2AA4 Assignment 4 Specification

Madhi Nagarajan

April 6, 2020

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the Dots game. The spec utilizes the Model-View-Controller design pattern.

The model creates and sets up the Dots Game board and its arrangement, as well as modifying the board's state. The view displays the Dots Game Board & game messages to the user, and also receives the input from the user. The controller contains the logic and action of the Dots game, calling both the model and view to modify both the game board and display messages based on the game logic.

The Dots game itself is a puzzle game where the player tries to connect as many adjacent (non-diagonally) dots of the same colour to gain points. There can be various game modes implemented with Dots, the infinite and target modes implemented in this specification. An example of the Dots game can be seen in the `https://www.gamesxl.com/think/two-dots`.

Colour Type Module

Module

ColourT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

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

//R stands for Red, G for Green, B for Blue, Y for Yellow, P for Pink

Exported Access Programs

Routine name	In	Out	Exceptions
getRandomColour		ColourT	

Semantics

State Variables

None

State Invariant

None

Considerations

Access Routine Semantics

getRandomColour():

- output: *out* := a random type of ColourT
- exception: None

Dots ADT Module

Template/Model Module

Dots

Uses

ColourT

Syntax

Exported Types

Dots = ?

Exported Access Programs

Routine name	In	Out	Exceptions
Dots	$\mathbb{Z}$	Dots	
matrix		Seq(Seq(ColourT))	
n		$\mathbb{Z}$	
getColour	$\mathbb{Z}$, $\mathbb{Z}$	ColourT	
setColour	$\mathbb{Z}$, $\mathbb{Z}$		
addRandomColour	$\mathbb{Z}$		
setRandomColour	$\mathbb{Z}$, $\mathbb{Z}$		
initializeDots			
isValidPath	Seq(Seq($\mathbb{Z}$))	$\mathbb{B}$	InvalidInputException
dropDots			
processDots			
hasValidCombo		$\mathbb{B}$	

Semantics

State Variables

n: $\mathbb{Z}$
matrix: Seq of (Seq of ColourT)

State Invariant

None

Assumptions

The n state variable is to serve the purpose of being able to expand and/or shrink the board size and pieces.

Another assumption that is made is that the isValidPath will produce an InvalidInputException. In reality, an exception would stop the game in progress, thus the InvalidInputException is meant to only signify that the given input is invalid

Access Routine Semantics

Dots(num):

- transition: n, matrix := num, new Seq of (Seq of ColourT)
- output: *out* := *self*
- exception: None

matrix():

- output: *out* := matrix
- exception: None

n():

- output: *out* := n
- exception: None

getColour(i, j):

- output: *out* := matrix[i][j]
- exception: None

setColour(i, j, c):

- transition: matrix[i][j] := c
- exception: None

addRandomColour(i):

- transition: matrix$[i][-1]$:= ColourT.getRandomColour()
- exception: None

setRandomColour(i, j):

- transition: matrix$[i][j]$:= ColourT.getRandomColour()
- exception: None

initializeDots():

- transition: $\forall i, j$: matrix$[i][j]$:= ColourT.getRandomColour()
- exception: None

isValidPath(in):

- out: $\forall i$: matrix[input$[i][0]$][input$[i][1]$] == matrix[input$[i+1][0]$][input$[i+1][1]$]
- exception: $exc := (\exists i : (|\ \text{input}[i][0] - \text{input}[i+1][0]\ | > 1) \vee (|\ \text{input}[i][1] - \text{input}[i+1][1]\ | > 1)) \implies$ InvalidInputException

processDots():

- transition: matrix$[i][j]$:=
- exception: None

hasValidCombo():

- out: $\exists i, j$: (matrix$[i][j]$ == matrix$[i][j+1] \vee$ matrix$[i][j]$ == matrix$[i+1][j]$)
- exception: None

Local Functions

processDots(): Drops dots based on existing null cells in each column (of the matrix).

Dots View Module

View Module

DotsView

Uses

N/A

Syntax

Exported Types

?

