Assignment 4, Part 1, Specification

SFWR ENG 2AA4

April 14, 2019

Cell Types Module

Module

CellTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

LifeT = $\{$ALIVE, DEAD$\}$

Exported Access Programs

None

Cell Module

Template Module

Cell

Uses

CellTypes

Syntax

Exported Types

Cell = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Cell		Cell	
new Cell	LifeT, $\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$	Cell	invalid_argument
get_cnt		$\mathbb{N}$	None
get_Alive		LifeT	None
get_x		$\mathbb{N}$	None
get_y		$\mathbb{N}$	None

Semantics

State Variables

S: LifeT
cnt: $\mathbb{N}$
x: $\mathbb{N}$
y: $\mathbb{N}$

State Invariants

None

Assumptions and Design Decisions

- The Cell constructor is called before any instance of the class. Constructor can only be called once.
- No setters are provided. Cells are immutable by design.

Access Routine Specifics

new Cell():

- transition: $S, cnt, x, y :=$DEAD, 0, 0, 0
- output: $out := self$
- exception: none

new Cell(c, a, b, c):

- transition: $S, cnt, x, y :=$c, a, b, c
- output: $out := self$
- exception: $exc := (a < 1 || b < 1 || c < 1) \implies invalid\_argument$

get_cnt():

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

get_x():

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

get_y():

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

Game Module

Template Module

Game

Uses

Celltypes, Cell

Syntax

Exported Types

Game = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Game		Game	
new Game	seq of (seq of Cell)	Game	invalid_argument
set_cell	Cell		none
get_cell	$\mathbb{N}$, $\mathbb{N}$	Cell	invalid_argument
get_rows		$\mathbb{N}$	None
get_cols		$\mathbb{N}$	None
get_matrix		seq of (seq of Cell)	None
update_game			None
print_state			None

Semantics

Environment Variables

d: An environment variable corresponding to a monitor representation of the standard output of the C++ compiler.

State Variables

S: Seq of (Seq of Cell)
$rows$: $\mathbb{N}$
$cols$: $\mathbb{N}$

State Invariants

None

Assumptions and Design Decisions

- The Game constructor is called before any instance of the class. Constructor can only be called once.
- Default constructor (with no arguments) initalizes Game with a 15 x 15 matrix full of dead cells with 0 cnt and x and y being their position in the matrix
- Assuming all matrices have uniform row and column sizes
- Both update_game and print_state are state transitions. It was chosen to make update_game a state transition because Conway's Game of Life is a state based game.

Access Routine Specifics

new Game():

- transition: $rows, cols, S$:=15, 15, v such that |v| = 15 $\wedge$ |v[0]| = 15 $\wedge$ $(\forall\, i : \mathbb{N} | i \in [0..14] : (\forall, j : \mathbb{N} | i \in [0..14] : v[i][j].get\_Alive() == DEAD))$ $\wedge$ $(\forall\, i : \mathbb{N} | i \in [0..14] : (\forall, j : \mathbb{N} | i \in [0..14] : v[i][j].get\_x() == i))$ $\wedge$ $(\forall\, i : \mathbb{N} | i \in [0..14] : (\forall, j : \mathbb{N} | i \in [0..14] : v[i][j].get\_y() == j))$ $\wedge$ $(\forall\, i : \mathbb{N} | i \in [0..14] : (\forall, j : \mathbb{N} | i \in [0..14] : v[i][j].get\_cnt() == 0))$
- output: $out := self$
- exception: none

new Game(e):

- transition: $S, cols, rows$:=e, |e|,|e[0]|
- output: $out := self$
- exception: $exc := (|S| < 1 || |S[0] < 1) \implies invalid\_argument$

get_cell(n1, n2):

- output: $out := S[n1][n2]$
- exception: none

set_cell(e):

- transition: $S[e.get\_x()][e.get\_y()] := e$
- exception: none

get_cols():

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

get_row():

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

get_matrix():

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

update_game(e):

- transition: $S := g$ such that $(\forall i : \mathbb{N} | i \in [0..rows] : (\forall, j : \mathbb{N} | i \in [0..cols] : g[i][j] = update\_cell(S[i][j]))$
- exception: none

print_state():

- transition: D := A representation of the seq of (seq cell) matrix of any given Game object, with X being alive cells and 0 being dead cells. This prints the state of the game to the screen.
- exception: none

