Assignment 4, Specification

SFWR ENG 2AA4

April 12, 2021

The following Module Interface Specification document provides the modules, types and methods needed to create the game 2048. Once `make main` is typed into the terminal, a grid of 4-by-4 is printed onto the screen, along with instructions to navigate the game. The grid consists of 2 values, each being either a 2 or a 4, with the remaining 14 tiles initialized as 0. The user can use four keys to indicate the direction in which they would like to move; entering i moves the tiles up, entering k moves the tiles down, entering j moves the tiles left, and entering l moves the tiles right. Once the tiles move at the users command, any tile adjacent to another in that direction is combined with that tile if the two tiles are equal in value, and continue shifting in that direction. The game terminates if a tile on the board has the value of 2048, indicating the user has won. The game also terminates if every tile on the board has a value greater than zero, and no two adjacent tiles have the same value. This indicates that the user has lost. In either case, when the game terminates, the board resets to its initial state. This game uses the `Display` module as the View, the `BoardT` module as the Model, the `Controller` as the Controller, and the `Game` module as a way to use `BoardT` based on user input from `Controller`.

Likely Changes

- The visual representation of the game such as UI layout
- The hardware on which the game is run
- The constraints on the input parameters

Display Module

Module

Display

Uses

None

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
print	seq of seq of $\mathbb{N}$		
introMessage			
intructions			
prompt			
errorMessage			
winMessage			
loseMessage			

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Considerations

Access Routine Semantics

print(*board*):

- Print the board in 4 by 4 grid, incrementing rows from left to right and incrementing columns top bottom. Include sufficient space between each value.

introMessage():

- Print out the message ”Welcome! Can you reach 2048?”

instructions():

- Print out the message ”Use ’i=up’, ’k=down’, ’j=left’, ’l=right’”

promtp():

- Print out the message ”Which direction?”

errorMessage():

- Print out the message ”Sorry! Invalid Input! Please use ’i’, ’j’, ’k’, or ’l’”

winMessage():

- Print out the message ”Congrats, you won!!”

loseMessage():

- Print out the message ”Uh oh! Try again!”

BoardT

Template Module

BoardT

Uses

None

Syntax

Exported Constants

span:$\mathbb{N}$

Exported Types

BoardT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new BoardT			
getGrid		seq of seq of $\mathbb{N}$	
getScore		$\mathbb{N}$	
newGrid			
newNumber			
wonGame		$\mathbb{B}$	
lostGame		$\mathbb{B}$	
up			
down			
left			
right			
move	String		
add	String		

Semantics

State Variables

score: $\mathbb{N}$
grid: seq of seq of $\mathbb{N}$

State Invariant

None

Assumptions

There is access to a function random that returns number between 0 and 1

Access Routine Semantics

new BoardT():

- transition: $grid, score = \langle i : \mathbb{N} | i \in [0..span - 1] : \langle j : \mathbb{N} | j \in [0..span - 1] : 0 \rangle \rangle$, 0,
- output: $out := self$
- exception: none

getGrid():

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

getScore():

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

newGrid( ):

- newNumber()
- newNumber()

newNumber():

- transition:
```
initialize boolean empty = true
while empty = true:
  initialize integer i = random number between 0 and 1 multiplied by 4
  initialize integer j = random number between 0 and 1 multiplied by 4
  initialize double k = random number between 0 and 1
  if grid cell of row i and column j has value of 0:
    if k is less than 0.1:
      grid cell of row i and column j = 4
      empty = false
    else:
      grid cell of row i and column j = 2
      empty = false;
```

wonGame():

- $output : \langle i : \mathbb{N} | i \in [0..span-1] : \langle j : \mathbb{N} | j \in [0..span-1] | grid[i][j] = 2048 \Rightarrow True | True \Rightarrow False \rangle \rangle$,

lostGame():

- transition:
```
for i in grid length:
  for j in grid length:
    if cell at row i column j has a value of 0:
      return false
    if the cell above cell at row i column j has the same value:
      return false
    if the cell below cell at row i column j has the same value:
      return flase
    if the cell to the left of cell at row i column j has the same value:
      return false
    ig the cell to the right of cell at row i column j has the same value:
      return false
return true;
```

up():

- move("i")

- add(”i”)
- move(”i”)

down():

