Programming Project 2

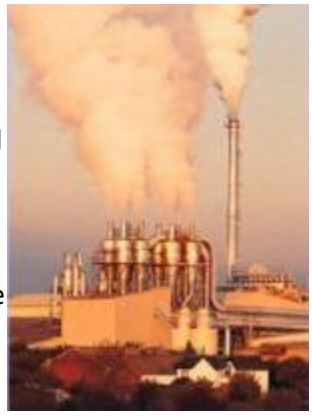
In this project, you'll be writing some code to create various classes that model how pollutants might propagate across a landscape. Although our modeling will be extremely limited in scope, given that we want to keep this project relatively simple, we will examine how a `Force` may be strengthened, weakened, or propagated by a certain type of landscape feature. In particular, after successfully completing this project, you'll have met the following goals:

- An understanding of inheritance.
- An understanding of abstract classes.
- An understanding of interfaces.
- An understanding of method overriding.
- An understanding of two dimensional primitive arrays in Java.

Once you have completed this project, you will be ready to take **Assessment 5**. You will be uploading and using your own code for Assessment 5, so make sure to spend time commenting your code and making it clean and easy for you to work with.

Introduction

Atmospheric dispersion modeling mathematically simulates how pollutants move through the atmosphere. Such research has many important uses, from planning for industrial accidents such as a radiation leak from a nuclear power plant, to designing strategies to control and work around vehicle emissions from major highways. In these sophisticated mathematical models, weather conditions, terrain, exit velocity, obstructions (such as buildings), among other factors, are all considered to examine the possible disperson of a pollutant.



In this project, we will attempt to model a very limited system where some `Force` disperses over a terrain made of `Tile`s. A force will have some initial load, to represent how much of a pollutant will be dispersed in the surrounding tiles/landscape. For this project, we will assume that the force will be wind, although one could modify the dispersion of pollutants in water or soil. We will be limiting our modeling for the sake of simplicity; as such, we will assume that the force contains some starting load of pollutant that will interact with, and dissipate amongst the environment. In this project, the force will propagate across tiles in a specified direction, and we've simplified the implementation of this feature to make it manageable for this project.

This assignment will ask you to write several classes, which we will detail below. Once you've written one or more classes, you can test them against our test cases. When all of your classes are completed,

you can run them against a simulator we've written to show how these various landscape elements impact the propagation of a pollutant in some direction.

You will have to do the following for this assignment:

- Create a project in Eclipse (or your IDE of choice) for this assignment
- Complete the `Force` class
- Complete the `Direction` enumeration
- Complete the `Modifiable` interface
- Complete the `Tile` abstract class
- Complete the `BodyOfWater` class
- Complete the `Building` class
- Complete the `IndustrialBuilding` class
- Complete the `Highway` class
- Complete the `Nature` class
- Complete the `Map` class
- Download the `Tester` class
- Download and run the `UnitTests.java` to check your work
- Download and run the `Main.java` to analyze your code

The diagram below shows the relationship between various classes:

Step 1: Write the following classes

class Force	
The purpose of this basic class is to model a force, such as wind, that is carrying a certain load of a pollutant.	
ATTRIBUTES (please do not change their names)	
double load	The load will record how much pollutant exists.
String name	The name of the pollutant and/or the force.
METHODS (please do not change their names)	

`public Force(double load, String name)`	This constructor will set the attributes to the incoming arguments.
getters/setters for `load` and `name`	Use the *Source* option to automatically generate these in Eclipse.
`public void decay()`	This will reduce the `load` to 80% of its original strength.
`public Force clone()`	This will return a new object that's identical to the current one in terms of `load` and `name`.
`public String toString()`	See the unit tests for formatting. Round the load to an integer using `Math.round()`.

`enum Direction`
The purpose of this enumeration is to code the four possible directions of the force: `NORTH`, `SOUTH`, `EAST` and `WEST`.

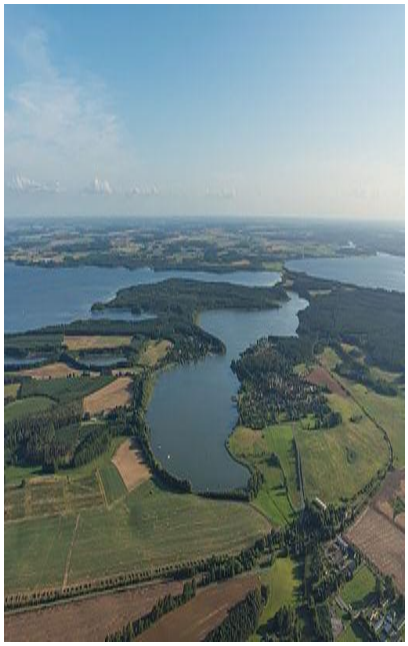
`interface Modifiable`	
The purpose of this interface is to strengthen or weaken a `Force`.	
METHODS (please do not change their names)	
`public Force strengthen(Force f)`	This method will clone the incoming argument and increase the clone's load, finally returning the clone.
`public Force weaken(Force f)`	This method will clone the incoming argument and reduce the clone's load, finally returning the clone.

