Assignment 4 Model & View Specification

Yousam Asham, ashamy1

April 2, 2020

Overview of Design

This specification outlines how to construct the model and view modules of an MVC design pattern for a game of Dots. It consists of six modules in total.

The PointT module is an ADT to express certain points in a 2D sequence. This module is an integral part of this design since it make referencing certain points of the game board easier. It also minimizes the number of parameters passed into a lot of functions.

The ColourT module is a module that enumerates the five colours used in a Dots game. It also has a `toString` method that maps each of the ColourT objects to their respective strings for ease of access in the PrintBoard module.

The RandomElem module is an interface module that provides a method called `setRandElem`. This interface allows the developer to change the way that the digital board game generates its new elements, given a PointT object which indicates which entry in the board is should change.

The GameBoard module is a generic module that outlines the usage and modification of an abstract object through the usage of exported access programs. This abstract object is the GameBoard to be used. This module implements the RandomElem interface, however, since the GameBoard module is generic (meaning that the elements are not specified), the implemented `setRandElem` method only sets the entry indicated by the PointT object to null since the GameBoard module is a generic module.

The DotsBoardT module is the main module for the model part of the MVC design pattern. It is essentially a GameBoard module but instead of being generic, it uses ColourT to satisfy the requirement of creating a model for a game of Dots.

The PrintBoard module is responsible for the view module which is part of the MVC design pattern.

This design considers many changes such as the following:

- The change to decide to create a different game that is similar to Dots in a sense that it uses a game board. The GameBoard module that is designed within this

specification can be used to create a different digital board game that essentially requires the generation of new game elements after some have been removed.

- It also allows for a change in the scoring system, as well as the level selection; this design allows for the changing of parameters to create new levels and add more challenging layers to a digital game.

- In addition, the way that the elements on the game board can be referenced is also account for as part of change to this specification. The following specification uses a simple PointT module that can be used to construct PointT objects to reference specific points in the game board object. If a developer wishes to change this requirement, they are free to introduce a new way to referrence the game board elements instead of PointT objects.

- Also, if a developer wishes to change their game's way of coming up with new elements for their game, they are free to do so through implementing the `RandomElem` interface their own way.

- Finally, if a developer wishes to change the actual objects that are the elements of the game board (i.e. the ColourT objects in this specification), they are free to do so by just changing the ColourT to suit their element type of choice.

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

row():

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

col():

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

ColourT Module

Module

ColourT

Uses

None

Syntax

Exported Constants

None

Exported Types

ColourT = {blue, green, purple, orange, red} //*Enumerated*

Exported Access Programs

Routine name	In	Out	Exceptions
toString			

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Sematics

toString():

- output: $out := self = \text{blue} \Rightarrow \text{"Blue"} \mid self = \text{green} \Rightarrow \text{"Green"} \mid self = \text{purple} \Rightarrow \text{"Purple"} \mid self = \text{orange} \Rightarrow \text{"Orange"} \mid \text{true} \Rightarrow \text{"Red"}$
- exception: none.

RandomElem Interface Module

Generic Module

RandomElem

Uses

PointT

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
setRandElem	PointT		IndexOutOfBoundsException

Generic GameBoard Module inherits RandomElem interface

Generic Module

GameBoard(T)

Uses

PointT, RandomElem

Syntax

Exported Constants

SIZE = 6 //*size of the board in each direction*

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
init	seq of (seq of T), $\mathbb{N}, \mathbb{N}$, T		
getBoard		seq of (seq of T)	
getElement	PointT		IndexOutOfBoundsException
getScore		$\mathbb{N}$	
getMoves		$\mathbb{N}$	
getTargetMoves		$\mathbb{N}$	
getTargetScore		$\mathbb{N}$	
getTargetType		T	
popElems	seq of PointT		IndexOutOfBoundsException, IllegalArgumentException, ArithmeticException
remove	PointT		IndexOutOfBoundsException
rebase			
setRandElem	PointT		IndexOutOfBoundsException
has_won		$\mathbb{B}$	

Semantics

State Variables

g: seq of (seq of T)
$score$: $\mathbb{N}$
$moves$: $\mathbb{N}$
$targetMoves$: $\mathbb{N}$
$targetScore$: $\mathbb{N}$
$targetElement$: T

State Invariant

count(T_0) + count(T_1) + count(T_2) + ... + count(T_n) = SIZE*SIZE
$//Where\ n\ represents\ the\ quantity\ of\ the\ number\ of\ elements\ in\ the\ type$ T

Assumptions

- The init method is called for the abstract object before any other access routine is called for that object. The init method can be used to return the state of the game to the state of a new game.
- There exists a function called random() that returns a number between 0 and 1.

