Assignment 3, Part 1, Specification

SFWR ENG 2AA4

March 3, 2019

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the state of a game of Forty Thieves solitaire.

[The parts that you need to fill in are marked by comments, like this one. In several of the modules local functions are specified. You can use these local functions to complete the missing specifications. —SS]

[As you edit the tex source, please leave the `wss` comments in the file. Put your answer **before** the comment. This will make grading easier. —SS]

Card Types Module

Module

CardTypes

Uses

N/A

Syntax

Exported Constants

TOTAL_CARDS = 104
ACE = 1
JACK = 11
QUEEN = 12
KING = 13

Exported Types

SuitT = {Heart, Diamond, Club, Spade}
RankT = [1..13]
CategoryT = {Tableau, Foundation, Deck, Waste}
CardT = tuple of (s: SuitT, r: RankT)

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Generic Stack Module

Generic Template Module

Stack(T)

Uses

N/A

Syntax

Exported Types

Stack(T) = ? [What should be written here? —SS]

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
new Stack	seq of T	Stack	none
push	T	Stack	none
pop		Stack	out_of_range
top		T	out_of_range
size		$\mathbb{N}$	
toSeq		seq of T	

Semantics

State Variables

S: sequence of T[What is the type of the state variable? —SS]

State Invariant

None

Assumptions & Design Decisions

- The Stack(T) constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- Though the toSeq() method violates the essential property of the stack object, since this could be achieved by calling top and pop many times, this method is provided as a convenience to the client. In fact, it increases the property of separation of concerns since this means that the client does not have to worry about details of building their own sequence from the sequence of pops.

Access Routine Semantics

new Stack(s):

- transition: $S := s$
- output: $out := self$
- exception: none

push(e):

- output: $out :=$ new Stack($S \mid\mid \langle e \rangle$)
- exception: none

pop():

- output: $S := S - S[|S| - 1]$ [What should go here? —SS]
- exception: ($|S| = 0 \Rightarrow$ out_of_range) [What should go here? —SS]

top():

- output: $out := S[|S| - 1]$
- exception: ($|S| = 0 \Rightarrow$ out_of_range) [What should go here? —SS]

size():

- output: $out := |S|$ [What should go here? —SS]
- exception: None

toSeq():

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

CardStack Module

Template Module

CardStackT is Stack(cardT)[What should go here? —SS]

Game Board ADT Module

Template Module

BoardT

Uses

CardTypes
CardStack

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new BoardT	seq of CardT	BoardT	invalid_argument
is_valid_tab_mv	CategoryT, $\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	out_of_range
is_valid_waste_mv	CategoryT, $\mathbb{N}$	$\mathbb{B}$	invalid_argument, out_of_range
is_valid_deck_mv		$\mathbb{B}$	
tab_mv	CategoryT, $\mathbb{N}$, $\mathbb{N}$		invalid_argument
waste_mv	CategoryT, $\mathbb{N}$		invalid_argument
deck_mv			invalid_argument
get_tab	$\mathbb{N}$	CardStackT	out_of_range
get_foundation	$\mathbb{N}$	CardStackT	out_of_range
get_deck		CardStackT	
get_waste		CardStackT	
valid_mv_exists		$\mathbb{B}$	
is_win_state		$\mathbb{B}$	

Semantics

State Variables

T: SeqCrdStckT # *Tableau*
F: SeqCrdStckT # *Foundation*
D: CardStackT # *Deck*
W: CardStackT # *Waste*

State Invariant

$|T| = 10$[What goes here? $---SS$]
$|F| = 8$[What goes here? $---SS$]
cnt_cards(T, F, D, W, $\lambda t \rightarrow True$ [What goes here? —SS]) = TOTAL_CARDS
two_decks(T, F, D, W) # *each card appears twice in the combined deck*

