Assignment 4, Two Dots game Specification

SFWR ENG 2AA4

April 6, 2020

This Module Interface Specification (MIS) document contains interfaces,ADT's and methods for implementing a game of Two Dots. Some modules and notation were borrowed from the A3 specification and all credit goes to it. The specification is generally formal but contains some informal parts for specifying for example; output to a screen, starting a timer and describing a iterative method which has no output. The strategy pattern was used in the Strategy interface and the classes that inherit it for different game modes. The singleton pattern was followed for the Timer module and MVC was used throughout to structure the interface of the application and to seperate concerns. A full MIS of the View and Controller is also provided. Comments are provided in the "Considerations" section at the end of each module where appropriate to clarify any misunderstandings and ambiguities.

Color Module

Module

Color

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Color = {R, G, B, P, Y}

//R stands for Red, G for green, B for blue, P for Purple, Y for yellow

Exported Access Programs

Routine name	In	Out	Exceptions
randomColor		Color	

Semantics

State Variables

colors: Color

State Invariant

None

Access Routine Semantics

randomColor():

- transition: none

- output: out := randomVal()
- exception: none

Local Functions

randomVal(): Color
randomVal() $\equiv (i = 0 \implies \mathrm{R} \mid i = 1 \implies \mathrm{G} \mid i = 2 \implies \mathrm{B} \mid i = 3 \implies \mathrm{P} \mid i = 4 \implies \mathrm{Y}\ )$
Where i is a uniformly-distributed random number in the range $0 \leq i \leq 4$

Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
PointT	$\mathbb{Z}$, $\mathbb{Z}$	PointT	
row		$\mathbb{Z}$	
col		$\mathbb{Z}$	

Semantics

State Variables

r: $\mathbb{Z}$
c: $\mathbb{Z}$

State Invariant

None

Assumptions

The constructor PointT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

PointT(row, col):

- transition: $r := row, c := col$
- output: $out := self$
- exception: None

row():

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

col():

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

Generic Board Module

Generic Template Module

Board(T)

Uses

PointT

Syntax

Exported Types

Board(T) = ?

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Board	$\mathbb{N}, \mathbb{N}$	Board	IllegalArgumentException
set	PointT, T		IndexOutOfBoundsException
get	PointT	T	IndexOutOfBoundsException
getNumRow		$\mathbb{N}$	
getNumCol		$\mathbb{N}$	

Semantics

State Variables

s: seq of (seq of T)
nRow: $\mathbb{N}$
nCol: $\mathbb{N}$

State Invariant

None

Assumptions

- The Board(T) constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- $s[i][j]$ means the ith row and the jth column. The 0th row is at the top of the grid and the 0th column is at the leftmost side of the grid.

Access Routine Semantics

Board(row, col):

- transition (note that the list does not enforce an *order* in which the transitions occur, only the transitions that must occur):
  1. $nRow := row$
  2. $nCol := col$
- output: $out := self$
- exception:
  exc := (row $\leq$ 0) $\vee$ (col $\leq$ 0) $\implies$ IllegalArgumentException

set(p, v):

- transition: $s[p.row()][p.col()] = v$
- exception:
  $\neg$ validPoint(p) $\implies$ IndexOutOfBoundsException

get(p):

- output: $out := s[p.row()][p.col()]$
- exception:
  $\neg$ validPoint(p) $\implies$ IndexOutOfBoundsException

getNumRow():

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

getNumCol():

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

Local Functions

validRow: $\mathbb{N} \rightarrow \mathbb{B}$
$\text{validRow}(r) \equiv r \geq 0 \wedge (r < \text{nRow})$

validCol: $\mathbb{N} \rightarrow \mathbb{B}$
$\text{validCol}(c) \equiv (c \geq 0) \wedge (c < \text{nCol})$

validPoint: PointT $\rightarrow \mathbb{B}$
$\text{validPoint}(p) \equiv \text{validCol(p.col())} \wedge \text{validRow(p.row())}$

BoardMoves Module

Template Module

BoardMoves is seq of PointT

Considerations

When implementing in Java. Extend Arraylist parameterized by the type PointT

TwoDotsBoard Module

Template Module

TwoDotsBoard is Board(Color)

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
validateMoves	BoardMoves	$\mathbb{B}$	
updateBoard	BoardMoves		

Semantics

Access Routine Semantics

validateMoves(b):

- output : out := $|b| > 1 \wedge \forall (p : PointT \mid p \in$b : validPoint(p)) $\wedge\, validPath(b) \wedge isDistinct(b)$
- exception: None

updateBoard(b):

- output : out := None
- transition : s := $\forall (p : PointT | p \in b \forall (i : \mathbb{N} | i \in [p.row()..1]) : s[i][p.col()]$:= randomColor())
- exception: None