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
startMenu			
printEnterNewInput			
displayScore	$\mathbb{Z}$		
displayTarget	$\mathbb{Z}$		
displayMovesLeft	$\mathbb{Z}$		
printInvalidMove			
printReshuffled			
printScoreReached			
printMovesOut			
renderDots	Seq(Seq(ColourT))		
getInput		string	

Semantics

State Variables

None

State Invariant

None

Assumptions

Since most of the routines in the View are just displaying print statements, the routines are ignored in the Access Routine Semantics. The purpose of each View routine is listed below:

- startMenu(): display the Game Menu message
- printEnterNewInput(): displays Enter New Input message
- displayScore(n): display Score of n
- displayTarget(n): display Target of n
- displayMovesLeft(n): display Moves of n
- printInvalidMove(): display Invalid Move message
- printReshuffled(): displays Board Reshuffled message
- printScoreReached(): displays Score Reached message
- renderDots(board): display Game Board by: $\forall i, j$: display each board$[i][j]$
- getInput(): read and return input message

Access Routine Semantics

N/A

Dots Controller Module

Controller Module

DotsController

Syntax

Uses

N/A

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
DotsController	Dots, DotsView	DotsController	
isValidInput	seq(string)	$\mathbb{B}$, $\mathbb{Z}$	InvalidInputException
startGame			
infiniteMode			
targetMode			

Semantics

State Variables

view: DotsView
model: Dots

State Invariant

None

Assumptions

One assumption that is made is that the isValidInput will produce an InvalidInputException. In reality, an exception would stop the game in progress, thus the InvalidInputException is meant to only signify that the given input is invalid.

Since the Game routines mainly consist of calling other view/model routines, they will not be described in the Access Routine Semantics.

- startGame(): Initiates the Game Board and menu messages
- infiniteMode(): Game mode that tracks user's score
- targetMode(): Game mode that challenges user to reach target score in a limited amount of moves

Access Routine Semantics

DotsController(dots, dotsview):

- transition: view, model := dots, dotsview
- out: $out := self$
- exception: None

isValidInput(input, n):

- output: $out := (|\text{input}| >= 2) \wedge (\forall i : |\text{input}[i]| == 2 \wedge (\text{input}[i][0] < n \wedge \text{input}[i][1] < n))$
- exception: $exc := \neg(\text{isValidInput(input,n)}) \implies \text{InvalidInputException}$

Critique of the Design

The overall design of the MVC design pattern flows well with the design aspect of a game, as it enables information hiding for the Dots Model and DotsView from DotsController. DotsController does not have direct access in modifying the matrix and size of the board in the Dots Model. The design pattern also restricts modification of the messages displayed or the input received in DotsView.

High cohesion is also present with this specification, as the DotsController is highly dependent on the behaviour of its model and view. The DotsController is the module that controls the logic of the Dots Model and the action of the DotsView.

In addition, miminality was taken into account for this specification. For all modules, its access routines either does a State Transition or an Output, but never both.

Generality is one area that is lacking in this specification, as there are no generic modules or interfaces implemented. While it's not a key design aspect for this project, generality can be improved by creating a generic module for the Dots Model. The model can accept any type (T) for it's 2D array matrix. We can further create a subclass/sub-model of the generic Dots class to implement it for ColourT types.

Finally, consistency is used throughout this specification, ensuring uniform naming conventions and exception handling etc.

Answer to Design Questions

1. While all of the UML diagrams for these design patterns are very similar, they all have their own intended uses when utilizing them in Software projects.

   The purpose of the Proxy pattern is to have an interface as a means for a client to access an object, class etc. through that interface. However, this prevents the client from directly accessing that object/class.

   The Adapter pattern is a pattern that connects two otherwise incompatible interfaces together by wrapping an existing class with the target interface. That way, the client gains functionality between both interfaces. This differs from the proxy because an Adapter provides access to another interface, while proxy provides access to the only interface created.

   The Strategy pattern is when we develop an interface that handles multiple implementations, enabling abstraction. The defining factor of the Strategy pattern is that the class behaviour (or implementation) can be selected/changed at runtime. The Strategy pattern is quite different compared to the other two, due to the runtime feature and that the interface deals with multiple implementations.

2. Control Flow Diagram: