# GraphStream: a Dynamic Graph Library

Frédéric Guinand

email:Frederic.Guinand@univ-lehavre.fr

web: http://litis.univ-lehavre.fr/∼guinand

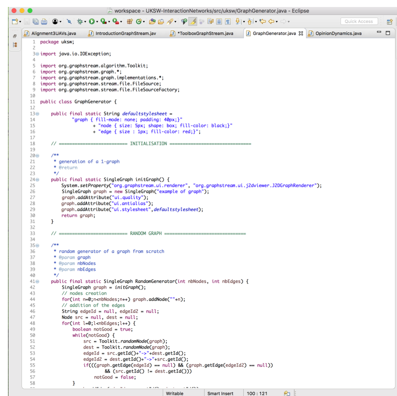
## 1 Introduction

GraphStream is a java library for generating and visualizing dynamic graphs. It offers an API for implementing algorithms for the manipulation and the analysis of dynamic graphs. In few lines you can: create a graph and display it using a specific style (CSS-like). You can also create a graph from a formatted text file (DGS files, a text-based description of the graph, nodes and links). The layout of the graph is automatic but you can force the engine to follow some position indications if needed (we will see that for euclidean graphs and grids/toruses for instance).

This document aims at introducing you this library that will be used during the whole course for illustrating the lectures and for developing examples.

## 2 Programming Environment

For the labs we will consider eclipse as your IDE and GraphStream as your dynamic graphs library[1]. Thus, all the examples will be developed on eclipse.



http://graphstream-project.org

### 2.1 GraphStream jar files

You can get the java archives of GraphStream, by downloading them on:

- http://graphstream-project.org or on

Only three jar files are needed for using graphstream:

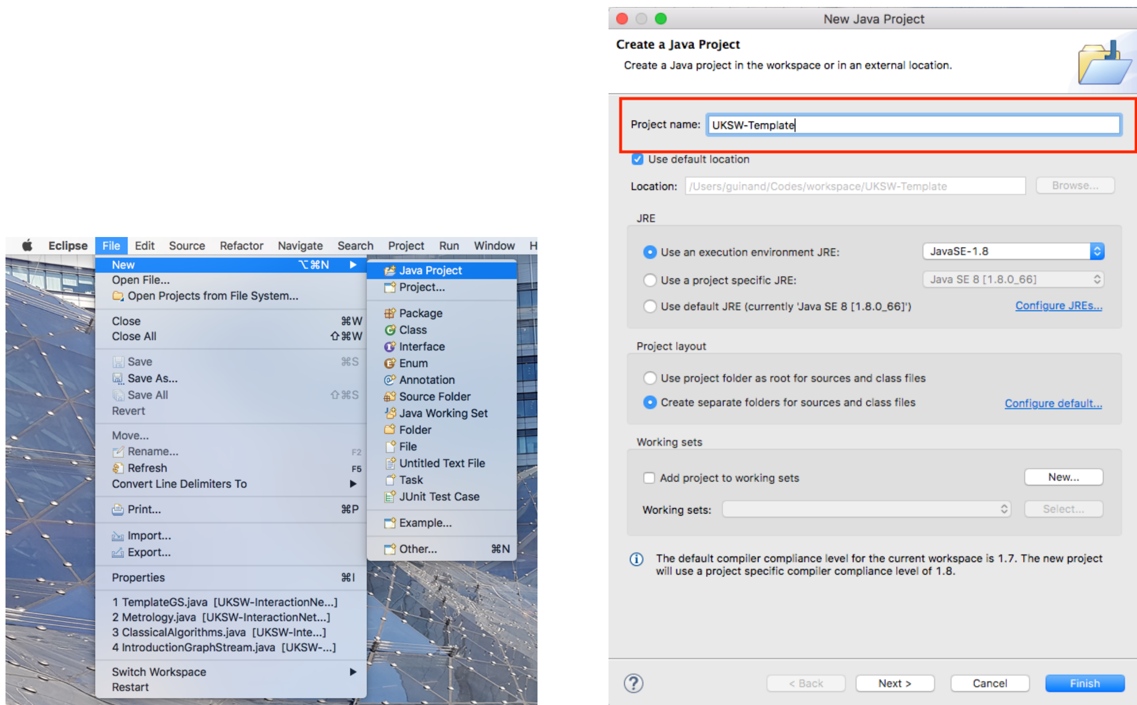
- gs-core-1.3.jar – the core jar file including all the fundamental graph classes