Local Functions

validRow: $\mathbb{N} \rightarrow \mathbb{B}$
$\text{validRow}(r) \equiv r \geq 0 \wedge (r < \text{nRow})$

validCol: $\mathbb{N} \rightarrow \mathbb{B}$
$\text{validCol}(c) \equiv (c \geq 0) \wedge (c < \text{nCol})$

validPoint: $\text{PointT} \rightarrow \mathbb{B}$
$\text{validPoint}(p) \equiv \text{validCol(p.col())} \wedge \text{validRow(p.row())}$

isDistinct: $\text{BoardMoves} \rightarrow \mathbb{B}$
$\text{isDistinct}(b) \equiv \forall(i : \mathbb{N} | i \in [0..|b|-1] : \forall(j : \mathbb{N} | j \in [(i+1)..|b|-1]) : \neg(b[i].row() = b[j].row()) \wedge (b[i].col() = b[j].col()))$

validPath: $\text{BoardMoves} \rightarrow \mathbb{B}$
$\text{validPath}(b) \equiv \forall(i : \mathbb{N} | i \in [0..|b|-2] : isAdjacent(b, i, i+1) \wedge sameColor(b, i, i+1))$

sameColor : $\text{BoardMoves} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
$\text{sameColor}(b, i, j) \equiv s[b[i].row()][b[i].col()] = s[b[j].row()][b[j].col()]$

isAdjacent: $\text{BoardMoves} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
$\text{isAdjacent}(b, i, j) \equiv b[i].row() = b[j].row() \wedge b[i].col() = b[j].col() + 1$
$\vee\ b[i].row() = b[j].row() \wedge b[i].col() = b[j].col() - 1$
$\vee\ b[i].row() = b[j].row() - 1 \wedge b[i].col() = b[j].col()$
$\vee\ b[i].row() = b[j].row() + 1 \wedge b[i].col() = b[j].col()$

Considerations

In Java, calling randomColor() represents a call to a static function so it would actually be done as Color.randomColor()

Strategy Interface Module

Interface Module

Strategy

Syntax

Exported Constants

None

Exported types

None

Exported Access Programs

Routine name	In	Out	Exceptions
play	TwoDotsBoard		

BoardView Module

Template Module

BoardView

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
printBoard	TwoDotsBoard	$\mathbb{B}$	
modePrompt		Strategy	
getInput		BoardMoves	
closeStream			
printMsg	$msg : string$		

Semantics

Environment variables

s : 2D sequence of pixels displayed on a standard Unix Shell/console
r : an object to write text out on a standard Unix Shell/console

Access Routine Semantics

printBoard(b):

- transition s := Modify the Console so that the TwoDotsBoard b is printed in a tabular manner. The contents of each row from b should be on individual line. Their should be horizontal and vertical numbering indicating each row and column from one upto and including the row and column size of the board
- exception: None

modePrompt():

- transition :

  - s := Modify the console to print a message asking the user to enter "T" for the timed version of the game and "M" for the mode of the game with a set number of moves
  - r:= read a single line of text from the standard input. Store this value in memory and then determine what to output as follows:
    If the line read in is "T" or "t" output : out := new TimedStrategy()
    If the line read in is "M" or "m" output: out := new MovesStrategy()
    Otherwise keep reading a line from the standard input until one of the above two conditions are met

- exception: None

getInput():

- transition :
  - r:= read a single line of text from the standard input to determine the coordinates of the dots the user would like to eliminate. Note that the desired input format is u,v w,x y,z .... These are pairs of natural numbers with a comma between them and each pair is separated by a space. Store this value in memory and then determine what to output as follows:
    If the line read in is in the correct format then *output* : *out* := new BoardMoves() containing the pairs of integers
    Otherwise keep reading a line from the standard input until one of the above conditions are met

closeStream():

- transition : s:= close the input stream

printMsg(msg : $string$):

- transition : s:= Modify the output console to print out text contain in the string msg

Considerations

In java, closing the input stream corresponds to closing the System.in object

BoardController Module

Template Module

BoardController

Uses

TwoDotsBoard, BoardView, Color, PointT, BoardMoves

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
BoardController	TwoDotsBoard, BoardView	BoardController	
get	PointT	Color	
set	PointT, Color		
validateMoves	BoardMoves	$\mathbb{B}$	
updateBoard	BoardMoves		
updateView			
printMsg	$msg : string$		
modePrompt		Strategy	
closeViewStream			
getInput		BoardMoves	

Semantics

State variables

m : TwoDotsBoard v : BoardView

State invariant

None

Access Routine Semantics

BoardController(*model*,*view*)

- output: out := self
- transition: m := model, v := view
- exceptions: none

get(p)

- output : *out* := m.get(p)
- transition: none
- exceptions: none

set(p,c)

