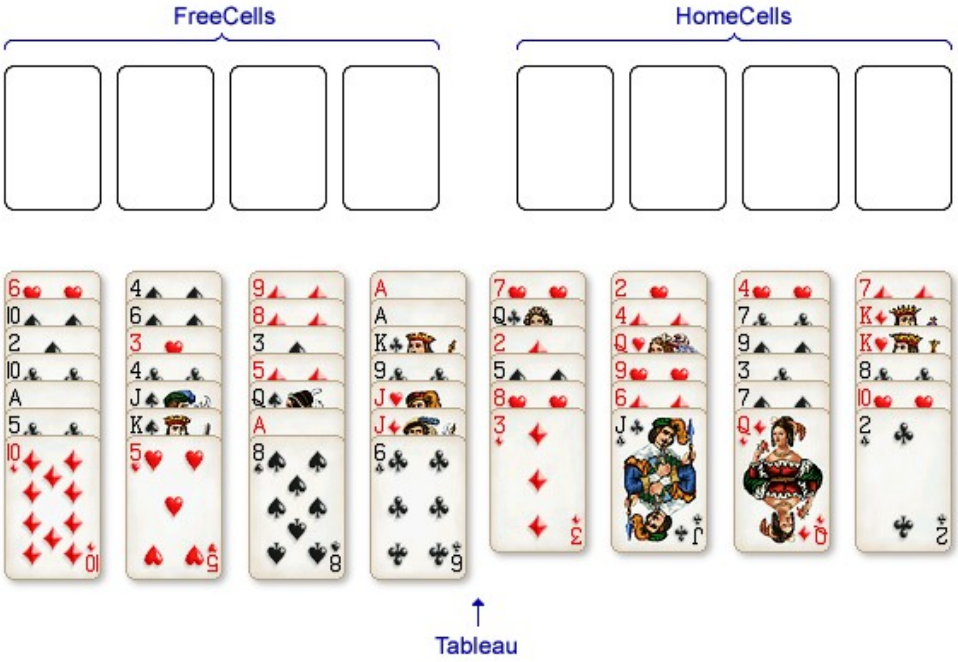
Assignment 4 Specification

SFWR ENG 2AA4

April 9, 2018

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game state of a game of FreeCell solitaire. The game involves a standard 52-card deck of cards shuffled into 8 piles called Tableau cascades, four having 7 cards each and four having 6 cards each at the beginning. The player has 4 FreeCells where they may choose to place any card as part of their moves. Tableaux can be built by placing alternately-coloured cards of descending suits on top of one another. The goal is to fill all four HomeCells, each stacked from Ace to King of the four different suits. Throughout this document, each of these will be referred to as a different type of "cell", following naming conventions from the following gameboard visualization from http://www.solitairecity.com/Help/FreeCell.shtml:



Card Types Module

Module

CardTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

SuitT = {H, D, C, S}
RankT = {A, C2, C3, C4, C5, C6, C7, C8, C9, C10, J, Q, K}
ColourT = {Red, Black}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game Board Types Module

Module

GameBoardTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

CellT = {FreeCell, HomeCell, Tableau}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Card ADT Module

Template Module

CardT

Uses

CardTypes for SuitT, RankT, ColourT

Syntax

Exported Types

CardT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
CardT	SuitT, RankT	CardT	
getSuit		SuitT	
getColour		ColourT	
getRank		RankT	

Semantics

State Variables

suit: SuitT
rank: RankT

State Invariant

None

Assumptions

The constructor CardT 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.

Access Routine Semantics

CardT(s, r):

- transition: $suit, rank := s, r$
- output: $out := self$
- exception: None

getSuit():

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

getColour():

- output: $out := (suit = H \vee suit = D \Rightarrow \text{Red} \mid suit = C \vee suit = S \Rightarrow \text{Black})$
- exception: None

getRank():

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

Generic Stack Module

Generic Template Module

Stack(T)

Uses

N/A

Syntax

Exported Types

Stack(T) = ?

