Assignment 3, Part 1, Specification

SFWR ENG 2AA4
Jay Mody - 400195508 - modyj

March 2, 2020

This Module Interface Specification (MIS) document contains modules, types and methods for implementing a generic 2D sequence that is instantiated for both land use planning and for a Discrete Elevation Model (DEM).

In applying the specification, there may be cases that involve undefinedness. We will interpret undefinedness following [?]:

If $p : \alpha_1 \times .... \times \alpha_n \rightarrow \mathbb{B}$ and any of $a_1, ..., a_n$ is undefined, then $p(a_1, ..., a_n)$ is False. For instance, if $p(x) = 1/x < 1$, then $p(0) =$ False. In the language of our specification, if evaluating an expression generates an exception, then the value of the expression is undefined.

[The parts that you need to fill in are marked by comments, like this one. In several of the modules local functions are specified. You can use these local functions to complete the missing specifications. —SS]

[As you edit the tex source, please leave the `wss` comments in the file. Put your answer **after** the comment. This will make grading easier. —SS]

Land Use Type Module

Module

LanduseT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Landtypes = {R, T, A, C}

//R stands for Recreational, T for Transport, A for Agricultural, C for Commercial

Exported Access Programs

Routine name	In	Out	Exceptions
new LanduseT	Landtypes	LanduseT	

Semantics

State Variables

landuse: Landtypes

State Invariant

None

Access Routine Semantics

new LandUseT(t):

- transition: $landuse := t$

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

Considerations

When implementing in Java, use enums (as shown in Tutorial 06 for ElementT).

Point ADT Module

Template Module inherits Equality(PointT)

PointT

Uses

N/A

Syntax

Exported Types

[What should be written here? —SS] PointT = ?

Exported Access Programs

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

Semantics

State Variables

r: [What is the type of the state variables? —SS] $\mathbb{Z}$
c: [What is the type of the state variables? —SS] $\mathbb{Z}$

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: [What should the state transition be for the constructor? —SS]
  $r, c := row, col$
- output: $out := self$
- exception: None

row():

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

col():

- [What should go here? —SS] output: $out := c$
- exception: None

translate(Δr, Δc):

- [What should go here? —SS] output: $out := \text{PointT}(r + \Delta r, c + \Delta c)$
- exception: [What should go here? —SS] None

Generic Seq2D Module

Generic Template Module

Seq2D(T)

Uses

PointT

Syntax

Exported Types

Seq2D(T) = ?

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Seq2D	seq of (seq of T), $\mathbb{R}$	Seq2D	IllegalArgumentException
set	PointT, T		IndexOutOfBoundsException
get	PointT	T	IndexOutOfBoundsException
getNumRow		$\mathbb{N}$	
getNumCol		$\mathbb{N}$	
getScale		$\mathbb{R}$	
count	T	$\mathbb{N}$	
countRow	T, $\mathbb{N}$	$\mathbb{N}$	
area	T	$\mathbb{R}$	

Semantics

State Variables

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

State Invariant

None

Assumptions

- The Seq2D(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.
- Assume that the input to the constructor is a sequence of rows, where each row is a sequence of elements of type T. The number of columns (number of elements) in each row is assumed to be equal. That is each row of the grid has the same number of entries. $s[i][j]$ 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

Seq2D(S, scl):

- transition: [Fill in the transition. —SS]
  $s, scale, nRow, nCol := S, scl, |S|, |S[0]|$
- output: $out := self$
- exception: [Fill in the exception. One should be generated if the scale is less than zero, or the input sequence is empty, or the number of columns is zero in the first row, or the number of columns in any row is different from the number of columns in the first row. —SS]

	$exc :=$
$scl \leq 0$	IllegalArgumentException
$\|S\| = 0$	IllegalArgumentException
$\|S[0]\| = 0$	IllegalArgumentException
$\exists$ row $\in S \, . \, \neg(\|\text{row}\| = \|S[0]\|)$	IllegalArgumentException

set(p, v):

- transition: [? —SS] $s[p.row()][p.col()] := v$
- exception: [Generate an exception if the point lies outside of the map. —SS]
  $exc := \neg(\text{validPoint(p)}) \Rightarrow$ IndexOutOfBoundsException

get(p):

- output: [? —SS] $out := s[p.row()][p.col()]$
- exception: [Generate an exception if the point lies outside of the map. —SS] $exc := \neg(\text{validPoint(p)}) \Rightarrow \text{IndexOutOfBoundsException}$

getNumRow():

- output: $out := \text{nRow}$
- exception: None

getNumCol():

- output: $out := \text{nCol}$
- exception: None

getScale():

- output: $out := \text{scale}$
- exception: None

count(t: T):

- output: [Count the number of times the value t occurs in the 2D sequence. —SS] $out := +(i, j : \mathbb{N} | \text{validRow(i)} \wedge \text{validCol(j)} \wedge s[i][j] = t : 1)$
- exception: None

countRow(t: T, $i : \mathbb{N}$):

- output: [Count the number of times the value t occurs in row i. —SS] $out := +(j : \mathbb{N} | \text{validCol(j)} \wedge s[i][j] = t : 1)$
- exception: [Generate an exception if the index is not a valid row. —SS] $exc := \neg(\text{validRow(i)}) \Rightarrow \text{IndexOutOfBoundsException}$

area(t: T):

- output: [Return the total area in the grid taken up by cell value t. The length of each side of each cell in the grid is scale. —SS] $out := +(i, j : \mathbb{N} | \text{validRow(i)} \wedge \text{validCol(j)} \wedge s[i][j] = t : \text{scale}^2)$
- exception: None

