2ME3 Assignment 4

Justin Staples (staplejw)

April 5, 2017

Contents

The purpose of this software design exercise to design and specify a set of modules that are capable of storing the state of a typical game of battleship. In terms of the MVC design pattern, these modules represent the model. They store the current state of the game. They are able to keep track of the state of the board, and whose turn it is.

This design has three modules. The first is a basic abstract data type, called CellT, that represents one single cell of the game board. It stores information about whether or not that cell has a ship on it and also whether or not the cell has been the target of a shot.

The second module, BoardT, is another abstract data type that represents the state of the game for 1 player. It contains a two dimensional sequence of cells, as well a few other state variables that describe the turn status, or which shots have been fired.

The third module, called Battleship, stores the entire game state. In contains two state variables, one board for each of the players, and a routine to initialize the boards. Each board can be accessed individually to place ships and fire shots.

The interface presented is believed to be minimal, meaning that all access routines presented provide just one service. As well, the interface is essential in the sense that all of the access routines are necessary and provide a unique functionality. There are a few routines that perform similar functionality, but the rationale for why they are kept is provided in the comments section of the MIS. Lastly, the interface is general. It is general because not only does it allow for the standard game of battleship to be played, it also allows variants of the game to be played. Such variants include games where the dimensions of the board are variable, where the number of ships can be up to 5, not exactly 5, and where the dimensions of the ships are also adjustable and not limited to sizes between 2 and 5.

1 Cell ADT Module

Template Module

CellT

Uses

N/A

Syntax

Exported Types

CellT = ?

Exported Constants

N/A

Exported Access Programs

Routine name	In	Out	Exceptions
CellT		CellT	
getShipID		integer	
getShot		boolean	
setShipID	integer		InvalidShipIdException
setShot	boolean		

Semantics

State Variables

$shipID$: integer
$shot$: boolean

State Invariant

none

Assumptions

The constructor CellT is called before any other access routine is. The constructor cannot be called on an existing object.

Access Routine Semantics

CellT():

- transition: $shipID, shot := -1, false$
- output: $out := self$
- exception : none

getShipID():

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

getShot():

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

setShipID(i):

- transition: $shipID := i$
- exception:

$$\neg(0 \leq i \leq 4) \Rightarrow \text{InvalidShipIdException}$$

setShot(b):

- transition: $shot := b$
- exception: none

2 Board ADT Module

Template Module

BoardT

Uses

CellT

Syntax

Exported Types

BoardT = ?

Exported Constants

MAX_SHIPS = 5

Exported Access Programs

Routine name	In	Out	Exceptions
BoardT	integer, integer, boolean	BoardT	InvalidSizeException
getCells		sequence [][] of CellT	
placeShip	integer, integer, integer, integer		OutOfBoundsException InvalidShipException
isMyTurn		boolean	
changeTurn			
fireShot	integer, integer, BoardT		OutOfBoundsException InvalidShotException WrongPlayerException
pastShot	integer	tuple of (i, j : integer)	OutOfBoundsException
didItHit	integer, BoardT	boolean	OutOfBoundsException
isLoser		boolean	
progress	BoardT	real	

Semantics

State Variables

r : integer
c : integer
s : sequence of [sequence of CellT]
$myTurn$: boolean
$myShots$: sequence of [tuple of (i, j : integer)]
$shipCounter$: integer

State Invariant

$shipCounter \leq$ MAX_SHIPS

Assumptions

The constructor BoardT is called before any other access routine is. The constructor cannot be called on an existing object.

Access Routine Semantics

BoardT($rows$, $columns$, $turn$):

- transition: $r, c, s, myTurn, myShots, shipCounter := rows, columns,\ sequence\ [rows][columns]\ of\ CellT, turn, <>, 0$
- output: $out := self$
- exception :

$$\neg(r > 0 \wedge c > 0) \Rightarrow \text{InvalidSizeException}$$

getCells(): *// getter method for two dimensional sequence of cells. by indexing into this, any cell of the game board can be examined*

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

placeShip(i, j, k, l): *// i and j represent the row and column indices of the start of the ship and k and l are the indices of the end of the ship. i must be a smaller index than k, j must be a smaller index than l*

- transition: $shipCounter := shipCounter + 1$and s such that

$$\forall(a, b : \mathbb{N} | i \leq a \leq k \wedge j \leq b \leq l : s[a][b].\text{setShipID(ShipCounter)})$$

// for all cells on the board that are covered by the ship, give those cells a unique shipID, then increment the ship counter

- exception:

$$\neg(0 \leq i < r \wedge 0 \leq k < r \wedge 0 \leq j < c \wedge 0 \leq l < c) \Rightarrow \text{OutOfBoundsException}$$

$$i > k \vee j > l \Rightarrow \text{InvalidShipException}$$

$$\neg(k - i = 0 \vee l - j = 0) \Rightarrow \text{InvalidShipException}$$

$$shipCounter = 5 \Rightarrow \text{InvalidShipException}$$

$$\exists(a, b : \mathbb{N} | i \leq a \leq k \wedge j \leq b \leq l : s[a][b].\text{getShipID}() \geq 0) \Rightarrow \text{InvalidShipException}$$

