Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new PointT	real, real	PointT	
getLat		real	
getLong		real	
distance	PointT	real	
latFilter	real	$< real, real >$	
equals	PointT	boolean	

Semantics

State Variables

$x : \mathbb{R}$
$y : \mathbb{R}$

Access Routine Semantics

PointT($lat, long$):

- transition: $x, y := lat, long$
- output: $out := self$

getLat():

- output: $out := x$

getLong():

- output: $out := y$

distanceTo(that):

- output: $out :=$ d such that d is the distance(in km) between current point and that

latFilter(radius):

- output: $out :=< minLat, maxLat >$ such that $\forall(p : PointT | distanceTo(p) \leq radius : p.getLat() \geq minLat \wedge p.getLat() \leq maxLat$

equals(that):

- output: $out := (x = that.getLat()) \wedge (y = that.Long())$

City ADT Module

Template Module

CityT

Uses

N/A

Syntax

Exported Types

CityT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new CityT	string, string, real	CityT	
getCityName		string	
getProvince		string	
getPopDensity		real	
equals	CityT	boolean	

Semantics

State Variables

cityName : String
province : String
popDensity : $\mathbb{R}$

Access Routine Semantics

CityT($city, pro, pop$):

- transition: $cityName, province, popDensity := city, pro, pop$
- output: $out := self$

getCityName():

- output: $out := cityName$

getProvince():

- output: $out := province$

getPopDensity():

- output: $out := popDensity$

equals(that):

- output: $out := (cityName = that.getCityName()) \wedge (province = that.getProvince()) \wedge (popDensity = that.getPopDensity())$

City Position ADT Module

Template Module

CityPostT

Uses

PointT

Syntax

Exported Types

CityPostT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new CityPostT	string, real, real	CityPostT	
getPoint		PointT	
getCityName		string	

Semantics

State Variables

cityName : String
point : PointT

Access Routine Semantics

CityT($city, lat, lon$):

- transition: $cityName, point := city, newPointT(lat, lon)$
- output: $out := self$

getPoint():

- output: $out := point$

getCityName():

- output: $out := cityName$

CityT collection ADT Module

Template Module

GeoCollection

Uses

CityT

Syntax

Exported Types

Graph = ?

Exported Access Programs

Routine name	In	Out	Exceptions
add	CityT		
getCities	string	sequence of CityT	
getAllCities		set of tuple of(string, sequence of CityT)	
isEmpty		boolean	

Semantics

State Variables

s : set of tuple of (string, sequence of CityT)

Access Routine Semantics

add(city):

- transition: $s := \{< str, cities >:< String,$ sequence of CityT $> \mid < str, cities > \in s : (str = getFirstCityLetter(city) \Rightarrow < str, cities || [city] > \mid true \Rightarrow < str, cities >)\}$

isEmpty():

- output: $out := (|s| = 0 \Rightarrow true \mid true \Rightarrow false)$

getCities(firstLetter):

- output: $out := \{< str, cities >:< String,$ sequence of CityT $> \mid < str, cities > \in s \wedge str = firstLetter : cites\}$

getAllCities():

- output: $out := s$

Local Functions

getFirstCityLetter : string $\rightarrow$ string
getFirstCityLetter($city$) $\equiv city[0]$

edge ADT Module

Template Module

Edge

Uses

N/A

Syntax

Exported Types

Edge = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Edge	String, String, $\mathbb{Z}$	Edge	
weight		$\mathbb{Z}$	
from		String	
to		String	

Semantics

State Variables

v : String
w : PointT
$weight$: $\mathbb{Z}$

Access Routine Semantics

Edge($from, to, w$):

- transition: $v, w, weight := from, to, w$
- output: $out := self$

weight():

- output: $out := weight$

from():

- output: $out := v$

to():

- output: $out := w$

graph ADT Module

Template Module

Graph

Uses

Edge

Syntax

Exported Types

Graph = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Graph		Graph	
addEdge	Edge		
adj	string	sequence of Edge	

Semantics

State Variables

adj : set of tuple of (string, sequence of Edge)

Access Routine Semantics

CityGraph():

- transition: $adj :=$
- output: $out := self$

addEdge(e):

- transition: $adj := \{< str, edges >:< String,$ sequence of Edge $> \mid < str, edges > \in adj : (str = e.from() \Rightarrow < str, edges || [e] > \mid true \Rightarrow < str, edges >)\}$

adj(v):

- output: $out := \{< str, edges >:< String,$ sequence of Edge $> \mid < str, edges > \in adj \wedge str = v : edges\}$

search earthquakes Module

Module

SearchEarthquakes

Uses

RedBlackBST, PointT

Syntax

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
searchEarthquakeInCircle	RedBlackBST, PointT, $\mathbb{R}$	sequence of EarthquakeT	

Semantics

State Variables

N/A

Access Routine Semantics

searchEarthquakeInCircleh(bst, location, radius):

- output: $out := \{e : EarthquakeT \mid e \in bst \wedge location.distanceTo(e.getPointT()) \leq radius : e\}$

risk assessemnt Module

Template Module

RiskAssessment

Uses

SearchEarthquakes, GeoCollection, cityGraph, CityPostT

Syntax

Exported Types

RiskAssessment = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new RiskAssessment	RedBlackBST, PointT		
getRisk		$\mathbb{Z}$	
getCity		String	
getFrequency		$\mathbb{Z}$	
getMag		$\mathbb{R}$	
getPoplationDensity		$\mathbb{R}$	
nearestLowerRiskCity	CityGraph	string	

Semantics

State Variables

$earthquakeTree : RedBlackBST < Double, EarthquakeT >$
$cityProv :<>$
$frequency : \mathbb{Z}$
$averageMag : \mathbb{R}$
$populationDensity : \mathbb{R}$
$rating : \mathbb{Z}$

Access Routine Semantics

RiskAssessment(bst, location):

- transition: earthquakeTree := bst,
  cityPov := getCityProv(location, SearchEarthquakes.searchEarthquakeInCircle(bst, location, 100)),
  frequency :=getFrequency(),
  averageMag :=getAverageManitude(),
  populationDensity := getPopulation() ,
  rating := OverallRating(frequency, averageMag, populationDensity)

getRisk():

- output: $out := rating$

getCity():

- output: $out := cityProv[0]$

getFrequency():

- output: $out := frequency$

getMag():

- output: $out := averageMag$

getPoplationDensity():

- output: $out := populationDensity$

nearestLowerRiskCity(graph):

- output: $out := min.to()$ such that $min \in graph.adj(getCity) \wedge RiskAssessment(earthquakeTree, getLocation(min.to()).getRisk() < rating \wedge \forall(e : Edge \mid e \in graph.adj(getCity) \wedge RiskAssessment(earthquakeTree, getLocation(e.to()).getRisk() < rating : e.weight \geq min.weight)$

Local Functions

getLocation : string $\rightarrow$ PointT
getLocation($city$) $\equiv$ $cityPost.getPoint()$

getCityProv : PointT, sequence of EarthquakeT $\rightarrow$ tuple of (string, string)

getPopulation :

getFrequency : sequence of EarthquakeT $\rightarrow \mathbb{Z}$

getAverageMagnitude : sequence of EarthquakeT $\rightarrow \mathbb{R}$

magnitudeRating : $\mathbb{R} \rightarrow \mathbb{Z}$

populationdensityRating : $\mathbb{R} \rightarrow \mathbb{Z}$

OverallRating : $\mathbb{Z}, \mathbb{R}, \mathbb{R} \rightarrow \mathbb{Z}$