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
Stack	$\mathbb{N}$, seq of T	Stack	invalid_argument
push	T	Stack	stack_full
pop		Stack	stack_empty
top		T	stack_empty
getSize		$\mathbb{N}$	
getMaxSize		$\mathbb{N}$	
toSeq		seq of T	

Semantics

State Variables

S: seq of T
max_size: $\mathbb{N}$

State Invariant

$|S| \leq max\_size$

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

Stack(sze, s):

- transition: $S, max\_size := s, sze$
- output: $out := self$
- exception: $exc := (sze = 0 \Rightarrow \text{invalid\_argument} \mid |s| > sze \Rightarrow \text{invalid\_argument})$

push(e):

- output: $out := \text{Stack}(max\_size, S \mid\mid \langle e \rangle)$
- exception: $exc := (|S| = max\_size \Rightarrow \text{stack\_full})$

pop():

- output: $out := \text{Stack}(max\_size, S[0..|S| - 2])$
- exception: $exc := (|S| = 0 \Rightarrow \text{stack\_empty})$

top():

- output: $out := S[|S| - 1]$
- exception: $exc := (|S| = 0 \Rightarrow \text{stack\_empty})$

getSize():

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

getMaxSize():

- output: $out := max\_size$
- exception: None

toSeq():

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

CardStack Module

Template Module

CardStackT is Stack(CardT)

Game Board ADT Module

Template Module

GameBoardT

Uses

CardTypes for SuitT, RankT and ColourT
GameBoardTypes for CellT
CardADT for CardT
CardStack for CardStackT

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
GameBoardT		GameBoardT	
GameBoardT	seq of CardT	GameBoardT	invalid_argument
place	CellT, $\mathbb{N}$, CardT		invalid_config, invaid_cell
isValidMove	CellT, $\mathbb{N}$, CellT, $\mathbb{N}$	$\mathbb{B}$	invalid_cell
makeMove	CellT, $\mathbb{N}$, CellT, $\mathbb{N}$		invalid_cell, invalid_move
getCell	CellT, $\mathbb{N}$	CardStackT	invalid_cell
validMoveExists		$\mathbb{B}$	
isInWinState		$\mathbb{B}$	

Semantics

State Variables

T: seq of CardStackT
F: seq of CardStackT
H: seq of CardStackT

State Invariant

$|T| = 8$
$|F| = 4$
$|H| = 4$

Assumptions & Design Decisions

- Either of the GameBoardT constructors is called before any other access routine is called on that instance. Once a GameBoardT has been created, the constructor will not be called on it again.
- place() is only called before the game is started, as a way to build the game state. It will not be used during the game by the client code. It is up to the client to build the game state in a way that makes sense, i.e. all cards being present and no duplicate cards, like the real game of FreeCell Solitaire. The RandomDeck module provides functions for creating a valid shuffled deck of cards.
- The main stacks on the board are referred to as the Tableau stacks. The freecells are called FreeCell and the cells where the player must stack all the cards in order to win the game are called HomeCells.
- As per FreeCell rules, HomeCells must start with an ace, but any HomeCell can start with any suit. Once an Ace of that suit is placed there, this HomeCell becomes that type of stack and only those type of cards can be placed there.
- Once a card has been move to a HomeCell, moving it back to the board is a pointless move. For simplicity and to help a would-be player using this model, it is considered invalid by the current model.
- 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 getCell() 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.

Access Routine Semantics

GameBoard():

- transition: $T, F, H := \text{stackSeq}(8, 19), \text{stackSeq}(4, 1), \text{stackSeq}(4, 13)$
- exception: None

GameBoard(*deck*):

- transition: $T, F, H := \text{tableauDeck}(deck), \text{stackSeq}(4, 1), \text{stackSeq}(4, 13)$

- exception: $exc := (|deck| \neq 52 \Rightarrow \text{invalid\_argument})$

place($cell, i, card$):

- transition:

$cell$ = Tableau	T[i] := T[i].push($card$)
$cell$ = FreeCell	F[i] := F[i].push($card$)
$cell$ = HomeCell	H[i] := H[i].push($card$)

- exception:

	$exc :=$
$cell$ = Tableau	$(((0 \leq i < 8) \wedge (\text{T}[i].\text{getSize}() = \text{T}[i].\text{getMaxSize}())) \Rightarrow \text{invalid\_config} \mid (i \geq 8) \Rightarrow \text{invalid\_cell})$
$cell$ = FreeCell	$(((0 \leq i < 4) \wedge (\text{F}[i].\text{getSize}() = \text{F}[i].\text{getMaxSize}())) \Rightarrow \text{invalid\_config} \mid (i \geq 4) \Rightarrow \text{invalid\_cell})$
$cell$ = HomeCell	$(((0 \leq i < 4) \wedge (\text{H}[i].\text{getSize}() = \text{H}[i].\text{getMaxSize}())) \Rightarrow \text{invalid\_config} \mid (i \geq 4) \Rightarrow \text{invalid\_cell})$

isValidMove($c0, n0, c1, n1$):