Assumptions & Design Decisions

- The BoardT constructor is called before any other access routine is called on that instance. Once a BoardT has been created, the constructor will not be called on it again.
- The Foundation stacks must start with an ace, but any Foundation stack can start with any suit. Once an Ace of that suit is placed there, this Foundation stack becomes that type of stack and only those type of cards can be placed there.
- Once a card has been moved to a Foundation stack, it cannot be moved again.
- For better scalability, this module is specified as an Abstract Data Type (ADT) instead of an Abstract Object. This would allow multiple games to be created and tracked at once by a client.
- The getter function is provided, though violating the property of being essential, to give a would-be view function easy access to the state of the game. This ensures that the model is able to be easily integrated with a game system in the future. Although outside of the scope of this assignment, the view function could be part of a Model View Controller design pattern implementation (`https://blog.codinghorror.com/understanding-model-view-controller/`)
- A function will be available to create a double deck of cards that consists of a random permutation of two regular decks of cards (TOTAL_CARDS cards total). This double deck of cards can be used to build the game board.

Access Routine Semantics

GameBoard(*deck*):

- transition:

  $T, F, D, W :=$ tab_deck($deck$[0..39]), init_seq(8), CardStackT($deck$[40..103]), CardStackT($\langle\rangle$)

- exception: $exc :=$ ($\neg$two_decks(init_seq(10), init_seq(8), CardStackT($deck$), CardStackT($\langle\rangle$)) $\Rightarrow$ invalid_argument) is_valid_tab_mv(c, n_0, n_1):

- output:

	out :=
c = Tableau	valid_tab_tab(n_0,n_1)
c = Foundation	valid_tab_foundation(n_0,n_1)
c = Deck	False [What goes here? —SS]
c = Waste	False [What goes here? —SS]

- exception:

	exc :=
$c = \text{Tableau} \wedge \neg(\text{is\_valid\_pos}(\text{Tableau}, n_0) \wedge \text{is\_valid\_pos}(\text{Tableau}, n_1))$	out_of_range
$c = \text{Foundation} \wedge \neg(\text{is\_valid\_pos}(\text{Tableau}, n_0) \wedge \text{is\_valid\_pos}(\text{Foundation}, n_1))$	out_of_range

is_valid_waste_mv(c, n):

- output:

	out :=
c = Tableau	valid_waste_tab(n)
c = Foundation	valid_waste_foundation(n)
c = Deck	False[What goes here? —SS]
c = Waste	False [What goes here? —SS]

- exception:

	exc :=
W.size() = 0	invalid_argument
$c = \text{Tableau} \wedge \neg\text{is\_valid\_pos}(\text{Tableau}, n)$	out_of_range
$c = \text{Foundation} \wedge \neg\text{is\_valid\_pos}(\text{Foundation}, n)$	out_of_range

is_valid_deck_mv():

- output:

	out :=
c = Tableau	False
c = Foundation	False
c = Deck	False
c = Waste	$\|D\| > 0$

[What goes here? The deck moves involves moving a card from the deck stack to the waste stack. —SS]

- exception: None

tab_mv(c, n_0, n_1):

- transition:

c = Tableau	T[n_0], T[n_1] := T[n_0].pop(),T[n_1].push(T[n_0].top()) [What goes here? —SS]
c = Foundation	T[n_0], F[n_1] := T[n_0].pop(),F[n_1].push(T[n_0].top())[What goes here? —SS]

- exception: $exc := (\neg \text{is\_valid\_tab\_mv}(c, n_0, n_1) \Rightarrow \text{invalid\_argument})$

waste_mv(c, n):

- transition:

c = Tableau	W, T[n] := W.pop(),T[n].push(W.top())[What goes here? —SS]
c = Foundation	W, F[n] := W.pop(),F[n].push(W.top())[What goes here? —SS]

- exception: $exc := (\neg \text{is\_valid\_waste\_mv}(c, n) \Rightarrow \text{invalid\_argument})$

deck_mv():