Local Functions

alive_neighbour_cnt: Cell → Cell
alive_neighbour_cnt(a) ≡ Cell(a.get_Alive(), c, a.get_x(), a.get_y()) where
c = $(+c : Cell | c \in adj \wedge c.get\_alive() = ALIVE)$ where adj = $\{s : Cell | S \in$ S[c.get_x() ± 1][c.get_y()] $\bigcup$ S[c.get_x()][c.get_y() ± 1] $\bigcup$ S[c.get_x() ± 1][c.get_y() ± 1] : s$\}$

update_life : Cell → Cell
update_life(c) ≡

get_neighbour_cnt(c) <2	Cell(DEAD, c.get_cnt(), c.get_x(), c.get_y())
get_neighbour_cnt(c) = 2 ∧ c.get_Alive() = DEAD	Cell(ALIVE, c.get_cnt(), c.get_x(), c.get_y())
get_neighbour_cnt(c) >3	Cell(DEAD, c.get_cnt(), c.get_x(), c.get_y())

update_all_cnt: Seq of (Seq of Cell) → Seq of (Seq of Cell)
update_all_cnt(g)≡ s such that $(\forall i : \mathbb{N} | i \in [0..|g|-1] : (\forall, j : \mathbb{N} | i \in [0..|g[0]|-1] : s[i][j] = update\_cell(g[i][j]))$

IO Module

Module

IO

Uses

CellTypes, Cell, Game

Syntax

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Read	String	Game	
Write	String, Game	d: text file (environment variable)	

Semantics

Environment Variables

input: File containing initial state for game
output: File where game state is to be written to

State Variables

None

State Invariants

None

Assumptions and Design Decisions

- Input will match following description

Access Routine Semantics

read(s)

- read data from the file associated with the string s. This data is then converted into a Game object, with the state of the object determined by the data in the file.

  The text file has the following format. All elements are to be separated with a single space in between. The first line will contain the width and the height respectively. The next lines will be 'X's and 'O's arranged in a matrix. This matrix is a representation of the game state. X is to represent a live cell, and O is to represent a dead cell. The data is shown below:

```
3  3
X  X  X
O  O  O
X  X  X
```

- exception: none

write(g, s)

- writes the current state of the game to the file specified by the string s. This will write a representation of the game state, with X representing live cells and O representing dead ones. The data will be written to the file in the same format as above
- exception: none

Design Critique and Analysis

- Essential - Several getter and setter functions are provided for the Game ADT. While this would normally violate the property of being essential (Only methods required are to display and update Game state), they are required to read and write to files, and need to be called upon in the IO module. Other than those, all methods are essential, and cannot be replaced by a combination of any other methods.
- Information Hiding - Information hiding was done by making non required functions and state variables opaque to the user. This was done by encapsulation: modules were formed that each created ADTs. This allowed each module to develop independent of each others' implementation, since they only interfaced through the exported access programs. For example, if a different implementation was done to update the game, it wouldn't affect the IO module at all. This allows for flexibility in implementation and preserves design decisions.
- Consistent - Consistency was preserved by enforcing naming conventions and parameter orders. Methods were overwhelmingly named with all undercase letters separated by underscores. One instance where this is broken is the name of the Cell method "get_Alive(). This should be corrected in future updates. Parameters were consistently done; for example, the parameters for any function requiring row and column indexes were always in the same order: row first, than column. Local variables were also consistenly done. In the Game module, 'c' usually denotes a cell input, while 'g' usually denotes a Game. This is however broken in the IO module, and should be corrected.
- Generality - For 2D Cellular automata, the solution is quite general. While in this implementation the Moore neighborhood of cells is used, the implementation could easily be changed to look at the von Neumann neighborhood, as the methods that define the neighborhood are implementation specific. Another example would be the rules table. While in this implementation Conway's Game of Life rules are used, any other rules could be implemented without impact to the other modules, or any of the exported access programs.
- Minimality - The design preserves minimality. Each exported access program only did one service. For example, no state changes and outputs were made. Each program either outputted something (e.g get_cell) or changed the state (e.g set_cell).
- Cohesion - Cohesion was preserved by use of encapsulation and interface design. Any changes to the implementation would not change the class at all, provided the interface remained the same. This helped to reduce coupling, and helped increase cohesion. Informational cohesion was also achieved, since all methods encapsulated would work on the same data fields. For example, in the Cell class, all methods would work on Cell

fields (e.g cnt, x, y) However, certain methods (e.g get_matrix) did not contribute to the defined task of the module, weakening cohesion. Worse, these methods were implemented because they were required by other modules, increasing the amount of coupling between them.