- transition: m.set(p,c)
- exception : none

validateMoves(b)

- output : *out* := m.validateMoves(b)

updateBoard(b)

- transition: m.updateBoard(b)

updateView()

- transition: v.printBoard(m)

printMsg(*msg* : *string*):

- transition view.printMsg(*msg*)

modePrompt():

- output: *out* := v.modePrompt()

closeViewStream():

- transition :v.closeStream()

getInput():

- output: if (m.validateMoves(v.getInput())) then *out* := else *out* := getInput()

StrategyGameMode Module

Template Module inherits Strategy

StrategyGameMode

Uses

Strategy, BoardView, BoardController, BoardMoves, TwoDotsBoard

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
play	TwoDotsBoard		
startUp	TwoDotsBoard		
checkWin		$\mathbb{B}$	
canContinue		$\mathbb{B}$	
updateData			
introMsg			
endMsg			

Semantics

State variables

c : BoardController
v : BoardView
$moves$: BoardMoves

State invariant

None

Access Routine Semantics

play(b)

- transition:
  - startUp(b)
  - introMsg()
  - if canContinue() then:
    * c.updateView()
    * c.updateBoard(c.getInput())
    * if checkWin() then
    * updateData()
    * if canContinue() then repeat these steps labeled with *

Consideration

In Java, this module would be implemented as an abstract class that implements the Strategy interface. Unimplemented methods are ones that are abstract methods and will be overridden by its children
This is the best that could be done to convey the idea of a "abstract class" given that MIS does not have the notion of an abstract class, following Dr Smith's advice to use Inheritance and leave a note for reader. Source: Here (will have to login to avenue)

MovesStrategy Module

Template Module inherits StrategyGameMode

MovesStrategy

Uses

StrategyGameMode, BoardView, BoardController, BoardMoves, TwoDotsBoard

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
play	TwoDotsBoard		
startUp	TwoDotsBoard		
checkWin		$\mathbb{B}$	
canContinue		$\mathbb{B}$	
updateData			
introMsg			
endMsg			

Semantics

Environment variables

State variables

$moveCount : \mathbb{N}$
$TARGET : \mathbb{N}$

State invariant

None

Access Routine Semantics

startUp(b)

- transition: $v, c, moveCount, TARGET :=$ new BoardView(), new BoardController(b, v), 15, 5

checkWin()

- transition: if $|moves| < TARGET$ then $moves := moves - 1$
- outupt: $out :=$ if $|moves| >= TARGET$ then $out := true$ else $out := false$

canContinue()

- output: $out := moveCount > 0$

updateData()

- transition: if $moveCount = 0$ then c.exit()
  else $c :=$ using c.printMsg(), modify the screen to print out how many moves are left, i.e print the value of $moveCount$

introMsg()

- transition: $c :=$ using c.printMsg(), modify the screen to print a message to state the rules. This includes instruction for how to enter input, how to win and how many total moves the user has

endMsg()
transition: $c :=$ using c.printMsg(), modify the screen by printing a message telling the user the game is over

CountDownTimer Module

Module

CountDownTimer

Uses

None

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
newTimer	$\mathbb{Z}$		IllegalArgumentException
isCancelled		$\mathbb{B}$	

Semantics

Environment variables

t : Represents the system clock

State variables

$cancelled : \mathbb{B}$
$multiplier : \mathbb{Z}$

State invariant

None

Access Routine Semantics

newTimer(time : $\mathbb{Z}$)

- transition: $cancelled := false$
  $multiplier := 1000$
  $t :=$ Use the system clock to start tracking the current time. Once time*multiplier amount of time has passed then the transition $cancelled := true$ happens

- exception: $exc :=$ time $< 1 \implies$ IllegalArgumentException

isCancelled()

- output: $out := \neg cancelled$

TimedStrategy Module

Template Module inherits StrategyGameMode

TimedStrategy

Uses

Strategy, BoardController, StrategyGameMode, CountDownTimer

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
play	TwoDotsBoard		
startUp	TwoDotsBoard		
checkWin		$\mathbb{B}$	
canContinue		$\mathbb{B}$	
updateData			
introMsg			
endMsg			

Semantics

Environment variables

t : Represents the system clock

State variables

$TARGET : \mathbb{Z}$
$TIME : \mathbb{Z}$

State invariant

None

Access Routine Semantics

startUp(b)

- transition: $v, c, moveCount, TARGET, TIME :=$ new BoardView(), new BoardController(b, v), 5, 60
  newTimer(TIME)

checkWin()

- output: $out := |moves| >= TARGET$

canContinue()

- output: $out :=$ isCancelled()

updateData()

- transition: $c :=$ using c.printMsg(), modify the screen to print out how much time has elapsed since the start of the game. Formally, print out $t.now() - TIME$

introMsg()