- transition: D, W := D.pop(),W.push(D.top())[What goes here? —SS]
- exception: $exc := (\neg \text{is\_valid\_deck\_mv}() \Rightarrow \text{invalid\_argument})$

get_tab(i):

- output: $out := T[i]$
- exception: $exc : (\neg \text{is\_valid\_pos}(\text{Tableau}, i) \Rightarrow \text{out\_of\_range})$

get_foundation(i):

- output: $out := F[i]$
- exception: $exc : (\neg \text{is\_valid\_pos}(\text{Foundation}, i) \Rightarrow \text{out\_of\_range})$

get_deck():

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

get_waste():

- output: $out := W$

- exception: None

valid_mv_exists():

- output: $out :=$ valid_tab_mv $\vee$ valid_waste_mv $\vee$ is_valid_deck_mv() where

  valid_tab_mv $\equiv (\exists c : \text{CategoryT}, n_0 : \mathbb{N}, n_1 : \mathbb{N} |$is_valid_pos($Tableau$,n_0)$\wedge$is_valid_pos($c$,n_1) [What goes here? $---SS$] : is_valid_tab_mv$(c, n_0, n_1))$

  valid_waste_mv $\equiv (\exists c : \text{CategoryT}, n : \mathbb{N} |is_valid_pos(c, n)$[What goes here? $---SS$] : is_valid_waste_mv$(c, n))$

- exception: None

is_win_state():

- output: $cnt\_cards\_seq(F, \lambda t \rightarrow \text{True}) = \text{TOTAL\_CARDS}$ [What goes here? —SS]

- exception: None

Local Types

SeqCrdStckT = seq of CardStackT

Local Functions

two_decks : SeqCrdStckT $\times$ SeqCrdStckT $\times$ CardStackT $\times$ CardStackT $\rightarrow \mathbb{B}$
two_decks$(T, F, D, W) \equiv$ [This function returns True if there is two of each card in the game —SS]

$(\forall st : \text{SuitT}, rk : \text{RankT} | st \in \text{SuitT} \wedge rk \in \text{RankT} :$
$(\exists a, b : CardT | a, b \in (T[0..9].toSeq() || F[0..7].toSeq() || D.toSeq() || W.toSeq()) : a.st = b.st \wedge a.rk = b.rk)$[What goes here? $---SS$])

cnt_cards_seq : SeqCrdStckT $\times$ (CardT $\rightarrow \mathbb{B}$) $\rightarrow \mathbb{N}$
cnt_cards_seq$(S, f) \equiv (+s : \text{CardStackT} | s \in S :$ cnt_cards_stack$(s, f))$

cnt_cards_stack : CardStackT $\times$ (CardT $\rightarrow \mathbb{B}$) $\rightarrow \mathbb{N}$
cnt_cards_stack$(S, f) \equiv (+s : \text{CardT} | s \in S.toSeq() \wedge f(s) : 1)$
[What goes here? —SS]

cnt_cards : SeqCrdStckT $\times$ SeqCrdStckT $\times$ CardStackT $\times$ CardStackT $\times$ (CardT $\rightarrow \mathbb{B}$) $\rightarrow \mathbb{N}$
cnt_cards$(T, F, D, W, f) \equiv$ cnt_cards_seq(T, f)+cnt_cards_seq(F, f)+cnt_cards_stack(D, f)+ cnt_cards_stack(W, f)

init_seq : $\mathbb{N} \rightarrow$ SeqCrdStckT
init_seq$(n) \equiv s$ such that $(|s| = n \wedge (\forall\, i \in [0..n-1] : s[i] =$ CardStackT$(\langle\rangle))$

tab_deck : (seq of CardT) $\rightarrow$ SeqCrdStckT
tab_deck$(deck) \equiv T$ such that $(\forall i : \mathbb{N} | i \in [0..9] : T[i]$.toSeq() $= deck[i * 4 ... 4 * (i+1) - 1$[What goes here? $---SS$]])

is_valid_pos: CategoryT $\times \mathbb{N} \rightarrow \mathbb{B}$
is_valid_pos$(c, n) \equiv (c =$ Tableau $\Rightarrow n \in [0..9] | c =$ Foundation $\Rightarrow n \in [0..7] |$True $\Rightarrow$ True)

valid_tab_tab: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
valid_tab_tab $(n_0, n_1) \equiv$

