Assignment 3, Part 1, Specification

SFWR ENG 2AA4

March 12, 2018

The purpose of this software design exercise is to design and implement a portion of the specification for a Geographic Information System (GIS). This document shows the complete specification, which will be the basis for your implementation and testing. In this specification natural numbers ($\mathbb{N}$) include zero (0).

[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]

Map Types Module

Module

MapTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

CompassT = {N, S, E, W}
LanduseT = {Recreational, Transport, Agricultural, Residential, Commercial}
RotateT = {CW, CCW}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Point ADT Module

Template Module

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	
x		$\mathbb{Z}$	
y		$\mathbb{Z}$	
translate	$\mathbb{Z}$, $\mathbb{Z}$	PointT	

Semantics

State Variables

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

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(x, y):

- transition: xc, yc := x, y [What should the state transition be for the constructor? —SS]
- output: *out* := *self*
- exception: None

x():

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

y():

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

translate(Δx, Δy):

- output: PointT $(xc + \Delta x, yc + \Delta y)$ [What should go here? —SS]
- exception: None [What should go here? —SS]

Line ADT Module

Template Module

LineT

Uses

[What should go here? —SS] PointT, MapTypes

Syntax

Exported Types

LineT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
LineT	PointT, CompassT, $\mathbb{N}$	LineT	invalid_argument
strt		PointT	
end		PointT	
orient		CompassT	
len		$\mathbb{Z}$	
flip		LineT	
rotate	RotateT	LineT	
translate	$\mathbb{Z}$, $\mathbb{Z}$	LineT	

Semantics

State Variables

s: PointT
o: CompassT
L: $\mathbb{N}$

State Invariant

None

Assumptions

The constructor LineT 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

LineT($st, ornt, l$):

- transition: $s, o, L := st, ornt, l$
- output: $out := self$
- exception: $(l = 0 \Rightarrow \text{invalid\_argument})$ [Write the spec for an exception when the length of the line is 0 —SS]

strt():

- output: $out := \text{PointT}(s.x, s.y)$
- exception: None

end():

- output: $(o = N \Rightarrow \text{PointT}(s.x, s.y + L - 1) | o = E \Rightarrow \text{PointT}(s.x + L - 1, s.y)$ $| o = S \Rightarrow \text{PointT}(s.x, s.y - (L - 1)) | o = W \Rightarrow \text{PointT}(s.x - (L - 1), s.y))$ [Write the spec for returning the end point of the line. —SS]
- exception: None

orient():

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

len():

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

flip():

- output: $(o = N \Rightarrow \text{LineT}(s, S, L) | o = E \Rightarrow \text{LineT}(s, W, L)$
  $| o = S \Rightarrow \text{LineT}(s, N, L) | o = W \Rightarrow \text{LineT}(s, E, L))$
  [Write the spec for returning a new line that is the mirror image of the current line. That is, the start point and length of the new line will remain the same, but the orientation will be changed by 180 degrees —SS]

- exception: None

rotate(r):

- output:

		out :=
$r =$ CW	$o =$ N	LineT(s, E, L)[? —SS]
	$o =$ S	LineT(s, W, L) [? —SS]
	$o =$ W	LineT(s, N, L) [? —SS]
	$o =$ E	LineT(s, S, L) [? —SS]
$r =$ CCW	$o =$ N	LineT(s, W, L) [? —SS]
	$o =$ S	LineT(s, E, L) [? —SS]
	$o =$ W	LineT(s, S, L) [? —SS]
	$o =$ E	LineT(s, N, L) [? —SS]

- exception: None

translate(Δx, Δy):

- output: LineT(s.translate(Δx, Δy), o, L) [Add the missing spec —SS]

- exception: None

Path ADT Module

Template Module

PathT

Uses

PointT, LineT, MapTypes

Syntax

Exported Types

PathT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PathT	PointT, CompassT, $\mathbb{N}$	PathT	
append	CompassT, $\mathbb{N}$		invalid_argument
strt		PointT	
end		PointT	
line	$\mathbb{N}$	LineT	outside_bounds
size		$\mathbb{N}$	
len		$\mathbb{N}$	
translate	$\mathbb{Z}$, $\mathbb{Z}$	LineT	

Semantics

State Variables

s: sequence of LineT

State Invariant

None

Assumptions

- The constructor PathT 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

PathT($st, ornt, l$):