[1] you are free to use another IDE as soon as you know how to use it for java programming and for using external jar files in your java classes

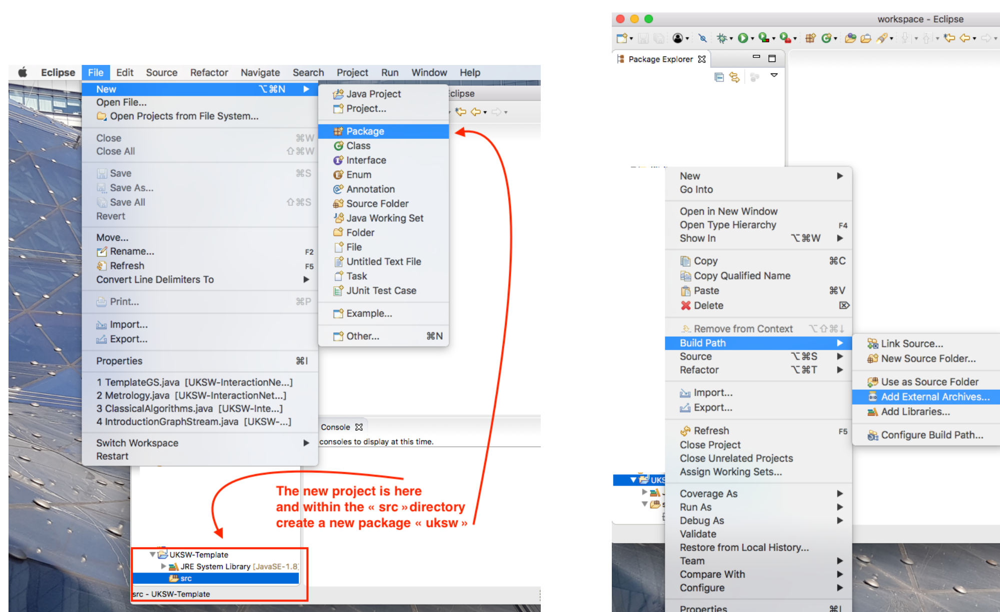
- gs-algo-1.3.jar – an archive including classical graph algorithms (shortest part, spanning tree, etc.)
- gs-ui-1.3.jar – this archive contains the graphstream classes necessary for displaying a graph and interacting with it

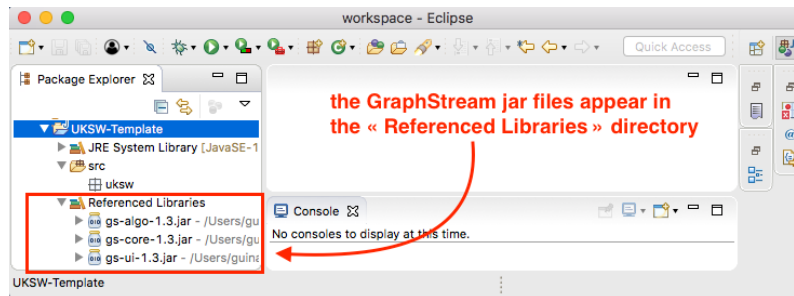
## 2.2 Setting up a new GraphStream project

Create a new java project:




Create a new package and add the GraphStream jar files as external archives:






We are now ready to start!!

