Lab — Abstract Data Types

Polytech Montpellier — IG3

Interfaces are the best way to program Abstract Data Types in Java. We will apply what we learned to propose a graph library. For this purpose, you will work in teams of two persons, using the *GitHub* code repository to give to communicate your work to other groups.

PART 1 - The bases of a package for representing graphs

(1) You've just started to see the Graph formalism (some of you for the first time), which can be used to model a very large number of engineering problems. Gather the first elements you've seen on this topic to propose a **`graph`** package containing:

- a **`Vertex`** class (allowing to store an Object associated to it)
- an **`Edge`** class or interface (which is better ?) that you will then refine into a class `DirectedEdge` and a class `UndirectedEdge`
- a **`Graph`** interface containing the necessary methods to be used for solving a graph problem (write no implementation yet for this interface!). For now, allowed data types are only Java primitives types, arrays of those, `Vertex` and `Edge`.
- a **`TestGraph`** class that just declares and initializes a `Vertex` and an `Edge`. These files should be working fine.

Your files should be in `src` and `bin` folders. Classes for vertices and edges should contain constructors so that new instances can be generated easily. It's recommended that vertices and edges know the graph to which they belong.

PART 2 - Packing up and distributing your package

(2) Document your classes and interfaces (use tags), then generate the documentation of the package in a `doc` folder.

(3) Pack up the `bin` and the `doc` (only those files) into a `graph.jar` file to distribute your library. In the manifest file, indicate that TestGraph is the Main class.

(4) Create a code repository named `graphJava` on GitHub hosting only the two following elements:

- ➡ you library (`graph.jar`)
- ➡ a README explaining in very few instances what we've got here, mentioning the name of the coders, the date, the name of the module and the school.

PART 3 - Online documentation of your package

(5) Put the javadoc of your library online using GitHub Pages:

(1) carefully follow guidelines for such an operation (for instance, see https://vaadin.com/blog/-/blogs/host-your-javadoc-s-online-in-github).

(2) When you have your index.html («Hello World») file in your `username.github.io` and it can be seen online, then in Linux go to your clone of `username.github.io` and create a folder `graphJava` inside (should be the same name as the code repository on GitHub), add it (`git add graphJava`)

(3) Put the latest javadoc files inside this new folder (`cp ../...../doc/* graphJava`) then add them to git (`git add --all`)

(4) Commit this new files and push online (should go into the repository called `username.github.io` in the `graphJava` folder).

(5) Consult at : http://username.github.io/graphJava/

(6) Indicate the address of this online documentation in the `README.md` file of your local code repository, then push on Github, check to see whether everything is

fine. It should look like this: https://github.com/vberry/essai with a working html address for the documentation.

PART 4 - Proposing an Implementation of the Graph interface

(6) Choose a team name (as you already guessed I'm sure, today's theme for names is «animals»). Add an entry in the Moodle resource of session 6 (here for G1 and for G2) to the name of your animal and precise the address of the GitHub page of the package you just developed.

(7) **Ask** the lecturer for the name of another team, then get the latest working version of **their** `graph` package (`git clone` or simply download from their gh page). Have a look at the documentation (online or included within their `jar` file (just extract this part, not the bytecode files)

(8) Run their library with the `-jar` flag (it should rune the `TestGraph` class). In case of problem detected in the graph library you fetched, inform the design team (create an issue in their gh repository) so that they patch their library, update their code & doc repository, then download the new version to go on in your tests. This procedure should be repeated later when needed.

(9) Choose a physical representation for a graph and propose a class implementing their `Graph` interface (adjacency matrix, incidence array, ...) that relies **only** on primitive data types and arrays of these types (no Vector nor any Java Collection allowed yet).

(10) Write a `TestGraphImpl` class that declares a graph, initialize it with a call to the constructor of your implementing class, then adds 3 vertices and 2 edges between them. The code should then ask for the list of vertices (obtaining an array), or be able to check whether there is an edge between this and that vertex.

(11) Once you're happy with your work, package the implementation and `TestGraphImpl` inside a `jar` file such as `AdjacencyListsGraph.jar`, then test that it is runnable (you should need to indicate `-cp Graph.jar`).

(12) Document your code if not already done and generate the documentation in a doc folder.