- transition: $s := s||$ < LineT($st, ornt, l$) > [What is the spec to add the first element to the sequence of LineT? —SS]
- output: $out := self$
- exception: None

append($ornt, l$):

- transition: $s := s||$ < LineT(adjPt($ornt$), $ornt, l$) >
  [What is the missing specification? The appended line starts at a point adjacent to the end point of the previous line in the direction $ornt$. The lines are not allowed to overlap. —SS]
- exception: The transition above creates a new line NL : LineT such that
  $\exists(ln : LineT | ln \in s : \exists(x : PointT | x \in \text{pointsInLine}(NL) \wedge x \in \text{pointsInLine}(ln))) \Rightarrow invalid\_argument$
  [What is the specification for the exception? An exception should be generated if the introduced line overlaps with any of the previous points in the existing path. —SS]
  ****

strt():

- output: $s[0].strt$[What is the missing spec? —SS]
- exception: None

end():

- output: $s[|s| - 1].end$ [What is the missing spec? —SS]
- exception: None

line(i):

- output: $s[i]$
  [Returns the ith line in the sequence. What is the missing spec? —SS]
- exception: $\neg(0 \leq i \leq |s| - 1) \Rightarrow outside\_bounds$
  [Generate the exception if the index is not in the sequence. —SS]

size:

- output: $|s|$
  [Output the number of lines in the path. —SS]
- exception: None

len:

- output: $+(x : \mathbb{N} | x \in [0..|s| - 1] : s[x].len)$
  [Output the number of points on the line. —SS]
- exception: None

translate(Δx, Δy):

- output: Create a new PathT object with state variable s' such that:

$$\forall(i : \mathbb{N} | i \in [0..|s| - 1] : s'[i] = s[i].\text{translate}(\Delta x, \Delta y))$$

- exception: None

Local Functions

pointsInLine: LineT $\rightarrow$ (set of PointT)

pointsInLine (l)

$$\equiv \{i : \mathbb{N} | i \in [0..(l.\text{len} - 1)] : l.\text{strt.translate}$$

($l.orient = N \Rightarrow (0, i)$
$l.orient = E \Rightarrow (i, 0)$
$l.orient = S \Rightarrow (0, -i)$
$l.orient = W \Rightarrow (-i, 0)$) [Complete the spec. —SS]

adjPt: CompassT $\Rightarrow$ PointT
adjPt($ornt$) $\equiv$

$ornt = \text{N}$	$s[\lvert s\rvert - 1].\text{end.translate}(0, 1)[? - - - SS]$
$ornt = \text{S}$	$s[\lvert s\rvert - 1].\text{end.translate}(0, -1)[? - - - SS]$
$ornt = \text{W}$	$s[\lvert s\rvert - 1].\text{end.translate}(-1, 0)[? - - - SS]$
$ornt = \text{E}$	$s[\lvert s\rvert - 1].\text{end.translate}(1, 0)[? - - - SS]$

Generic Seq2D Module

Generic Template Module

Seq2D(T)

Uses

N/A

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	invalid_argument
set	PointT, T		outside_bounds
get	PointT	T	outside_bounds
getNumRow		$\mathbb{N}$	
getNumCol		$\mathbb{N}$	
getScale		$\mathbb{R}$	
count	T	$\mathbb{N}$	
count	LineT, T	$\mathbb{N}$	invalid_argument
count	PathT, T	$\mathbb{N}$	invalid_argument
length	PathT	$\mathbb{R}$	invalid_argument
connected	PointT, PointT	$\mathbb{B}$	invalid_argument

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 bottom of the map and the 0th column is at the leftmost side of the map.

Access Routine Semantics

Seq2D(S, scl):

- transition: $s, scale, nRow, nCol := S, scl, |S|, |S[0]|$ [Fill in the transition. —SS]
- output: $out := self$
- exception: $(scl < 0 \ \vee \ |S| = 0 \vee \ |S[0]| = 0 \vee \ \exists(r|r \in S : |r| \neq |S[0]|) \Rightarrow invalid\_argument$
  [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]

set(p, v):

- transition: $s := s[p.x][p.y] = v$ [? —SS]
- exception: $(\neg$ validPoint(p)$) \Rightarrow outside\_bounds$
  [Generate an exception if the point lies outside of the map. —SS]

get(p):

- output: $s[p.x][p.y]$[? —SS]