# 3 API Presentation

## 3.1 Fondamental Elements

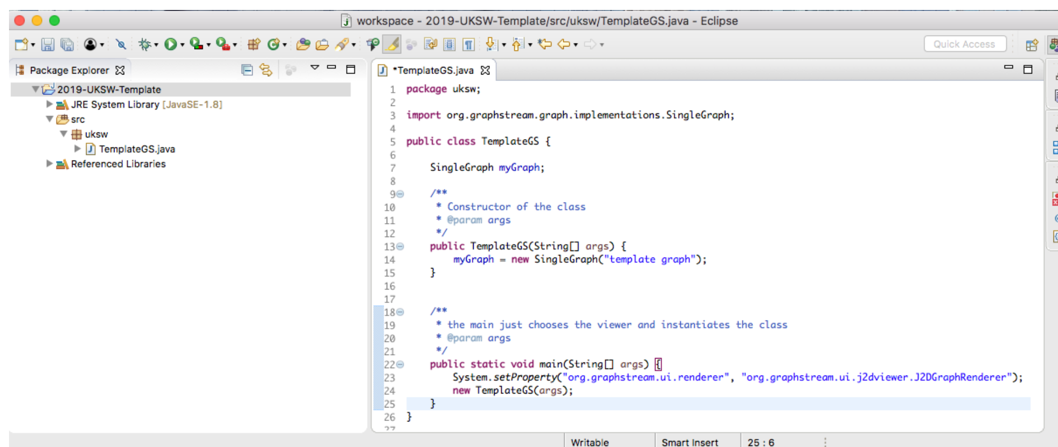
Using GraphStream, a 1-graph, a graph in which two vertices are only connected by one edge or two arcs in opposite direction, is an instance of the object `SingleGraph`. A vertex in GraphStream is an instance of the `Node` object and an edge/arc is an instance of the `Edge` object.

For creating/instantiating any of these objects it is necessary to import them:

```
import org.graphstream.graph.implementations.SingleGraph;

import org.graphstream.graph.Node;

import org.graphstream.graph.Edge;
```

For starting:

```
package iwocs;

import org.graphstream.graph.implementations.SingleGraph;

public class TemplateGS {

   SingleGraph myGraph;

   public TemplateGS(String[] args) {
        myGraph = new SingleGraph("template graph");
   }


   /** the main just chooses the viewer and instantiates the class */

   public static void main(String[] args) {
       System.setProperty("org.graphstream.ui.renderer",
            "org.graphstream.ui.j2dviewer.J2DGraphRenderer");
       new TemplateGS(args);
   }
}
```




## 3.2 Graph/Node/Edge Creation

- `SingleGraph graph = new SingleGraph("graph")`

  creates an instance of a SingleGraph. It contains neither vertices nor edges. For adding new nodes and edges, the API offers several methods. The main ones are:

- `Node thenode = graph.addNode("idf")`

  returns a Node which identifier is `idf`. **Two distinct nodes cannot have the same identifier**. For verifying the identifier of a node just call the method `thenode.getId()` on it.

- `Edge theedge = graph.addEdge("u--v", u.getId(),v.getId())`

  returns an Edge which identifier is `u--v`. This edge is a non oriented link (thus not an arc). For adding an arc, the method should add an additional boolean parameter positioned at true: `Edge arc = graph.addEdge("u-->v",u.getId(),v.getId(),`**true)**. It is also possible to create an Edge using directly the references of the Node extremities: `Edge arc = graph.addEdge("u-->v",u,v,true)`, where `u` and `v` are Node references.

- For removing a give Node from the graph: `graph.removeNode(nodeIdentifier)`

- For removing a given Edge: `graph.removeEdge(edgeIdentifier)`

- For removing an attribute: `nodeOrEdge.removeAttribute("nameOfAttribute")`

- For testing the existence of an attribute: `nodeOrEdge.hasAttribute("nameOfAttribute")` return true if the attribute exists and false otherwise

## 3.3 Graph Display

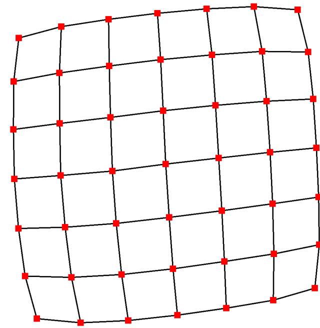
For displaying the graph, a simple call to a method is enough. The method accepts one optional boolean parameter.

```
graph.display(boolean)
```

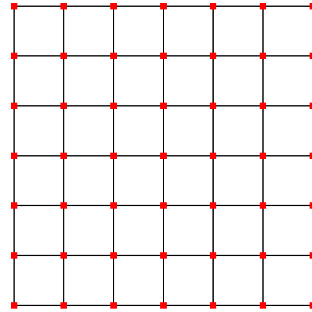
If positioned at true (the default value), the layout will be automatic which is very convenient if you don't have any constraints on the position of your Nodes. If the boolean parameter is positioned at false, you will have to fix the position of your Nodes by adding their coordinates:

```
Node n = graph.addNode("u");
n.addAttribute("x",230);
n.addAttribute("y",400);
```