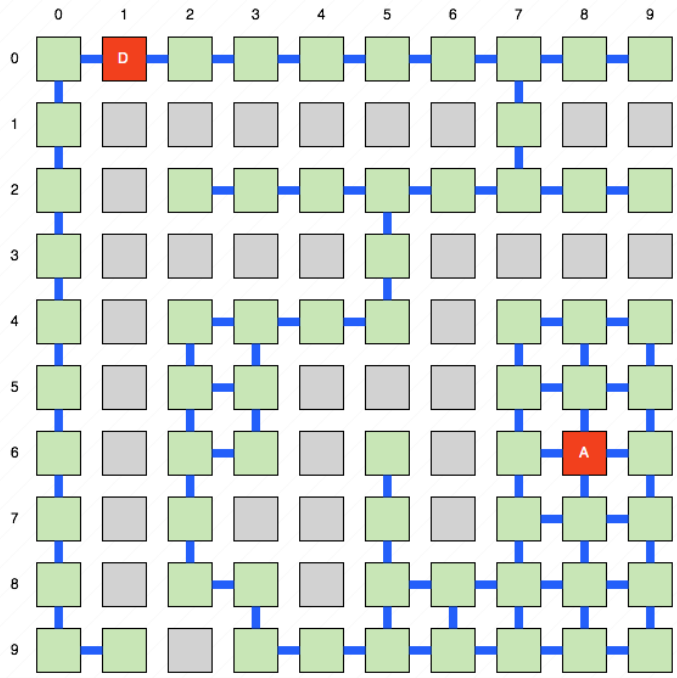
(13) Now you will propose your implementation as an addition to the repository where you found the Graph interface:
   (1) Fork their repository (see https://help.github.com/articles/fork-a-repo/ for this), this should create a new repository for you (attached to your gh account)
   (2) Clone this new repository in a new folder on your computer, then add your files in this folder and its subfolders (src, doc and bytecodes separately), test that the integration is working fine (java execution) and push online.
   (3) Issue a pull request on the repository of the other team (see https://help.github.com/articles/using-pull-requests/) so that they consider the modifications of their repo that you propose in your forked repo.

PART 5 - Representing a maze by a graph

A labyrinth can be represented as a graph with an edge from cells are adjacent and communicating (see the picture). We restrict our programs to the case of cells connected horizontally and vertically. We will develop here a recursive algorithm that explores a maze to find a path between a departure and a final cell.

(14) In a new project/package, start a `Maze` class, referring to the `Graph` class (should we inherit from it or just use it preferring composition over inheritance?). The labyrinth as

an height and a width measured in number of cells.



(15) Implement what is necessary to represent these cells objects: a `Cell` mainly needs to encode its coordinates and indicate of which kind it is:

- `DBox` (*departure box*), red cell in the picture
- `ABox` (*arrival box*), also red in the picture
- `EBox` (*empty box*) for cells where we can walk (green cells in the picture)
- `WBox` (*wall box*) represented in gray in the picture

(16) Propose a `Game` class declaring a 3x3 `Maze` whose departure box is at (0,0), arrival box is at (2,2) and manually put some edges between cells of the labyrinth so that there are at least two paths between these cells.

(17) Propose a toString() method for the Maze class so that you can have an ASCII description of the labyrinth such as:

```
D . .
. O .
. . A
```

Use this method to display the maze you've created in `Game`.

PART 6 - Exploring the labyrinth

(18) Now is it possible for a cell to know its neighbors? For instance, cell (2,2) has 2 neighbors: (1,2) and (2,1). If this is not possible, add a suitable mechanism. We need this as we want to explore the labyrinth going from a cell to a neighboring one.

(19) When exploring the maze we want to know whether we already walked on a cell that we encounter (so as not to go round and round ;). Add what is necessary to your cells to know whether they have been explored or not.

(20) Implement an algorithm `findPath` recursively exploring the labyrinth from the departure cell until the arrival cell is reached (or you obtain a conclusion that no path exists between these two cells (this could be for some labyrinth). Don't forget to mark the cells as visited when you step on them, and to unmark them later if you come back from a dead end to try an alternative path. If you find a solution path in the given maze, print the labyrinth by replacing dots («.») by «@» signs.

(21) Write a variant `findAllPath` of the exploration algorithm that prints all possible paths from D to A (it just does not stop when encountering a correct solution.

(22) Write a further variant `findShortestPath` that outputs the shortest discovered path (you just need to store the shortest path discovered previously when looking for a new path).