- output:

		$out :=$
$c0$ = Tableau	$c1$ = Tableau	validTabTab($n0$,$n1$)
	$c1$ = FreeCell	validTabFree($n0$,$n1$)
	$c1$ = HomeCell	validTabHome($n0$,$n1$)
$c0$ = FreeCell	$c1$ = Tableau	validFreeTab($n0$,$n1$)
	$c1$ = FreeCell	validFreeFree($n0$,$n1$)
	$c1$ = HomeCell	validFreeHome($n0$,$n1$)
$c0$ = HomeCell	$c1$ = Tableau	false
	$c1$ = FreeCell	false
	$c1$ = HomeCell	false

- exception: $exc := ((\neg\text{isValidCell}(c0, n0) \vee \neg\text{isValidCell}(c1, n1)) \Rightarrow \text{invalid\_cell})$

makeMove($c0, n0, c1, n1$):

- transition:

$c0$ = Tableau	$c1$ = Tableau	T[$n0$], T[$n1$] := T[$n0$].pop(), T[$n1$].push(T[$n0$].top())
	$c1$ = FreeCell	T[$n0$], F[$n1$] := T[$n0$].pop(), F[$n1$].push(T[$n0$].top())
	$c1$ = HomeCell	T[$n0$], H[$n1$] := T[$n0$].pop(), H[$n1$].push(T[$n0$].top())
$c0$ = FreeCell	$c1$ = Tableau	F[$n0$], T[$n1$] := F[$n0$].pop(), T[$n1$].push(F[$n0$].top())
	$c1$ = FreeCell	F[$n0$], F[$n1$] := F[$n0$].pop(), F[$n1$].push(F[$n0$].top())
	$c1$ = HomeCell	F[$n0$], H[$n1$] := F[$n0$].pop(), H[$n1$].push(F[$n0$].top())

- exception: exc := (($\neg$ isValidCell($c0, n0$) $\vee$ $\neg$isValidCell($c1, n1$)) $\Rightarrow$ invalid_cell | ($\neg$ isValidMove($c0, n0, c1, n1$)) $\Rightarrow$ invalid_move)

getCell($cell, n$):

- output:

	$out :=$
$cell =$ Tableau	$T[n]$
$cell =$ FreeCell	$F[n]$
$cell =$ HomeCell	$H[n]$

- exception: $exc :=$ (($\neg$ isValidCell($cell, n$)) $\Rightarrow$ invalid_cell)

validMoveExists():

- output: $out := (\exists\, c0, c1 : \text{CellT}, i, j : \mathbb{N} : \text{isValidMove}(c0, i, c1, j))$
- exception: None

isInWinState():

- output: $out :=$
  isWinningHomeCell(0) $\wedge$
  isWinningHomeCell(1) $\wedge$
  isWinningHomeCell(2) $\wedge$
  isWinningHomeCell(3)
- exception: None

Local Functions

stackSeq: $\mathbb{N} \times \mathbb{N} \rightarrow$ (seq of CardStackT)
stackSeq $(l, n) = s$ such that $(|s| = l \wedge (\forall\, i \in [0..l-1] : s[i] = \text{CardStackT}(n, \langle\rangle))$

tableauDeck: (seq of CardT) $\rightarrow$ (seq of CardStackT)
tableauDeck $(deck) =$
$\langle \text{CardStackT}(deck[0..6], 19) \rangle ||$
$\langle \text{CardStackT}(deck[7..13], 19) \rangle ||$
$\langle \text{CardStackT}(deck[14..20], 19) \rangle ||$
$\langle \text{CardStackT}(deck[21..27], 19) \rangle ||$
$\langle \text{CardStackT}(deck[28..33], 18) \rangle ||$
$\langle \text{CardStackT}(deck[34..39], 18) \rangle ||$

$\langle$CardStackT($deck[40..45], 18$)$\rangle$||
$\langle$CardStackT($deck[46..51], 18$)$\rangle$

isValidCell: CellT $\times \mathbb{N} \to \mathbb{B}$
isValidCell $(c, n) \equiv$

$c =$ Tableau	$0 \leq n < 8$
$c =$ FreeCell	$0 \leq n < 4$
$c =$ HomeCell	$0 \leq n < 4$

validTabTab: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validTabTab $(t0, t1) \equiv$

