Assignment 4, Specification

Seda Mete

April 9, 2019

This MIS document contains modules, types, and methods for implementing the state of a game of Conway's Game of Life. The modules cover the Model and View portions of the Model View Controller design pattern. The GameBoard module stores the state of the game and the View module can display the state of the game board using ASCII graphics.

Game Board Types Module

Module

GameBoardTypes

Uses

N/A

Syntax

Exported Constants

SIZE = 20 *//size of the board in each direction*

Exported Types

cellT = { DEAD, ALIVE } *//the two states a cell can be*

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game Board Module

Module

GameBoard

Uses

GameBoardTypes

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
init	string		invalid_file
next			
getb	$\mathbb{Z}, \mathbb{Z}$	cellT	out_of_bounds

Semantics

Environment Variables

initial_state: File containing initial game state

State Variables

b: boardT

State Invariant

$|b| = 20$

Assumptions

- The input file will match the given specification.
- The init method is called for the abstract object before any other access routine is called for that object.

Access Routine Semantics

init(s):

- transition: Read data from the file initial_state associated with the string s. Use this data to initialize the state of the game board b.
  The text file has the following format. Each line represents a row in the game board. There are 20 lines, each containing 20 numbers (0 or 1) separated by a comma. Each comma-separated number represents a cell in the game board. The value of the number represents the state of that cell. A 0 represents a DEAD cell and a 1 represents an ALIVE cell.
- exception: $exc :=$ file named s doesn't exist $\Rightarrow$ invalid_file

next():

- transition: $+(\forall\ i, j : \mathbb{Z} \mid (0 \leq i < \text{SIZE}) \wedge (0 \leq j < \text{SIZE}) : (b[i,j] = \text{DEAD} \wedge \text{liveNeighbours} = 3) \Rightarrow temp[i,j] := \text{ALIVE} \mid (b[i,j] = \text{DEAD} \wedge \text{liveNeighbours} \neq 3) \Rightarrow temp[i,j] := \text{DEAD} \mid (b[i,j] = \text{ALIVE} \wedge (\text{liveNeighbours} \neq 2 \vee \text{liveNeighbours} \neq 3)) \Rightarrow temp[i,j] := \text{DEAD} \mid (b[i,j] = \text{ALIVE} \wedge (\text{liveNeighbours} = 2 \vee \text{liveNeighbours} = 3)) \Rightarrow temp[i,j] := \text{ALIVE})$, where temp is a new temporary sequence [SIZE, SIZE] of cellT
  $b := temp$
- exception: none

getb(i, j):

- output: $out := b[i,j]$
- exception: $exc :=$ ($\neg$validPosition(i, j) $\Rightarrow$ out_of_bounds)

Local Types

boardT = sequence [SIZE, SIZE] of cellT

Local Functions

liveNeighbours: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
liveNeighbours$(i, j) \equiv \ +(m, n : \mathbb{Z} \mid : i - 1 \leq m \leq i + 1 \wedge j - 1 \leq n \leq i + 1 \wedge (m \neq i \vee n \neq j) \wedge \text{validPosition}(m, n) \wedge b[m, n] = \text{ALIVE} : 1)$

validPosition: $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
validPosition$(i, j) \equiv (0 \leq i < \text{SIZE} \wedge 0 \leq j < \text{SIZE})$

View Module

Module

View

Uses

GameBoardTypes
GameBoard

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
view	GameBoard		
save	GameBoard, string		

Semantics

Environment Variables

console: Represents the console that text will be outputted to
outFile: Represents the output file that the save method creates

State Variables

None

State Invariant

None

Assumptions

- A GameBoard object has been created and the init method has been called for it before any of these access routines are called.
- A valid output file name is given for the save function.

Access Routine Semantics

view($game$):

- transition: console := ($\forall\ i : \mathbb{Z} \mid 0 \leq i < \text{SIZE} : (\forall\ j : \mathbb{Z} \mid 0 \leq j < \text{SIZE} :$ $game$.getb$(i, j) = \text{DEAD} \Rightarrow$ "| "| $game$.getb$(i, j) = \text{ALIVE} \Rightarrow$ "|O")"| $\backslash n$")
- exception: none

save($game, s$):

- transition: Creates a file outFile with the name s.
  outFile := ($\forall\ i : \mathbb{Z} \mid 0 \leq i < \text{SIZE} : (\forall\ j : \mathbb{Z} \mid 0 \leq j < \text{SIZE} :$ $game$.getb$(i, j) =$ $\text{DEAD} \Rightarrow 0 +$ ",| $game$.getb$(i, j) = \text{ALIVE} \Rightarrow 1 +$ ",")"$\backslash n$")
- exception: none

Critique of Design

Firstly, my design follows the principle of consistency because the coding style and variable naming is consistent, which makes the code easier to read. For example, a string argument is always named s, a GameBoard argument is always named $game$, and the boardT indices are always referred to using i and j. In addition, referring to the size of the game board using the exported constant SIZE, instead of the number 20 also makes the design more consistent. If someone ever wanted to change the design so it has a larger game board, they would only have to change one line of code (the #define SIZE 20 line) instead of having to change every place where the game board size is used.

In regards to cohesion, creating a GameBoard module and a View module instead of just one module is an example of how my design follows this principle. The functions in each module are now more closely related to one another than if they were all put in one module. The functions in the GameBoard module are all directly related to the game board itself, like initializing it, transitioning to the next state, and getting one of its cell values. The View module's functions are different from GameBoard's functions because they are solely for "viewing" the module. Both of View's functions output the game state either to the console or a text file. Since my design follows the MVC design pattern, it is more cohesive.

In regards to information hiding, making the variable b, which holds the state of each cell in the game board, private follows this principle. The contents of b can't be tampered with and can only be accessed through using the class' public functions.

A way my design follow the principle of generality is that it can take any input file (that still follows the proper format). Initially, I was going to make the init function not take a file name as an argument. It was just going to automatically read from the file input.txt. Enabling the init function to take any file makes the design more general because the initial state of the game board can now be anything, it is not predefined. Furthermore, my design follows the essentiality and minimality qualities as it only contains the functions that it needs to work properly. There are no additional uneeded or unused functions. The minimal functions that it does have are essential for the program to work as intended.