// out of bounds exception if the indices are not in the range of the board dimensions. there are four ways to throw an InvalidShipException. if i is bigger than k or j is bigger than l, if the ship is placed on a diagonal, if the board already 5 valid ships, or if any of the spots that the ship occupies already have a ship on them

isMyTurn():

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

changeTurn(): *// isMyTurn and changeTurn are kept as separate methods in the event that the client wants to check the turn status without changing it. they are also kept separate to avoid having access programs that change state and return a value*

- transition: $myTurn := \neg myTurn$
- exception: none

fireShot(i, j, b):

- transition: $myShots, myTurn := myShots \mid\mid < (i, j) >, false$
  as well as b.getCells[i][j].setShot(true) and b.changeTurn

- exception:

$$\neg(0 \leq i < r \wedge 0 \leq j < c) \Rightarrow \text{OutOfBoundsException}$$

$$b.\text{getCells[i][j].getShot()} \Rightarrow \text{InvalidShotException}$$

$$b.\text{isMyTurn} \Rightarrow \text{WrongPlayerException}$$

$$\neg myTurn \Rightarrow \text{WrongPlayerException}$$

pastShot(k): *// used to examine the coordinates of the shot at index k in the myShots sequence. pastShot and didItHit have similar functionality, but I decided to keep them separate in the event that the player would want to check if they made a shot without necessarily checking to see if it hit. As well, both of these methods return a different type of value. One returns a tuple of integers while the other returns a boolean. I thought it would feel very clunky to have a method that returns a tuple of these two types*

- output: $out := myShots[k]$
- exception:

$$\neg(0 \leq k < |myShots|) \Rightarrow \text{OutOfBoundsException}$$

didItHit(k, b): *// did the shot k hit the board b? the tuple from myShots is used to examine that cell of the board. if its shipID is non-negative, then that spot has a ship and so it was a hit. the method returns true if it was a hit*

- output: $out := b.\text{getCells[myShots[k].i][myShots[k].j].getShipID} \geq 0$
- exception:

$$\neg(0 \leq k < |myShots|) \Rightarrow \text{OutOfBoundsException}$$

isLoser(): *// checks the board to see if all of the cells that have a ship on them have been shot at*

- output: $out := all\_ships\_sunk(self)$
- exception: none

progress(b): *// to give an idea of how the player is doing, this method returns a percentage out of a 100. it is the ratio of ship cells that they have hit to the total ship cells on the enemy board*

- output: $out := 100 \cdot \frac{\text{ship\_cells\_hit(b)}}{\text{total\_ship\_cells(b)}}$
- exception: none

Local Functions

all_ships_sunk: BoardT $\rightarrow$ boolean
all_ships_sunk(s) $\equiv \forall(i, j : \mathbb{N} | b.\text{getCells}[i][j].\text{getShipID}() \geq 0 : b.\text{getCells}[i][j].\text{getShot}())$
// for all cells on the board where there is a ship, that cell has been shot

ship_cells_hit: BoardT $\rightarrow$ integer
ship_cells_hit(b) $= +(i, j : \mathbb{N} | b.\text{getCells}[i][j].\text{getShot}() \wedge b.\text{getCells}[i][j].\text{getShipID}() \geq 0 : 1)$
// count the number of cells on the board that have been hit and have a ship

total_ship_cells: BoardT $\rightarrow$ integer
total_ship_cells(b) $= +(i, j : \mathbb{N} | b.\text{getCells}[i][j].\text{getShipID}() \geq 0 : 1)$
// count the number of cells on the board that have a ship

3 Battleship Module

Module

Battleship

Uses

BoardT

Syntax

Exported Constants

N/A

Exported Types

N/A

Exported Access Programs

Routine name	In	Out	Exceptions
init	integer, integer		InvalidSizeException
player1		BoardT	
player2		BoardT	

Semantics

State Variables

$p1$: boardT
$p2$: boardT

State Invariant

N/A

Assumptions

The init method is called for the abstract object before either of the game boards can be accessed. The init method can be used to return the state of the game to the state of a new game.

Both players should place all of their ships before the first shot is fired and should not place anymore ships after the first shot is fired.

Access Routine Semantics

init(r, c): *// initialize two battleship boards called player 1 and player 2. the boards are given dimensions based on user input. the boards are initialized so that it is player 1's turn to start*

- transition: $p1, p2 := \text{BoardT}(r, c, true), \text{BoardT}(r, c, false)$
- exception :

$$\neg(r > 0 \wedge c > 0) \Rightarrow \text{InvalidSizeException}$$

player1():

- output: $out := p1$
- exception : none

player2():

- output: $out := p2$
- exception : none