- move(”k”)
- add(”k”)
- move(”k”)

left():

- move(”j”)
- add(”j”)
- move(”j”)

right():

- move(”l”)
- add(”l”)
- move(”l”)

move(direction):

- Move each cell up if direction is equal to ”i”, down if direction is equal to ”k”, left if direction is equal to ”j”, or right if direction is equal to ”l”. it iterates through the cells of the grid by incrementing the row and column values. Rightmost cells cannot move right, the leftmost cells cannot move left, the topmost cells cannot move up and the bottommost cells cannot move down. Cells move over only if the adjacent cell in that direction is equal to 0, to avoid overwriting previously existing values. Each time the cell moves over, its previous position is overwritten to become zero. Each cell must be moved as far is it can be moved in the specified direction.

add(direction):

- If the specified direction is ”i”, iterate through each cell, starting from the second row and the first column, and compare it to the cell above. If their values are equal, overwrite the value of the cell above to be the sum of their values. Add to the score the sum of the values. Overwrite the current cell to become zero.
  If the specified direction is ”k”, begin at the third row and first column, and iterate through the cells, decrementing the row and incrementing the column. Compare each cell to the one below it; if their values are equal, overwrite the value of the cell below to be the sum of their values. Add to the score the sum of the values. Overwrite the value of the current cell to become 0.
  If the specified direction is ”j”, iterate through each cell, starting from the first row and the second column. Increment the row and column each time. Compare the value of each cell to the left adjacent column. If their values are equal, overwrite the value of the left cell to be the sum of their values. Add to the score the sum of the values. overwrite the current cell's value to be 0.
  If the specified direction is ”l” begin at the first row and third column and iterate through each cells, incrementing the row and decrementing the column. Compare the current cell's value to the value of the right adjacent cell's value; if they are equal, overwrite the right adjacent value to become the sum of both values. Add to the score the sum. Overwrite the current cell's value to be 0.

Controller Module

Module

Controller

Uses

Game
Display

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
getUserInput			
runGame			

Semantics

Environment Variables

input:Scanner

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Semantics

getUserInput():

- transition:

```
prompt()
initialize Scanner input = new Scanner
initialize string direction = next line input
if direction $\neq$ "i" and $\neq$ "k" and $\neq$ "j" and $\neq$ "l":
  errorMessage()
else
  whereToGo(direction)
```

runGame():

- transition:

```
introMessage()
Print a new line
instructions();
resetGame();
while true:
  print the game board
  getUserInput
```

Game Module

Module Module

Game

Uses

Controller
BoardT
Display

Syntax

Exported Constants

None

Exported Types

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
getBoard same	 seq of seq of $\mathbb{N}$	BoardT seq of seq of $\mathbb{N}$	
whereToGo	String		
resetGame			
checkGameStatus			

Semantics

State Variables

b : BoardT

State Invariant

None

Assumptions

`resetGame()` is called before any other Access Routine in the module

Access Routine Semantics

getBoard():

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

same($grid$):

- output: $out := \langle i : \mathbb{N} | i \in [0..grid - 1] : \langle j : \mathbb{N} | j \in [0..grid - 1] : grid[i][j] \rangle \rangle$
- exception: none

whereToGo($direction$):

- transition

```
initialize double array b = same($b$)
if direction = "i":
 $b$.up()
else if direction = "k":
  $b$.down()
else if direction = "j":
  $b$.left()
else if direction = "l":
  $b$.right()
initialize double array a = same($b$)
if a $\neq$ b:
  $b$.newNumber()
checkGameStatus
```

- exception: none

resetGame():

- transition: $b := new$BoardT
- b.newGrid()

checkGameStatus():

- transition:
```
if $b$.wonGame():
  winMessage()
  resetGame()
else if $b$.lostGame()
  loseMessage()
  resetGame()
```

Critique

The implementation of my design ensures minimality within each module, in that no two methods accomplish the same computation and outputs. This adds to the quality of essentiality, as every method created within the design is used and necessary for the performance of the game. This was not the case during the entirety of the design process. I had initially created a method `full()` which served to check if every value on the grid was greater than 0. This used a nested for loop to iterate through the rows and columns and a counter, which was incremented for every value in the grid that was greater than 0. If the counter by the end was equal to 16 by the end of the iteration, the method would return a boolean value true, or else it would return false. Soon into creating the `lostGame()` method, however, I realized I could accomplish the same computation in one simple step, and so the `full()` method omitted from the `BoardT` module. In doing so, BoardT has a great quality of minimality and essentiality, by reducing time complexity and an entire method that was not actually necessary.

