Introductory Programming, Fall 2014 exam

You may have heard about the *water cycle* from your biology or geography classes. The concept is that water follows a natural path through nature: clouds send water as rain to the land. The water seeps from the land through rivers to lakes and oceans, and the sun heats the water bodies to make the water dissipate into the air as vapour, creating clouds in the process (and then we repeat). The task at hand is to emulate the water cycle in Java.

Practical information

The assignment is released on Thursday, the 11th of December 2014, at 10AM, and is to be handed in no later than 24 hours later through LearnIt, unless special arrangements have been made in advance. The deliverable *must* be an Eclipse project, saved as a zip file using the Eclipse menu item File —> Export, General —> Archive file, and the project name *must* have your own name in it, e.g. “John_Doe_water_cycle_exam” or something like that. No report is to be handed in - only code.

The exam is individual. You are *not* allowed to work in groups. And I shouldn’t have to tell you that plagiarism, of course, is strictly prohibited.

Finally, remember to submit the ITU front page as well. The exam office has stated that they prefer to have it within the zip file you’re submitting if possible.

Good luck!

Part 1: modelling the system

For the purpose of creating the water cycle, we need the following types in our model:

Classes

WeatherSystem - this is a hub for tying the whole system together.
Sun - this is, well, the sun. There’s only one of these.
Cloud - these are created when the sun vaporises water from a lake into the air.
Lake - this is a body of water.
Soil - this is land. Water landed on soil will eventually flow to a nearby lake.

Abstract classes

SkyObject - this is a superclass for things in the sky (sun and clouds).
GroundObject - this is a superclass for things on the ground (lakes and soil).

Interfaces

WeatherPart - all parts of the model (the sun, clouds, lakes and soil) implement this interface.

The system starts out with the sun and any number of soil patches and lakes, and perhaps some clouds as well. Time will then progress, and for each unit of time, each part of the system will interact with the other components.

The flow of the system works, in detail, as follows:

- The sun shines on a ground unit (soil or lake). If this has any water in it, part (or all) of this water evaporates.
- The vaporised water forms a new cloud.
- The cloud eventually rains on soil or a lake. When a cloud has no more water in it, it disappears.
- Water in the soil will slowly flow to the adjacent lake.

First, create the *WeatherPart* interface. This interface has but one method, *progress*. This is called on each part of the system when one time unit passes.

Create the two abstract classes (*SkyObject* and *GroundObject*), implementing the WeatherPart interface. SkyObject has a field called *recipient* and a setter for this field, as well as a *hasRecipient* method to see whether the sun or cloud has a recipient selected. GroundObject has a field *water* with a getter, as well as a *receiveWater* and *sendWater* method, representing the ground object receiving water from and sending water to another part, respectively. The *progress* method should remain abstract, as this is to be implemented by the concrete classes themselves.

Now create the *Sun* class, which extends the *SkyObject*. This class has a field *waterInAir* that describes the amount of water the sun extracted from its recipient ground object, as well as a getter. The *progress* implementation calls a method *shine* with an amount of water (it's up to you whether it's a random or set amount), and this method extracts the water from the recipient (using the recipient's *sendWater* method) and increases the water in air correspondingly. Also, there is a method *cloudFormed* that is called by the weather system when a cloud is created; this method resets the water in air to zero.

Next is the *Cloud* class, which also extends *SkyObject*. It has a *size* field which corresponds to the amount of water in the cloud, and a corresponding getter. The *progress* method calls a method *rain* with an amount of water (again, how much water is your call to decide); the *rain* method then uses the recipient's *receiveWater* method to send water to it. The size of the cloud shrinks correspondingly. Also, an *isEmpty* method will report if the cloud has no more water in it (its size is zero).

Then create the *Lake* class, which extends *GroundObject*. It's quite simple: it has a constructor that takes the initial amount of water in the lake, and an *isDry* method that returns true if there is no water in the lake. The *progress* method actually does nothing, as all the lake does is remain silently stoic as a monument to peaceful calm and serenity - all interaction with the lake happens by the initiative of one of the other parts of the system.

Next up is the *Soil* class, also extending *GroundObject*. Soil requires access to an adjacent lake and thus its constructor requires that one such lake is provided - it is saved to the field *closestLake*. The *progress* method will, if the soil contains water, take some of this water and channel it to the lake using the *sendWater* method on the soil and the *receiveWater* method on the lake.