$T[t0].getSize() > 0$	T[$t1$].getSize() > 0	tabPlaceable(T[$t0$].top(), T[$t1$].top())
	T[$t1$].getSize() $= 0$	true
T[$t0$].getSize() $= 0$	T[$t1$].getSize() > 0	false
	T[$t1$].getSize() $= 0$	false

validTabFree: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validTabFree $(t, f) \equiv$ T[t].getSize() $> 0 \wedge$ F[f].getSize() $= 0$

validTabHome: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validTabHome $(t, h) \equiv$

T[t].getSize() > 0	H[h].getSize() > 0	homePlaceable(T[t].top(), H[h].top())
	H[h].getSize() $= 0$	T[t].top().getSuit() $=$ A
T[t].getSize() $= 0$	H[h].getSize() > 0	false
	H[h].getSize() $= 0$	false

validFreeTab: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validFreeTab $(f, t) \equiv$

F[f].getSize() > 0	T[t].getSize() > 0	tabPlaceable(F[f].top(), T[t].top())
	T[t].getSize() $= 0$	true
F[f].getSize() $= 0$	T[t].getSize() > 0	false
	T[t].getSize() $= 0$	false

validFreeFree: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validFreeFree $(f0, f1) \equiv$ F[$f0$].getSize() $> 0 \wedge$ F[$f1$].getSize() $= 0$

validFreeHome: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validFreeHome $(f, h) \equiv$

F[f].getSize() > 0	H[h].getSize() > 0	homePlaceable(F[f].top(), H[h].top())
	H[h].getSize() $= 0$	F[f].top().getRank() = A
F[f].getSize() $= 0$	H[h].getSize() > 0	false
	H[h].getSize() $= 0$	false

oneLess: CardT $\times$ CardT $\to \mathbb{B}$
oneLess $(c0, c1) \equiv$
let sts = [A, C2, C3, C4, C5, C6, C7, C8, C9, C10, J, Q, K]
in $(\exists\, i \in [0..|sts| - 2] : sts[i] = c0.getRank() \wedge sts[i + 1] = c1.getRank())$

oneMore: CardT $\times$ CardT $\to \mathbb{B}$
oneMore $(c0, c1) \equiv$
let sts = [A, C2, C3, C4, C5, C6, C7, C8, C9, C10, J, Q, K]
in $(\exists\, i \in [0..|sts| - 2] : sts[i] = c1.getRank() \wedge sts[i + 1] = c0.getRank())$

tabPlaceable: CardT $\times$ CardT $\to \mathbb{B}$
tabPlaceable $(c0, c1) \equiv$

c0.getColour() $\neq$ c1.getColour()	oneLess(c0, c1)
c0.getColour() = c1.getColour()	false

homePlaceable: CardT $\times$ CardT $\to \mathbb{B}$
homePlaceable $(c0, c1) \equiv$

c0.getSuit() = c1.getSuit()	oneMore(c0, c1)
c0.getSuit() $\neq$ c1.getSuit()	false

isWinningHomeCell: $\mathbb{N} \to \mathbb{B}$
isWinningHomeCell $(i) \equiv$
let $stSeq = H[i]$.toSeq() in
$stSeq[0].getRank() = \text{A} \wedge |stSeq| = 13 \wedge (\forall\, i \in [0..|stSeq| - 2] : \text{oneLess}(stSeq[i], stSeq[i + 1]))$

Card Deck Utils Module

Module

CardDeckUtils

Uses

CardTypes for SuitT, RankT and ColourT
CardADT for CardT

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
CardDeckUtils_randomDeck		seq of CardT	

Semantics

State Variables

State Invariant

Assumptions

Access Routine Semantics

CardDeckUtils_randomDeck():

- output: *out* := permute(deckSet())
- exception: None

Local Functions

deckSet: (set of CardT)
deckSet() = $\{s : \text{SuitT}, r : \text{RankT} : \text{CardT}(s, r)\}$

permute(*inSet*): set of CardT $\rightarrow$ seq of CardT
permute = S such that S is a random permutation of *inSet*, satisfying
$(\forall\, c \in S : c \in inSet) \wedge (\forall\, c \in inSet : c \in S) \wedge |S| = |inSet| \wedge (\forall\, i, j : \mathbb{N} | i \neq j : S[i] \neq S[j])$