$T[n_0]$.size() > 0	T$[n_1]$.size() > 0	T$[n_0]$.top().s $=$ T$[n_1]$.top().s $\wedge$ T$[n_0]$.top().r $=$ T$[n_1]$.top().r- 1 [What goes here? —SS]
	T$[n_1]$.size() $= 0$	True[What goes here? —SS]
T$[n_0]$.size() $= 0$	T$[n_1]$.size() > 0	False[What goes here? —SS]
	T$[n_1]$.size() $= 0$	False[What goes here? —SS]

valid_tab_foundation: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
valid_tab_foundation$(n_0, n_1) \equiv$

$T[n_0]$.size() > 0	F$[n_1]$.size() > 0	T$[n_0]$.top().s $=$ T$[n_1]$.top().s $\wedge$ T$[n_0]$.top().r $=$ T$[n_1]$.top().r-1
	F$n_1]$.size() $= 0$	T$[n_0]$.top().r = ACE
T$[n_0]$.size() $= 0$	F$[n_1]$.size() > 0	False
	F$[n_1]$.size() $= 0$	False

[What goes here? You may need a table? —SS]

valid_waste_tab: $\mathbb{N} \rightarrow \mathbb{B}$
valid_waste_tab $(n) \equiv$

T$[n]$.size() > 0	tab_placeable(W.top(), T$[n]$.top())
T$[n]$.size() $= 0$	True

valid_waste_foundation: $\mathbb{N} \rightarrow \mathbb{B}$
valid_waste_foundation $(n) \equiv$

F[n].size() > 0	foundation_placeable(W.top(), F[n].top())
F[n].size() $= 0$	W.top().r = ACE

tab_placeable: CardT $\times$ CardT $\rightarrow \mathbb{B}$
tab_placeable$(a, b) \equiv a.s = b.s \wedge a.r = b.r - 1 \Rightarrow$ True
[Complete this specification —SS]

foundation_placeable: CardT $\times$ CardT $\rightarrow \mathbb{B}$
foundation_placeable$(a, b) \equiv a.s = b.s \wedge a.r = b.r - 1 \Rightarrow$ True
[Complete this specification —SS]

Critique of Design

The interface for the modules has rigour and formality. It uses language from discrete math, which has predefined symbols formal syntax, and precise semantics, removing all ambiguity. As well, arguments are checked to see if they are valid, and if not, an exception is called. Modules exercise a proper amount of separation of concerns, with high cohesion and low coupling. All modules are components of a game, so they are closely related. However, modules only call upon each other when necessary. For example, the StackT object does not call upon any other modules , but only provides the necessary functions for itself. The StackT module exhibits generality, as the T value can be of any type. As a result, StackT can be used to represent stacks of other types. Each function only performs one task, so it is clear how each function could be, or will be used. For example, there is a function "is_valid_tab_mv," which checks to make sure the cardT values are correct, and then a "is_valid_pos" function which checks to make sure the arguments are valid, and then a "tab_mv" function which actually moves the values.

A possible improvement would be making the "valid_tab_tab" and other valid functions more general, so that the functions could be reused for other types of solitare . Another related improvement would be the sizes for the tableau and foundation could be a exported constant, so that the number of tableaus and foundation could be changed. Possible considerations would be limiting the size, or setting a constant for a maximum size of the stack of tableaus, as when the game is actually implemented, the tableau will show an image of the cards, and the number of cards in a tableau stack shouldn't be able to extend out of the game window. This will also limit ways the user could use the applications, and prevent possible bugs or errors

[Write a critique of the interface for the modules in this project. Is there anything missing? Is there anything you would consider changing? Why? —SS]