Assignment 4, Specification

SFWR ENG 2AA4

April 13, 2019

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

Gameboard Types Module

Module

GameboardTypes

Uses

N/A

Syntax

Exported Constants

MAX_ROWS = 20
MAX_COLS = 20

Exported Types

CellT = {Dead, Alive}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game Board ADT Module

Template Module

boardUniverse

Uses

GameboardTypes
ReadWrite

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new boardUniverse		boardUniverse	out_of_range
overSize	$\mathbb{Z}$, $\mathbb{Z}$	$\mathbb{B}$	
rowInBound	$\mathbb{Z}$	$\mathbb{B}$	
colInBound	$\mathbb{Z}$	$\mathbb{B}$	
getRows		$\mathbb{Z}$	
getCols		$\mathbb{Z}$	
cellState	$\mathbb{Z}$, $\mathbb{Z}$	CellT	out_of_range
numAliveNeighbors	$\mathbb{Z}$, $\mathbb{Z}$	$\mathbb{Z}$	
NextUniverse			
gameOver		$\mathbb{B}$	

Semantics

State Variables

U: Universe # *Gameboard of cells*
C: CellT # *Cells in the gamboard*

State Invariant

$|rows| <= MAX\_ROWS$
$|cols| <= MAX\_COLS$

$|rows| = |cols|$

Assumptions & Design Decisions

- The boardUniverse constructor is called before any other access routine is called on that instance. Once a boardUniverse has been created, the constructor will not be called on it again.
- The gameboard will always be a square with equal rows and columns.
- The gameboard will use fixed edges and will not wrap around to the other end of the board. Therefore any cells rotating of the board will not be shown as per the real simulation of the game.
- 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.

Access Routine Semantics

boardUniverse():

- transition:

  $Universe, cols, rows :=$ initUniverse(), Universe.size(), Universe.size()

- exception: $exc :=$ overSize($rows, cols$) $\Rightarrow$ out_of_range

overSize($rows, cols$):

- output: out := (rows $< MAX\_ROWS \wedge$ cols $< MAX\_COLS$)
- exception: None

rowInBound(row):

- output: out := (rows $< row \vee$ row < 0)
- exception: None

colInBound(col):

- output: out := (cols $< col \vee$ col < 0)
- exception: None

getRows():

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

getCols():

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

getUniverse():

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

cellState(row, col):

- output: out := state such that Universe$[row][col] = Alive \Rightarrow State = Alive|$ Universe$[row][col] = Dead \Rightarrow State = Dead$
- exception: $\neg(rowInBound(row) \wedge colInBound(col)) \Rightarrow out\_of\_range$

numAliveNeighbors(row, col):

- output: $out := +(\forall i : \mathbb{Z} | i \in [-1..1] \cdot \forall j : \mathbb{Z} | j \in [-1..1] \cdot ((i \text{ != } 0 \vee j \text{ != } 0) \wedge (rowInBound(row + i) \wedge colInBound(col + j)) \wedge (cellState(i + row, j + col) = Alive)) : 1)$

- exception: None

NextUniverse():

- $transition : newUniverse := (\forall\, i : \mathbb{Z} | i \in [0..rows - 1] \cdot \forall\, j : \mathbb{Z} | j \in [0..cols - 1] \cdot (numAliveNeighbors(i, j) < 2 \wedge numAliveNeighbors > 3) \Rightarrow newUniverse[i][j] = Dead | (cellState(i, j) = Dead \wedge numAliveNeighbors = 3) \Rightarrow newUniverse[i][j] = Alive | newUniverse[i][j] = Universe[i][j])$
- $transition : Universe := newUniverse$
- exception: None

gameOver():

- output: $out := +(\forall\, i : \mathbb{Z} | i \in [0..rows - 1] \cdot \forall\, j : \mathbb{Z} | j \in [0..cols - 1] \cdot \text{Universe}[i][j] = Alive \Rightarrow false) | true$
- exception: None

Display Module

Module

DisplayGen

Uses

GameboardTypes

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
showUniverse	seq of seq of CellT		

Semantics

Environment Variables

None

State Variables

None

State Invariant

None

Access Routine Semantics

showUniverse(Universe)

- transition: Iterate through the 2D vector Universe, and print the current state of the cells to the console, with one space between each cell and a newline when the row ends. The concole will print "*" if the cell at row i and column j is alive or else print "-" if the cell is dead.

