of Card		
updateNextTurnCards	sequence of Card		
updateField	sequence of Card		
updateNextTurn			
playPassTurn			
numberSort			
suitSort			
isGameOver			
displayPassTurn			

7.2 Interface Semantics

7.2.1 State Variables

playerCards: sequence of Cards
opponentLeftCards: sequence of Cards
opponentTopCards: sequence of Cards
opponentRightCards: sequence of Cards
playerField: sequence of Cards
opponentLeftField: sequence of Cards
opponentTopField:sequence of Cards
pponentRightField: sequence of Cards
startingTurn: boolean
turn: String
cardsPlayed: sequence of Cards
lastMove: sequence of Cards
lastMovePlayer: sequence of Cards
freeMove: boolean
gameOver: boolean

7.2.2 Environmental Variables

7.2.3 Assumptions

The constructor of Game is called for each instance before any access routine is called for that object. The constructor cannot be called on an existing object.

7.2.4 Access Program Semantics

Game():
Input: None.
Transition: Initialize the state variables for object Game:

- playerCards, opponentLeftCards, opponentTopCards, opponentRightCards := []
- playerField, opponentLeftField, opponentTopField, opponentRightField := []
- startingTurn:= true
- turn:= null
- cardsPlayed:= []
- lastMove:= []
- gameOver:= false

- freeMove:= false
- lastMovePlayer:= null

Output: None
Exceptions: None

resetGame()
Input: None
Transition: Resets the game to their initial states. Set seach player's deck to the randomly generated sequence of card.
Output: None
Exceptions: None

playerPlayCards(cards)
Input: None
Transition: playerFieldText:= "". $\neg validPlay \Rightarrow$ playerFieldText = "starting turn must be valid and contain 3 of diamonds"
Output: None
Exceptions: None

AIplayCards()
Input: None
Transition: Computes playableCards and update next turn.
Output: None
Exceptions: None

getCardsforTurn()
Input: None
Transition: None
Output: out := (($turn \equiv$ "$opponentLeft$" $\Rightarrow$ opponentLeftCards) $\cup$ ($turn \equiv$ "$opponentTop$" $\Rightarrow$ opponentTopCards) $\cup$ ($turn \equiv$ "$opponentRight$" $\Rightarrow$ opponentRightCards) $\cup$ ($turn \equiv$ "$player$" $\Rightarrow$ playerCards))
Exceptions: None

updateNextTurnCards(cards)
Input: cards: Sequence of cards.
Transition:

- cardsPlayed := cardsPlayed
- lastMove := cards
- lastMovePlayer := turn

- freeMove :=

Output: None
Exceptions: None

updateField(cards)
Input: cards: Sequence of cards.
Transition: $(turn \equiv "opponentLeft" \Rightarrow$ opponentLeftField := cards$) \cup (turn \equiv "opponentTop" \Rightarrow$ opponentTopField := cards$) \cup (turn \equiv "opponentRight" \Rightarrow$ opponentRightField := cards$)$
$\cup$ $(turn \equiv "player" \Rightarrow$ playerField := cards$)$

Output: None
Exceptions: None

updateNextTurn()
Input: None
Transition: $(turn \equiv "opponentLeft" \Rightarrow$ turn := "player"$) \cup (turn \equiv "opponentTop" \Rightarrow$ turn := "opponentLeft"$) \cup (turn \equiv "opponentRight" \Rightarrow$ turn := "opponentRight"$) \cup$
$(turn \equiv "player" \Rightarrow$ turn := "opponentRight"$)$
Output: None
Exceptions: None

playerPassTurn()
Input: None
Transition: $(startingTurn \Rightarrow$ playerFieldText := "You cannot pass the first turn"$) \cup$
$(\neg startingTurn \Rightarrow$ playerFieldText := ""$)$
Output: None
Exceptions: None

numberSort()
Input: None
Transition: playerCards := playerCards.sortCardsValue()
Output: None
Exceptions: None

suitSort()
Input: None
Transition:playerCards := playerCards.sortCardsSuit()
Output: None
Exceptions: None

isGameOver()
Input: None

Transition: ($len(currentPlayerCards) \equiv 0 \Rightarrow$ gameOver := true
Output: out := ($len(currentPlayerCards) \equiv 0 \Rightarrow$ true)
Exceptions: None

displayPassTurn()
Input: None
Transition: Display a text to the user to indicate pass turn.
Output: None
Exceptions: None

8 MIS of gameplayField Module

8.1 Uses

Game

8.2 Interface Syntax

8.2.1 Exported Types

8.2.2 Exported Access Programs

Name	In	Out	Exceptions
render		HTML Card	

8.3 Interface Semantics

8.3.1 State Variables

8.3.2 Environmental Variables

None

8.3.3 Assumptions

8.3.4 Access Program Semantics

render()
Input: None
Transition: None
Output: Arrangement of players in gameplay field.
Exceptions: None