`abstract class Tile implements Modifiable`

The purpose of this class is to model a small piece of the landscape the pollutant will interact with. We think of the landscape as composed of tiles, much like a Google Map.	
ATTRIBUTES (please do not change their names)	
`int row`	The row number of the tile. A landscape starts the numbering with a row of 0 at the top.
`int col`	The column number of the tile. A landscape starts the numbering with a col of 0 at the left.
`String measurement`	A reading of a `Force`'s load at the current tile, rounded to an integer and stored as a string.
`int PEOPLE_WASTE`	This constant represents a multiplier that acts on a pollutant load to contribute human pollution. Set it to 2.
`int CARS_WASTE`	This constant represents a multiplier that acts on a pollutant load to contribute car pollution. Set it to 5.
METHODS (please do not change their names)	
`public abstract boolean canPropagate()`	This method will determine whether or not the pollutant can propagate past this tile (for example, a large building would prevent propagation).
getters/setters for the non-constant attributes	Use the *Source* option to automatically generate these in Eclipse.
`public String toString()`	See the unit tests for formatting.
`public Force weaken(Force force)`	Implement the weakening of the force as described above by calling the `decay` method.

`class BodyOfWater`

The purpose of this class is to model how pollutants interact with a body of water, such as a lake. It extends the `Tile` class. You do not need to write a constructor for it.

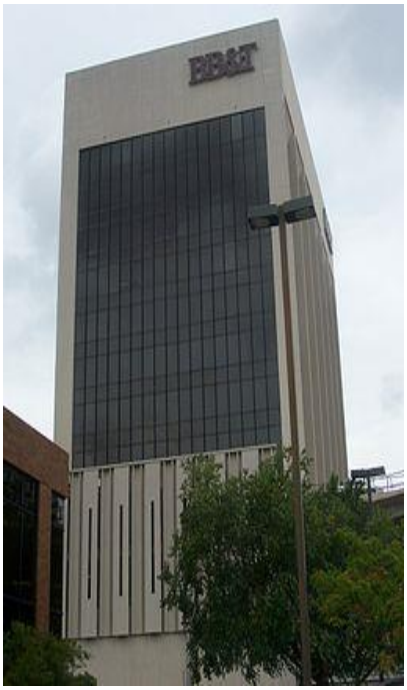


METHODS (please do not change their names)

`public Force strengthen(Force f)`	In our simulation a body of water will preserve the load of the force upon strengthening.
`public boolean canPropagate()`	A force can strengthen over a body of water.

`class Building`

The purpose of this class is to model how pollutants interact with a building. It extends the `Tile` class.

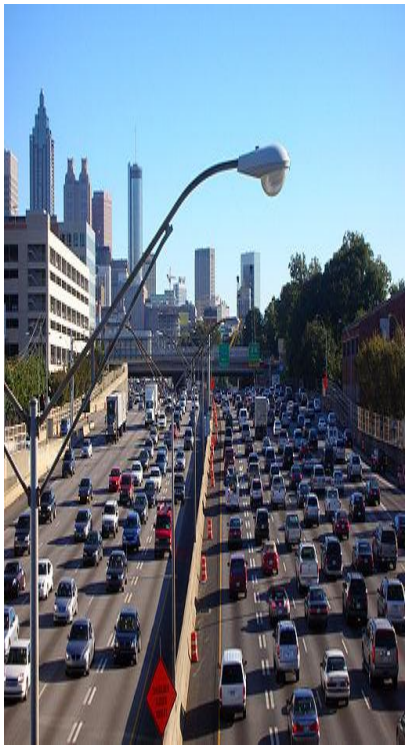


ATTRIBUTES (please do not change their names)	
`int people`	The number of people who occupy the building.
`int cars`	The number of cars which are parked in or near the building.
`int height`	The height of the building.

METHODS (please do not change their names)	
`public Building(int people, int cars, int height)`	The constructor initializes the attributes to the incoming arguments.
getters/setters for all attributes	Use the *Source* option to automatically generate these in Eclipse.
`public Force strengthen(Force force)`	The returned force's load will be incremented by the pollutants created by the number of cars and people.
`public boolean canPropagate()`	If the building's height is less than 100, it can propagate the force.

class IndustrialBuilding	
The purpose of this class is to model an industrial building that can add additional pollutants into the atmosphere. It extends the `Building` class. 	
ATTRIBUTES (please do not change their names)	
double emissions	This attribute keeps track of the amount of emissions a factory emits.
int EMISSIONS_CONSTANT	This constant represents a multiplier that acts on a pollutant load to contribute factory pollution. Set it to 10.
METHODS (please do not change their names)	
public IndustrialBuilding(int people, int cars, int height, double emissions)	This constructor sets all attributes of the object (including inherited ones).
public Force strengthen(Force force)	The returned force's load will be incremented by the pollutants created by the number of emissions,

	after strengthening the force through the parent's method.
`public boolean canPropagate()`	This method will return true.