- exception: ($\neg$ validPoint(p)) $\Rightarrow$ *outside_bounds*
  [Generate an exception if the point lies outside of the map. —SS]

getNumRow():

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

getNumCol():

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

getScale():

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

count(t: T):

- output: $+(p : \text{PointT} | \text{validPoint}(p) \wedge \text{get}(p) = t : 1)$
  [Count the number of times the value t occurs in the 2D sequence. —SS]
- exception: None

count(l: LineT, t: T):

- output: $+(p | p \in \text{pointsInLine}(l) \wedge \text{get}(p) = t : 1)$ [Count the number of times the value t occurs in the line l. —SS]
- exception: ($\neg$validLine(l) $\Rightarrow$ *invalid_argument*) [Exception if any point on the line lies off of the 2D sequence (map) —SS]

count(*pth*: PathT, t: T):

- output:$+(p | p \in \text{pointsInPath}(pth) \wedge \text{get}(p) = t : 1)$[Count the number of times the value t occurs in the path *pth*. —SS]
- exception: ($\neg$validPath(*pth*) $\Rightarrow$ *invalid_argument*) [Exception if any point on the path lies off of the 2D sequence (map) —SS]

length(*pth*: PathT):

- output: $pth.len \times scale$ [Use the scale to find the length of the path. —SS]
- exception: ($\neg$validPath(pth) $\Rightarrow$ invalid_argument) [Exception if any point on the path lies off of the 2D sequence (map) —SS]

connected(p_1: PointT, p_2: PointT):

- output: $\exists(pth : PathT | ((p_1 = p.strt \wedge p_2 = p.end) \vee (p_1 = p.end \wedge p_2 = p.strt)) \wedge \forall(pt1, pt2 | pt1, pt2 \in \text{pointsInPath}(pth) : pt1 = pt2))$ [Return true if a path exists between p_1 and p_2 with all of the points on the path being of the same value. —SS]
- exception: $\neg(\text{validPoint}(p_1) \wedge \text{validPoint}(p_2)) \Rightarrow$ invalid_argument [Return an exception if either of the input points is not valid. —SS]

Local Functions

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

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

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

validLine: LineT $\rightarrow \mathbb{B}$
validLine(l) $\equiv \wedge(x | x \in \text{pointsInLine}(l) : \text{validPoint}(x))$
[Returns true if all of the points for the given line lie within the boundaries of the map. —SS]
****
validPath: PathT $\rightarrow \mathbb{B}$
validPath(P) $\equiv \forall(x | x \in [0..|P| - 1] : \text{validLine}(P[x]))$
[Returns true if all of the points for the given path lie within the boundaries of the map. —SS]

pointsInLine: LineT → (set of PointT)
pointsInLine (l)

$$\equiv \{i : \mathbb{N} | i \in [0..(l.\text{len} - 1)] : l.\text{strt.translate}$$

($l.orient = N \Rightarrow (0, i)$
$l.orient = E \Rightarrow (i, 0)$
$l.orient = S \Rightarrow (0, -i)$
$l.orient = W \Rightarrow (-i, 0)$) [The same local function as given in the Path module. —SS]
pointsInPath: PathT → (set of PointT)
[Return the set of points that make up the input path. —SS]
pointsInPath(p) $\equiv \cup(s | s \in [0..|p| - 1] : \text{pointsInLine}(p_s))$

LanduseMap Module

Template Module

LanduseMapT is Seq2D(LanduseT)

DEM Module

Template Module

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

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?

We could analyze each interface based on wether the interface was consistent, essential, minimal and other properties

0.1 PointT

The PointT module interface is essential since each method is unique and its implementation and there is no combination os methods that can implement the functionality of another. It is also minimal since each method essentially offers only one service. The module interface is simple altogether so there is not much to critique about this module.

0.2 LineT

LineT uses 4 other modules which illustrates a fan-out uses relation, this is usually undesirable giving us high coupling between LineT and the rest of the modules. Similar to the PathT module, the interface of the LineT module is not essential since end can be implemented using strt.

0.3 Modifications

The design can be modified by finding the shortest path between two points or adding a load module that reads in data to create the 2D map required for the Seq2D constructor. Another possible modification is to define point in line since it's a very small module and can easily be implemented into the line module. Point could be define as a state variable represented by a list and addX,addY, translatePoint methods could be introduced in the module.

The specifications are overall consistent in terms of variable naming and overall consistency.