Read Write Module

Module

ReadWrite

Uses

GameboardTypes

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
readFile	String	seq of char	runtime_error
writeFile	String, seq of seq of CellT		runtime_error

Semantics

Environment Variables

Initial_Universe: File containing initial gameboard state
Output_Universe: File containing output of desired gameboard state

State Variables

None

State Invariant

None

Assumptions

The input file will match the given specification.

Access Routine Semantics

readFile(file)

- transition: read data from the a input file associated with the string s. Use this data to save a sequence of cahrs representing states of the gameboard.

  The text file has the following format, where " − ", and " ∗ " stands for strings that represent the state of the cells, CellT, in the gameboard, Dead and Alive, respectively. All data values in a row are separated by one space. Rows are separated by a new line. The data shown below is for a total of m rows with m columns. The rows and columns have to be equal as the gameboard is an n by n universe.

$$\begin{array}{cccccc} - & * & * & - & - & ... \\ - & * & - & * & - & ... \\ - & * & * & - & * & ... \\ ..., & ..., & ..., & ..., & ..., & ... \end{array} \tag{1}$$

- exception: exc : := File not found ⇒ runtime_error

writeFile(file, $vector < vector < CellT >>$)

- transition: write data from the seq of seq CellT to a file associated with the string s. Output state of the game to file. where " − ", and " ∗ " stands for strings that represent the state of the cells, CellT, in the gameboard, Dead and Alive, respectively. Rows are seperated by a new line and columns by one space. Size of columns and rows are the same, representing a n by n gamboard. The text file written has the following format.

$$\begin{array}{cccccc} - & * & * & - & - & ... \\ - & * & - & * & - & ... \\ - & * & * & - & * & ... \\ ..., & ..., & ..., & ..., & ..., & ... \end{array} \tag{2}$$

- exception: exc : := File not found ⇒ runtime_error

Critique of Design

- The modules in the design of this project include the gameboard module, display/view module, the ReadWrite module, and the gameboardTypes module. The gameboard module is responsible for the state of the gameboard and the status of the game. The ReadWrite module reads the state of the gameboard from a text file and outputs the state of the game to a text file. The display/view module prints the state of the gameboard to the console. The gameboardTypes module defines the state of the cells on the grid of the gameboard and defines the state invariants of size of the gameboard. This desgin supports high cohesion as the module gameboard groups methods that all contribute to a single well-defined task of the module which is to change/affect the state of the gameboard and cells of each row and column throught the NextUniverse() method. The methods of numAliveNeighbors, cellState, initUniverse, rowInBound, and colInBound all contribute to the single task of NextUniverse() method to get the next state of the gameboard. High cohesion reults in low coupling as modules do not rely on each other, and are independent of each other, except the gameboard module branches to the ReadWrite module to use one method to read from a text file and initialize the gameboard. The modules also support minimality, as no one method accomplishes two tasks. For example, the method rowInBound is only responsible for checking if the row falls within the gameboard bounds, or the NextUniverse() method which only changes the state of the gameboard, but calls other methods to accomplish the task, such as numAliveNeighbors to calculate Alive neighbors of a cell; it does not do this task within the NextUniverse method itself. Many other methods in the modules in this design follow this principle of minimality. This design and modules also support essentiality as no method in modules can perform the task of another method without violating minimality, therefore each method is essential to fulfil the goal of the game. For example, the rowInBound and colInBound can be combined in one method as having both of them is not essential, but doing that will violate minimality. This design also supports information hiding as the software module hides information by encapsulating the information into a module which presents the interface. This means the gameboard module supports information hiding as other parts of the program will be protected from extensive modification to gameboard if the design decision is changed. Consistency is used in this design in terms of the GUI. A consistent look and feel of using ASCII graphics to represent the state of the game makes it easier for users to examine the gameboard for Alive and Dead cells and the position of the cells. Generality is difficult to design in this design, as the game does not have many components to it, except a simple grid with rows and columns. There isn't a lot going on in the grid except the change it states

of the cells, therefore the methods are specific in their tasks, and generality is not really needed for this design. Overall, this design supports many of the software engineering principles discussed above.