Finally, we have the *WeatherSystem* class. This is the one that binds everything together. This class has the following members:

Fields

- weatherParts: a list of all parts in the system (sun, clouds, lakes, soil).
- time: the current time in the system (in time units).
- latestLake: a reference to the last lake created - useful when creating new Soil objects that need an adjacent lake.
- sun: a reference to the sun (which is a special part, as there's only one of these).

Methods

- getWeatherParts: getter for the field.
- getTime: getter for the field.
- getSunAndClouds: getter for the SkyObject objects in the weatherParts.
- getSoilAndLakes: getter for the GroundObject objects in the weatherParts.
- createSun: *private* method for creating the sun (remember: only one sun can exist in the system!).
- createCloud: creates a new cloud with a specified amount of water in it.
- createLake: creates a new lake with a specified amount of water in it.
- createSoil: creates some new soil, using the latest created lake.
- passTime: calls the progress method of each part in the system, and advances time with one unit.

Part 2: setting up rules

Now we've set up the basics. However, we need to enforce that the system works as intended.

First, create two private methods in the *WeatherSystem* class, *beforeProgress* and *afterProgress*. These are to be called in the beginning and the end of the *passTime* method, respectively.

The *beforeProgress* method should enforce that all sky objects have a recipient set, if one such exists - i.e., if soil or lakes has been created. Every time a time unit passes, a new recipient amongst the available ground objects should be set at random for each sky object.

The *afterProgress* deals with cloud management. First, all clouds have their *isEmpty* method checked; if a cloud is found to be empty, it is to be removed from the system. Then the sun is checked for surplus water in the air. If such water exists, a new cloud is created and put into the system, and the water used to create it is reset (using the *cloudFormed* method on the sun object).

You should also check for the following things:
- that one and only one sun exists in the system at all times.
- that no more water is removed from a part of the system than it actually contains.
- that it is not possible to create soil without an adjacent lake.
- that the system does not crash if no ground objects are available as sky object recipients.
- that it is not possible for a part to contain a negative amount of water.
- that it is possible for both the sun and a cloud to effect the same recipient in one time unit.

Throw a custom exception, *WeatherException*, if the system needs to tell the user that one of the rules have been broken (and there's no elegant way to fix it).

Our simple model of the system is intended to be a closed system, and as such, once the parts have been set up in the system, the total amount of water in the parts put together must always stay the same. Create a method *getTotalWaterInSystem* in the WeatherSystem class that shows the grand total of water in the system - water in the lakes and soil, the sizes of clouds, and evaporated water in the air, all put together. Then, create a class *WeatherSystemTest* that sets up a system, passes 1000 units of time, and uses an assertion to check that the total at the start of the test (after the parts were set up) is identical to the total at the end of the test.

Part 3: the interface

Now it's time to actually have some interaction with the system. You can *either* provide:

1) a command line interface using Scanners and System.out or
2) a graphical user interface using JavaFX

Whichever you choose (yes, showing off by doing both is allowed, if you have plenty of time), your interface *must* provide the user with the following options:

- create clouds, lakes and soil
- see the current contents of the system, as well as the current time unit and total water in it
- pass time

Part 4: extra stuff

If you for some reason have done *all* of the above and still have some time left, here are some suggestions for improvements:

- A save and load function, using a file (or a database, if you're up for that kind of challenge).
- Unit tests.
- Having the parts set and move in a coordinate system, so each part has an X and a Y coordinate, and e.g. a cloud only rains on some soil if it's actually on top of it (occupies the same X/Y space).
- Wrap it in an Android app.

Seriously, only do the above things (or others) if you're *really* sure that the rest of the assignment has been properly done.

Tips and tricks

- The assignment is intended to be as unambiguous as possible; but in case of doubt, use common sense and explain your choices in the comments.
- Adhere to the three pillars of object-oriented programming: encapsulation, inheritance and polymorphism. The assignment is designed to test your grasp of all of these.
- Test edge cases. Make sure your system never crashes - if an exception is thrown, it should be entirely on purpose.
- Use packages to structure your project.
- If you for some reason cannot do the entire assignment, hand in what you've got - something is better than nothing. However, make sure it compiles, otherwise you're off for a bad start.
- Finally, this class diagram may provide you with some inspiration for the system.