- transition: $c :=$ using c.printMsg(), modify the screen to print a message to state the rules. This includes instruction for how to enter input, how to win and how much time the user has

endMsg()
transition: $c :=$ using c.printMsg(), modify the screen by printing a message telling the user the game is over

Consideration

When calling methods such as newTimer() and isCancelled() these are references to the access programs defined in CountDownTimer module. In java these would static access program and would be accessed without creating an instance of the module. Also assume $t.now()$ gives the current time in seconds

Dots Module

Module

Dots

Uses

Strategy, BoardController, TwoDotsBoard, BoardView

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
main	seq of $string$		

Semantics

Environment variables

None

State variables

None

State invariant

None

Access Routine Semantics

main($args$: seq of $string$):
transition:
new BoardController(new TwoDotsBoard(6,6), new BoardView()).modePrompt().play(new TwoDotsBoard(6,6))

Considerations

The main role of this module, in terms of implementation is to instantiate the controller, start the desired mode the user wants and then play the game while manipulating the board and interacting with the user via the controller.

Discussion Question

Critique of design

1. Consistency:

2. Essentiality: In general I believe the design provided is essential for all the modules. However, essentially becomes more difficult to achieve the more complex a design becomes. To maintain essentiallity in all of the modules I did my best to modularize the design. Splitting different concerns of the game into different modules, ADT's and abstract objects. However, there are some parts of the design that are not essential and could be improved.

3. Generality: The MIS/Interface design is general. For example, the PointT class can represent any point in 2D space not just a point on a TwoDotsBoard. The Board module is a general purpose 2D grid that can be customized with any dimensions and has some useful and general access programs that one may want to use on a 2D grid. The StrategyGameMode is also very general. It defines a very high level iterative process for playing a game of TwoDots, along with several abstract/template methods to check conditions as the game is played. When looked at it more closely the StrategyGameMode can be adopted to many different games, not just TwoDots. Any game that has a set number of moves, time or other conditions can be created by inheriting the StrategyGameMode module.

4. Minimality: In general, I tried to keep all modules minimal and address only a single concern. Most of the modules are minimal and do not offer more than one service, however there are a few exception and this is the challenge with designing a large system while maintaining such qualities. One example of a function that returns a value and has a potential state transition depending on a boolean condition is the checkWin() method in MovesStrategy module. This module return whether the user just eliminated 5 or more dots in move. However, if the user did not achieve this target then the state variable "moves" counter is decremented and then the function return false. So, here we have a state transition and a output thus not minimal. In the future, to make this minimal this can be achieved by splitting the function into two functions. One that checks the winning condition only and another that decrements the "moves" variable only.

5. Cohesion: In general, all the modules have high cohesion. Every module only has methods and data structures/types that are related. For example the Color module has a enum for colors as well as a access program to produce a random color. This function does not belong anywhere else.

6. Information hiding: I have kept the inner working details of my modules as encapsulated as possible. For example in my TwoDotsBoard module there are getter methods for the row and column. The randomColor() function is described as general as possible giving the flexbility of being implemented in any way. Same goes for the BoardView and CountDownTimer modules. These describe at a high level what should be outputted to the screen and how the timer should operate. In terms, of Java the timer may be implemented using the System.nanoTime() or other functions provided in System to access time or it may be done using the Timer class and set a Timer in the background as the game plays. The MIS, abstracts this all way and only requires a isCancelled() method to indicate whether a required amount of time has past or not.

Design Patterns

Proxy Pattern

The proxy pattern is used when we want to provide a client with certain functionality/services but we do not want them to have direct access to our software/modules/code. This may due to a potential security risk or simply or it just may not be a good choice to have direct access. We introduce in a "middle" man that communicates with our client and software. It takes the clients requests gets the data from our software and gives it back to the client. This is particularly useful in the development of modern Web Services, specifically RESTful API's. A company exposes a Web API available that offers HTTP endpoints for requesting data for example. Instead of giving client direct access to their database, a client makes a request to the API ,the API retrieves the info from the database and sends the results to the client.

Strategy Design Pattern

The strategy design pattern is generally used to encapsulate a family of different algorithms/objects that have some common behavior/attributes but also have some difference. This two dots game is a good example. The Timed version and the version with a set number of moves is very similar. The game plays exactly the same way except our winning condition is different for each. The strategy design pattern address's this issue by defining a interface with generic methods to replicate a similar process. For two different game modes we may have a play() method defined in a interface. Then a class can inherit this interface and implement the play() method. The game play is identical for both modes so we can write code to simulate the playing of the game and introduce a method checkWin() to check the winning condition. Then two additional classes, one for each game mode can inherit this class and have a unique implementation of checkWin(). This way we reduce code duplication and use modularity to encapsulate a family of closely related objects. Both modes inherit from one interface so they both have a play method