Access Routine Semantics

init(s, tm, ts, te):

- transition: $g, score, moves, targetMoves, targetScore, targetElement := s, 0, 0, tm, ts, te$
- exception: none.

getElement(p):

- output: $out := g[p.row][p.col]$
- exception: $exc := \neg(\text{validPoint}(p)) \Rightarrow$ IndexOutOfBoundsException

getBoard():

- output: $out := g$

- exception: none.

getScore():

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

getMoves():

- output: $out := moves$
- exception: none.

getTargetMoves():

- output: $out := targetMoves$
- exception: none.

getTargetScore():

- output: $out := targetScore$
- exception: none.

getTargetType():

- output: $out := targetelement$
- exception: none.

popElems(s):

- transition: $score, moves := score + +(p : \text{PointT} \mid p \in s \wedge g[p.row][p.col] = targetColour : 1), moves + 1$
- exception: exc := $\exists(p : \text{PointT} \mid p \in s : \neg\text{validPoint}(p)) \Rightarrow$ IndexOutOfBoundsException | $\neg(\text{isAdjList}(s)) \vee (|s| < 2) \vee \neg(\text{sameColour}(s)) \Rightarrow$ IllegalArgumentException | $moves > targetMoves \Rightarrow$ ArithmeticException

remove(p):

- transition: $g[p.row][p.col] = null$

- exception: $exc := \neg(\text{validPoint}(p)) \Rightarrow \text{IndexOutOfBoundsException}$

rebase():

- transition: $g[i][j] = null \wedge \neg(\text{validRowCol}(i-1)) \Rightarrow g[i][j] := \text{getRandElem}((10 * random())\%5) \mid g[i][j] = null \wedge g[i-1][j] \neq null \Rightarrow g[i][j], g[i-1][j] := g[i][j-1], null$. This happens repeatedly while iterating through all the rows of the game board (excluding the first row, $i = 0$), SIZE times. The end state of the game board should satisfy $\forall(p : \text{PointT} \mid \text{validPoint}(p) : g[p.row][p.col] \neq null)$ This is to re-populate the game board.
- exception: none.

setRandElem(p):

- transition: $g[p.row][p.col] = null$
- exception: $exc := \neg(\text{validPoint}(p)) \Rightarrow \text{IndexOutOfBoundsException}$

has_won():

- output: $out := (moves \leq targetMoves) \wedge (score \geq targetScore)$
- exception: none.

Local Functions

isAdjList: seq of PointT $\to \mathbb{B}$
isAdjList(s) $\equiv \forall(i : \mathbb{N} \mid 0 \leq i \leq |s| - 2 : (s[i].row = s[i+1].row - 1) \vee (s[i].row = s[i+1].row + 1) \vee (s[i].col = s.col[i+1] - 1) \vee (s[i].col = s[i+1].col + 1) \wedge \neg((s[i].row = s[i+1].row + 1) \wedge (s[i].col = s[i+1].col + 1)) \wedge \neg((s[i].row = s[i+1].row - 1) \wedge (s[i].col = s[i+1].col - 1))$

validRowCol:$\mathbb{N} \to \mathbb{B}$
validPoint(i)$\equiv (0 \leq i < SIZE)$

validPoint: PointT $\to \mathbb{B}$
validPoint(p)$\equiv$ validRowCol($p.row$) $\wedge$ validRowCol($p.col$)

sameColour: seq of PointT $\to \mathbb{B}$
sameColour(s) $\equiv \forall(i : \mathbb{N} \mid 0 \leq i \leq |s|-2 : g[s[i].row][s[i].col] = g[s[i+1].row][s[i+1].col])$

DotsBoardT Module and inherits the RandomElem interface

Module

DotsBoardT is GameBoard(ColourT)

Uses

PointT, RandomElem, GameBoard

Syntax

Exported Constants

SIZE = 6 //*size of the board in each direction*

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
setRandElem	PointT		IndexOutOfBoundsException

Semantics

State Invariant

count(blue) + count(green) + count(purple) + count(orange) + count(red) = SIZE*SIZE

Assumptions

- The init method is called for the abstract object before any other access routine is called for that object. The init method can be used to return the state of the game to the state of a new game.
- There exists a function called random() that returns a number between 0 and 1.