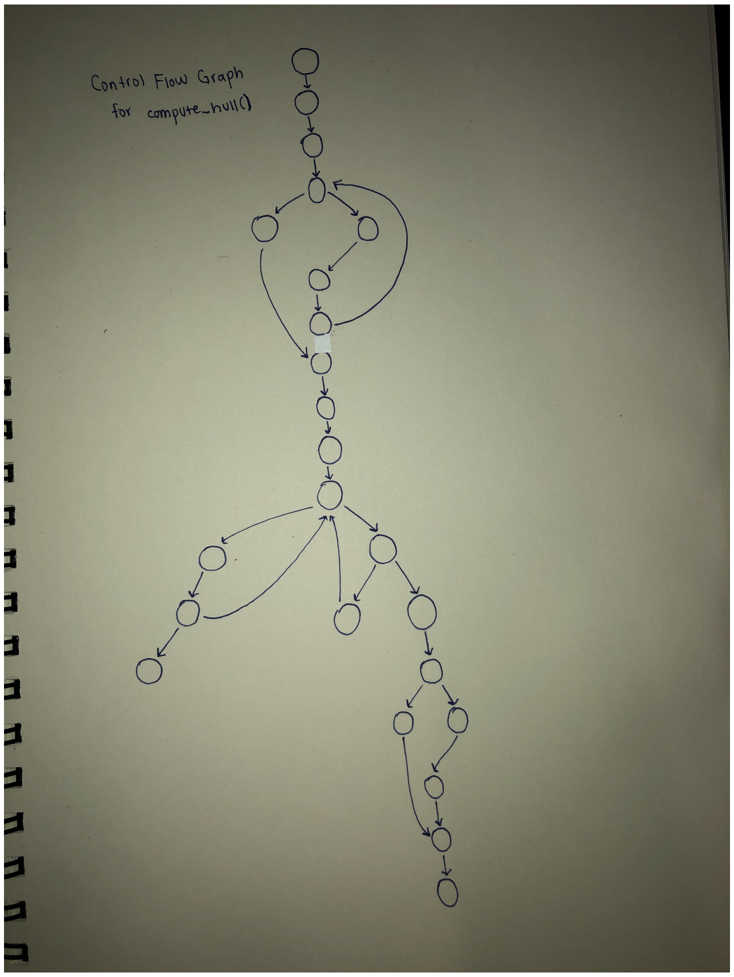
The MIS for each module is general in that it specifies only what needs to be outputted, but is open ended for the developer to decide how they would wish to implement the methods. For a few methods, however, there is no way to make the specification general; the `up()`, `down()`, `left()`, and `right()` methods call on other methods and thus cannot be implemented in more ways than one.

The design is consistent in naming conventions, in that each method with more than one word is named using Camel Case, while one word methods are lowercase. The parameters used for methods are all name consistently; every parameter of type `int[][]` is given the variable name `board`, which is synonymous to `grid` but is used to avoid confusion. Every parameter of type `String` is given the variable name `direction` to make the code easily to follow along with. Iteration through the cells of the grid use the same variables, where `int i` and `int j` represent the row and column of the grid, respectively. The design was created to make errors legible to the user in the form of a displayed message, as opposed to raising an error. If the user input is not equal i, k, j or l, the system displays a message informing them that their input is invalid, and that they should use the appropriate keys to navigate the game. By not raising an error for every invalid key pressed, the game does not halt, which allows the user to continue playing and avoids having to restart every time.

The design additionally has high cohesion, in that the properties of each module as closely related. They work an accomplish specific things, however in order to run the game, including the controller, the model of the board, and the display of the board and prompt, each module works together. In terms of information hiding, methods used by other modules were made public. The instances of the `BoardT` created were made private to avoid user manipulation. This allows whatever happens with the model of the board based on user input to happen behind the scenes.

Question 1

Question 2


Control Flow Graph
for compute_hull()