Local Functions

validRow: $\mathbb{N} \rightarrow \mathbb{B}$
[returns true if the given natural number is a valid row number. —SS]
validRow(n) $\equiv 0 \leq n \leq (\text{nRow} - 1)$

validCol: $\mathbb{N} \rightarrow \mathbb{B}$
[returns true if the given natural number is a valid column number. —SS]
validCol(n) $\equiv 0 \leq n \leq (\text{nCol} - 1)$

validPoint: PointT $\rightarrow \mathbb{B}$
[Returns true if the given point lies within the boundaries of the map. —SS]
validPoint(p) $\equiv$ validRow(p.row()) $\wedge$ validCol(p.col())

LanduseMap Module

Template Module

[Instantiate the generic ADT Seq2D(T) with the type LanduseT —SS]
LanduseMapT is Seq2D(LanduseT)

DEM Module

Template Module

DemT is Seq2D($\mathbb{Z}$)

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
total		$\mathbb{Z}$	
max		$\mathbb{Z}$	
ascendingRows		$\mathbb{B}$	

Semantics

Access Routine Semantics

total():

- output: [Total of all the values in all of the cells. —SS]
  $out := +(i, j : \mathbb{N} | \text{validRow(i)} \land \text{validCol(j)} : s[i][j])$
- exception: None

max():

- output: [Find the maximum value in the 2d grid of integers —SS]
  $\exists\ x, y : \mathbb{Z}\ .\ (\text{validRow}(x) \land \text{validCol}(y) \land$
  $(\forall i, j : \mathbb{Z}\ .\ \text{validRow}(i) \land \text{validCol}(j) \land s[x][y] \geq s[i][j])) \Rightarrow out := s[x][y])$
- exception: None

ascendingRows():

- output: [Returns True if the sum of all values in each row increases as the row number increases, otherwise, returns False. —SS]
  $\forall\ i, j : \mathbb{Z}\ .\ \text{validRow}(i) \land \text{validRow}(i-1) \land \text{validCol}(j) \land +(s[i][j] - s[i-1][j]) > 0$
- exception: None

Local Functions

validRow: $\mathbb{N} \rightarrow \mathbb{B}$
[returns true if the given natural number is a valid row number. —SS]
validRow(n) $\equiv 0 \leq n \leq (\text{nRow} - 1)$

validCol: $\mathbb{N} \rightarrow \mathbb{B}$
[returns true if the given natural number is a valid column number. —SS]
validCol(n) $\equiv 0 \leq n \leq (\text{nCol} - 1)$

Critique of Design

[Write a critique of the interface for the modules in this project. Is there anything missing? Is there anything you would consider changing? Why? One thing you could discuss is that the Java implementation, following the notes given in the assignment description, will expose the use of ArrayList for Seq2D. How might you change this? There are repeated local functions in two modules. What could you do about this? —SS]

The most notable mistake in the design specification is that it's missing 2 modules, the Equality module and the Landtypes module. Equality is inherited by PointT and Landtypes is used by LanduseT, but neither are ever defined anywhere in the MIS. To make matter worse, LanduseT is redundant. It's sole purpose is to store a Landtype variable, adding no additional functionality. To remove this redundancy, it would be advantagous to use Landtype directly, removinng an unecessary layer of abstraction. Additionally, there is a redundancy with validRow/validCol, as they are duplicated in Seq2D and DemT as local functions. A better design would be to include validRow, validCol, and validPoint all as part of the interface for Seq2D, so that any modules that inherit it will also have access to those functions. Furthermore, the use of an ArrayList in the implementation for Seq2D as stated in the assignment description, may not be the optimal option for the task. Seq2D is analagous to a matrix, with an equivalent number of elements in each row. To make things simpler, we can make the use of a 2 dimensional array, rather than an ArrayList of ArrayLists.

In addition to your critique, please address the following questions:

1. The original version of the assignment had an Equality interface defined as for A2, but this idea was dropped. In the original version Seq2D inherited the Equality interface. Although this works in Java with the LanduseMapT, it is problematic for DemT. Why is it problematic? (Hint: DEMT is instantiated with the Java type Integer.)

   By default, classes that are built into the Java langauge (Integer, ArrayList, etc ...) already inherit an equivalence method. By default, this method ".equals()", compares object ids, however, in many cases this method is overriden to test for equivalence for whatever data it is representing. Since LanduseT is a class we defined, this is no problem if we define our own

2. Although Java has several interfaces as part of the standard language, such as the Comparable interface, there is no Equality interface. Instead equals is provided

through inheritance from Object. Why do you think the Java language designers decided to use inheritance for equality, instead of providing an interface?

One reason why the Java designers may have decided to use inheritance for equality is that it allows them to define the default equivalence behaviour of an object by if their id's are equivalent (aka the same object in memory). Another, more important reason, is that it reduces ambiguity and redundancy. Many objects may have an equality property, but not a less than or greater than property. To make things less ambigious, it's easier to have equality inherited and overridden than via some workaround with compareTo (or as a seperate interface). As for redundancy, equality is a very common property (like toString), so it's easier to have it as it's own inhereited method rather than as an interface.

3. The qualities of good module interface push the design of the interface in different directions. Why is it rarely possible to achieve a module interface that simultaneously is essential, minimal and general?

   To create the most general interface, you often really need to think ahead in the future and realize the most modular and encaplsulating form of the module. Often this involves cutting down many functions that may have been essential to the initial design, but not to it's more general version. Or, on the other hand, you need to define added functionality to encaplsulate a wider range of modules that may implement an interface. In this sense, designing interfaces that are general, essential, and minimal proves to be a difficult task. You may have to redesign you're interface, add new modules/interfaces, and go through many iterations before you can come to a healthy balance of all three.