Access Routine Semantics

setRandElem(p):

- transition: $g[p.row][p.col]$ = getRandomElem()
- exception: $exc :=$ ¬validPoint(p) ⇒ IndexOutOfBoundsException.

Local Functions

getRandomElem: ColourT
getRandomElem() $\equiv 0 <$ random() $\leq 0.2 \Rightarrow$ *blue* | $0.2 <$ random() $\leq 0.4 \Rightarrow$ *green* | $0.4 <$ random() $\leq 0.6 \Rightarrow$ *purple* | $0.6 <$ random() $\leq 0.8 \Rightarrow$ *orange* | $0.8 <$ random() $\leq 1.0 \Rightarrow$ *red*

isAdjList: seq of PointT $\rightarrow \mathbb{B}$
isAdjList(s) $\equiv \forall(i : \mathbb{N} \mid 0 \leq i \leq |s| - 2 : (s[i].row = s[i+1].row - 1) \vee (s[i].row = s[i+1].row + 1) \vee (s[i].col = s.col[i+1] - 1) \vee (s[i].col = s[i+1].col + 1) \wedge \neg((s[i].row = s[i+1].row + 1) \wedge (s[i].col = s[i+1].col + 1)) \wedge \neg((s[i].row = s[i+1].row - 1) \wedge (s[i].col = s[i+1].col - 1))$

validRowCol:$\mathbb{N} \rightarrow \mathbb{B}$
validPoint(i)$\equiv (0 \leq i < SIZE)$

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

sameColour: seq of PointT $\rightarrow \mathbb{B}$
sameColour(s) $\equiv \forall(i : \mathbb{N} \mid 0 \leq i \leq |s|-2 : g[s[i].row][s[i].col] = g[s[i+1].row][s[i+1].col])$

count: T $\rightarrow \mathbb{N}$
count(e) $\equiv +(p$:PointT| validPoint(p) $\wedge\ g[p.row][p.col] = e : 1)$

PrintBoard Module

Module

PrintBoard

Uses

ColourT, PointT, GameBoard

Syntax

Exported Constants

None

Exported Types

PrintBoard = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new PrintBoard	seq of (seq of ColourT)	Print	IllegalArgumentException
getElem	PointT	ColourT	IndexOutIOfBoundsException
print			

Semantics

Environmental Variables

screen: a two dimensional sequence of positions on the screen, where each position holds a character.

State Variables

g : seq of (seq of characters)

State Invariant

None

Assumptions

- The constructor is called for the class before any other access routine is called for that object.

Access Routine Semantics

new PrintBoard(s):

- transition: $s[i][j]$ = ColourT.blue $\Rightarrow$ $g[i][j]$:= 'b' | $s[i][j]$ = ColourT.green $\Rightarrow$ $g[i][j]$:= 'g' | $s[i][j]$ = ColourT.purple $\Rightarrow$ $g[i][j]$:= 'p' | $s[i][j]$ = ColourT.orangle $\Rightarrow$ $g[i][j]$:= 'o' | $s[i][j]$ = ColourT.red $\Rightarrow$ $g[i][j]$:= 'r' such that $i : \mathbb{N}$, $j : \mathbb{N}$ iterate over the every element in s.
- output: $out := self$
- exception: exc := An element in s is not of type ColourT $\Rightarrow$ IllegalArgumentException.

getElem(p):

- output: $out := g[p.row][p.col]$
- exception: $exc := \neg(\text{validPoint}(p)) \Rightarrow$ IndexOutOfBoundsException.

print():

- transition: Change the state of *screen* by changing the element returned by getElem(p), where the state variables of p: PointT are changed so every element of the board is returned sequentially row by row with a newline charachter being printed every DotsBoard.SIZE elements, changing the state of *screen* to be on the next row. After the last row, the *score*, *moves*, $targetMoves$, $targetScore$, and the $targetElement$ is displayed on *screen* in the format "$score/targetScore$ $moves/targetMoves$ Target: $targetElement$"
- exception: none

Local Functions

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

validRowCol:$\mathbb{N} \rightarrow \mathbb{B}$
validPoint(i)$\equiv (0 \leq i < SIZE)$

Specification Critique

This specifcation was very challenging and thought provoking to come up with. I had to make a lot of design decisions in the process such as: whether the `setRandElem` method should be included in the MIS as an interface or just an exported access program that is included within the ColourT module. I decided to include it in an interface; that way the selection of a new element entry can be customized based on the game to be implemented. This supports the design for change principle talked about earlier in the semester.

