Assignment 3, Part 1, Specification

SFWR ENG 2AA4

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():

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

translate(Δr, Δc):

- output: [What should go here? —SS] $out :=$ 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, $\mathbb{N}$	$\mathbb{N}$	IndexOutOfBoundsException
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] $s, scale, nRow, nCol := S, scl, |S[0]|, |S|$
- 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]
  $(scale \leq 0 \vee |S| = 0 \vee |S[0]| = 0 \vee \neg\forall(l : \text{seq of T} | l \in S : |l| = |S[0]|)) \implies$ IllegalArgumentException

set(p, v):

- transition: [? —SS] $s[p.r][p.c] := v$
- exception: [Generate an exception if the point lies outside of the map. —SS]
  $\neg validPoint(p) \implies$ IndexOutOfBoundsException

get(p):

- output: [? —SS] $out := s[p.r][p.c]$
- exception: [Generate an exception if the point lies outside of the map. —SS]
  $\neg validPoint(p) \implies$ 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]
  $out := +(j : \mathbb{N} | \text{validRow}(i) \wedge \text{validRow}(j) \wedge s[i][j] = t : 1)$
- exception: [Generate an exception if the index is not a valid row. —SS]
  $\neg\text{validRow}(i) \implies \text{IndexOutOfBoundsException}$

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]
  $out := count(t) * scale$
- exception: None

Local Functions

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

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

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

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		$\mathbb{B}$	

Exported Constants

$maxPoint := s[0][0]$

Semantics

Access Routine Semantics

total():

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

max():

- output: [Find the maximum value in the 2d grid of integers —SS]
  $out := \forall(i, j : \mathbb{N} : P : \text{PointT} | validRow(i) \wedge validCol(j) \wedge s[i][j]) < maxPoint)$
- 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]
  $out := \neg\exists(i : \mathbb{N} | \text{validRow}(i-1) : +(j : \mathbb{N} | \text{validCol}(j) : s[i-1][j]) < +(j : \mathbb{N} | \text{validCol}(j) : s[i][j]))$
- exception: None

Local Functions

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

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

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]

One such change that I would consider changing is to have the DEM module to also inherit the local functions (eg. validRow). Since these functions provide the same purpose in both modules, we do not need to instantiate these functions in both the Seq2D and DEM module. This would improve the modularity and simplicity of the specification. In addition, declaring in the specification to utilize ArrayLists defeats the idea of abstraction, as it reveals the implementation details to the user/client. Rather, the specification should allow the programmer to utilize whatever type that they feel is most suitable.

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.)

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?

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?

Answers:

1. The Equality interface would work for a seq2D of LanduseMapT since it utilizes enum (int) values as representation. However, the DemT implementation utilizes Integer object types. Thus, the Equality interface would not work for DemT, as you have to utilize equals() for checking the values of any object type. Furthermore, Java has it's own built-in Equality module, which developers can take advantage of, rather than creating a new Equality interface.

2. Java utilizes inheritance for Equality, rather than an interface, because it's simple to implement an equals() method that works for both primitive and built-in object types (like String and Arraylist). Since types like String, Integer, and Arraylist act as objects in java and are commonly used in programs all the time, it makes sense that Java designers ensured that equals() works with object types. If a developer wished to use equals(), all they have to do is to create an override equals() method that handles equality for a custom object type.

3. It is very difficult to implement a module interface that has essentiality, minimality and generality. This is because you will be very limited, in terms of functionality, for that interface if you wish to have all three qualities. Essentiality aims to omit unnecessary features that could be done with less routines instead. However, this would make an interface more specific because its routines would have more functionality. Thus, generality would be sacrificed. Minimality would also cause issues, as it's difficult to ensure that setting state variables are only being set once with your access routines, since generality prevents this.