`class Highway`	
The purpose of this class is to model a highway. It extends the `Tile` class. 	
ATTRIBUTES (please do not change their names)	
`int carDensity`	This attribute keeps track of the density of cars on the highway.
METHODS (please do not change their names)	
`public Highway(int carDensity)`	This constructor sets all attributes of the object.
`public Force strengthen(Force force)`	The returned force's load will be incremented by the pollutants created by the number of cars on the highway.

public boolean canPropagate()	This method will return true.

class Nature	
The purpose of this class is to model a block of nature such as a field or forest. It extends the `Tile` class.	
ATTRIBUTES (please do not change their names)	
int treeDensity	This attribute keeps track of the density of trees, as a percentage.
METHODS (please do not change their names)	

public Nature(int treeDensity)	This constructor sets all attributes of the object.
public Force strengthen(Force force)	In our simulation nature will preserve the load of the force upon strengthening.
public Force weaken(Force force)	In our simulation nature will reduce the strength of a force proportional to the tree density. The load of a force cannot be negative.
public boolean canPropagate()	This method will return true when the tree density is less than or equal to 50%, false otherwise.

class Map	
The purpose of this class is to model the landscape of `Tile`s, similar to how Google uses tiles in their applications.	
ATTRIBUTES (please do not change their names)	
Tile[][] map	This attribute uses a two dimensional array to store the tiles of the landscape.
int freeRow	A map starts out empty; this attribute stores the row a new tile can be placed on.
int freeColumn	A map starts out empty; this attribute stores the column a new tile can be placed on.
METHODS (please do not change their names)	
public Map(int width, int height)	This constructor initializes the map to the incoming size.
public boolean addTile(Tile tile)	Adds the incoming tile to the next free location (starting in the top left corner and filling up each row before moving to the next one), updating the three attributes of the `Map` class accordingly. If the map is already full, this method returns false, otherwise if it successfully placed the tile, it returns true.

public Tile getTile (int row, int col)	Returns the tile at the specified row and column.
public Tile[] getNeighbors(Tile tile, Direction direction)	This returns an array of all the neighbors of the tile in the direction specified; a direction can return up to three neighboring tiles. Any tile that is touching the incoming tile in at least one point in the direction specified is considered a neighbor.

public void propagate(Force force, int row, int column, Direction direction)	This method has been completed for you below, because it uses *recursion*, a topic we will discuss in the future. Notice how the method calls itself inside its own body.
	
	`Tile tile = getTile(row,column);`
	
	`Force weaken = tile.weaken(force);`
	`Force strengthen = tile.strengthen(force);`
	`tile.setMeasurement((strengthen.getLoad() + weaken.getLoad())/2);`
	`Tile[] neighbors = getNeighbors(tile, direction);`
	`force.setLoad((strengthen.getLoad() + weaken.getLoad())/2);`
	
	`//recursion below`
	`for (int i = 0; i < neighbors.length; i++)`
	`  propagate(force,neighbors[i].getRow(),neighbors[i].getColumn(), direction);`
	
	Recall, that the `tile`'s `strengthen` method is something defined in the `Modifiable` class.

public String toString()	See the unit tests for formatting of this method's return value.

Step 2: Testing Your Code For Functionality and Elegance

Once you have written your classes, you should download and run the unit tests to check for basic correctness. Afterwards, you should run your code on our Main.java simulator to make sure it works properly when tested as a system.
The unit tests will, in addition to correctness, measure the following elegance metrics:

Coding style and readability	Make sure you use descriptive variables names, proper indentation, etc.
Class, attribute, and method documentation	Make sure you comment the purpose of each of these items.
Adherence to abstraction barriers	• Public methods are used to access private class attributes, rather than accessing them directly. • The `propagate` method is called, rather than checking for individual types of objects.
Code elegance	• `getNeighbors` should be simplified to be less than 150 lines of code.
Inheritance correctness and safety	• All child classes appropriately use the parent class' constructors. • The `@Override` tag is used in all the appropriate places. • All appropriate attributes are private.

Step 3: Running your code as a simulation

Now that you've successfully completed and debugged your code, you can use your classes to simulate what might happen in the event of a cruise ship smoke stack propagating over a shoreline city, by running `Main.java`. You'll be able to see how the pollutant changes depending on what kind of terrain it passes over.

Feel free to play around with the simulator code, for example:

- You can change the types of tiles in the map, as well as its size.
- You can modify the initial load of the force.

Step 4: Preparing for Assessment5

Create a copy of all of your files in a separate directory (preferably a new project in Eclipse). Then, modify those files to get them to pass the unit tests of Sample Assessment 5.