In terms of software qualities, this specification is pretty consistent. This is because formal language (discrete maths) was used to explain what each of the access programs (and local functions) did. However, there were some parts that were not understandable if discrete math formal language was used; for these parts some informal language had to be used. I have tried to make this specification as consistent as possible. Also, adding the PointT module to this specification made it more consistent since now all the points in a 2D sequence can be referenced using a standardized way, which is with the use of a PointT object.

This specification is mostly essential. Keeping this specification essential was hard since I had to separate the move of popping dots into three stages that had to be independent of each other. These three stagees are represented by the following three method of the GameBoard module: `popElem, remove, rebase`.

As for generality, this specification would be high scoring. The GameBoard module is general since it is generic. Moreover, the choice to have the RandomElem interface adds more generality to this specification since it has gives the option for the usage of other randomization/non-randomization methods.

This specification had to sacrifice some minimality. This can be seen in the `popElems()` method in the GameBoard module. This sacrifice had to be made since it would be too much work for a controller to call two separate methods one changing the score while the other changes the number of moves already taken in a dots game. This sacrifice had to be made for the sake of usability. I also had the option to combine the `remove` and the `rebase` methods into one method (since they both change only one state variable, the grid) but this would have made the resulting method a large one that would be hard to understand. Anyways, both of these options would be considered minimal.

In terms of cohesion, I believe that all of these modules go and work well together. The only module that would be hard to connect to the other would be the PrintBoard module, however, this module had to be included to serve as the main module for the view module of an MVC design pattern.

Information hiding was well maintained through out the implementation and the specification. To make sure of that, all the local functions were implemented as private functions, and the state variables were declared as `protected`. Also, since the specification

maintains generality, this means that it maintains design for change. Design for change is an influencing factor of information hiding.

Answers to questions

1. Proxy pattern: The proxy pattern allows for an intermediary object between a client trying to communicate to a component. This intermediary object can be seen as a representative of the component that the client is trying to communicate to. This design pattern is usually used to prevent the client from accessing a component that is unsafe or non-secure. This design pattern would be used while browsing to help keep the client safe while browsing the web and to make sure that the wesites the client would be accessing are safe websites.

   Adapter pattern: The adapter pattern is used to accomodate two incompatible interfaces. For example, in a video game, there is a HumanPerson interface that defines what a human can do as exported access programs such as `walk()` and `jump()`. However if you wanted to instantiate an object thats not human (suppose that object we need to instantiate is a basketball), the ball cannot perform the same actions as a human. Therefore the developer would need an adapter class that would implement methods `walk()` that points to a different method called `roll()`, and it would implement the `jump()` method to point to a method called `bounce()`. This is how an adapter design pattern could be used.

   Strategy pattern: The strategy design pattern allows for the creation of an interface and then creating a class that implements that interface in two ways, and all of that would happen in one module. For example, suppose we have a Animal super class and a dog module that extends the Animal module and a bird module that also extends the Animal module. In order to give the dog and the bird flying abilities (or something to indicate if they can fly or not), we would create a FlyOrNo interface and within that interface module we could define two classes (one called CanFly while the other is called CantFly) that has access programs that implement that FlyOrNo interface so that it could fly, and the other class implements it as if they cannot fly. It would then add a new instance variable pertaining to whether the animal has the ability to fly or not. This type of design pattern would avoid repetition of code within the dog and the bird class and will instead make use of an instance variable to indicate whether an animal could fly or not.

   All of these design patterns allow the developer to implement their code without any loss of generality and therefore lead to abstraction. All these design patterns allow for designing with change in mind. If any changes need to be made to the

implementation of one of these design patterns the change would not be problematic and would be easy to make.

2.