For instance, for a grid this can lead to the following displays:



`graph.display(true);`



`graph.display(false);`

## 3.4 Exercise 1

Modify the TemplateGS.java file in order to:

1. add four nodes $\{A, B, C, D\}$ and the edges $\{A, B\}$, $\{B, C\}$, $\{C, D\}$ and $\{D, A\}$, and display this graph using automatic display.
2. add Attributes "x" and "y" to each node and choose a display that respects these coordinates.

## 3.5 Node and Edge Attributes

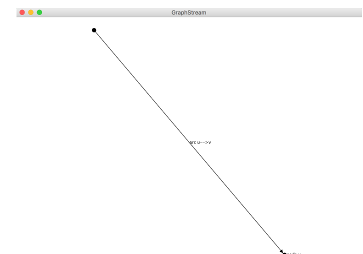
It is possible to attach to each Node/Edge some information. It is done through the addition of an Attribute given as a pair (key, value). For instance if you want to attribute a cost to an egde $e$:

```
e.addAttribute("cost",10.5);
```

There are several attributes that are already defined in GraphStream: `ui.label, ui.style, ui.stylesheet, x, y`.

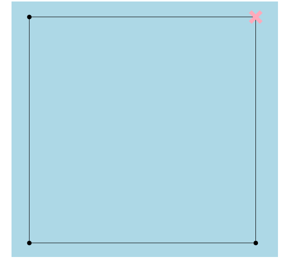
For displaying the name given to each Node and Edge you just have to set the "ui.label" attribute to these elements. Example:

```
SingleGraph sg = new SingleGraph("first");
Node u = sg.addNode("u");
Node v = sg.addNode("v");
v.addAttribute("ui.label","node v");
Edge e = sg.addEdge("u-->v", u, v, true);
e.addAttribute("ui.label","arc u-->v");
```



It is possible to modify the style of the graph by modifying the attribute ui.stylesheet. But it is also possible to apply some style to the vertices and/or of the edges by modifying the value of the attribute `ui.style`. Examples:

```
shortStylesheet = "graph { fill-color: lightblue; "
                + "padding: 40px; }";
graph.addAttribute("ui.stylesheet",shortStylesheet);
nodeStyle = "fill-color:#ffaabb; shape:cross;"
                + "size: 30px;";
Node n = Toolkit.randomNode(graph);
n.addAttribute("ui.style",nodeStyle);
```



But it is also possible to define the style of the whole graph as a global String variable:

```
protected String myStylesheet =
          "graph { fill-color: white; padding: 40px; }"
          + "node { size: 10px; shape: box; fill-color: red; }"
          + "edge { size : 2px; fill-color: black;}";
```

### 3.6 Exercise 2

Starting from the graph created in Section 3.4:

1. change the style of the graph such as using "lightblue" as the background color
2. change the style of on randomly chosen node such that its shape is a cross, its size is 30 pixels and its color is another color defined in hexadecimal.

## 4 Save/Restore a Graph to/from a File

GraphStream offers the possibility to read and display a graph from a dgs formatted file and to save a graph to a file into the same format. GraphStream first purpose was to allow a dynamic representation of graph. For that, a dynamic graph is considered as a serie of events modifying the graph (nodes and/or edges addition/deletion/modification). From that results the text-based DGS format.

### 4.1 DGS format

- `an, cn, dn` is for adding, changing and deleting a node
- `ae, ce, de` is for adding, changing and deleting an edge
- `cg` is for changing some graph properties (attributes for instance)
- `st i` corresponds to the begining of the $i^{th}$ step (each step corresponds to a set of "simultaneous" events)

The full specifications can be found at:
**http://graphstream-project.org/doc/Advanced-Concepts/The-DGS-File-Format/**

Examples:

- `an n3 ui.label="the node" x=34 y=56`

→ add to the graph a Node with "n3" as identifier, "the node" as name (which will be displayed aside the node) and with coordinates 34 for the x axis and 56 for the y axis

- `ce e34 ui.style="size:3px; fill-color:red;"`

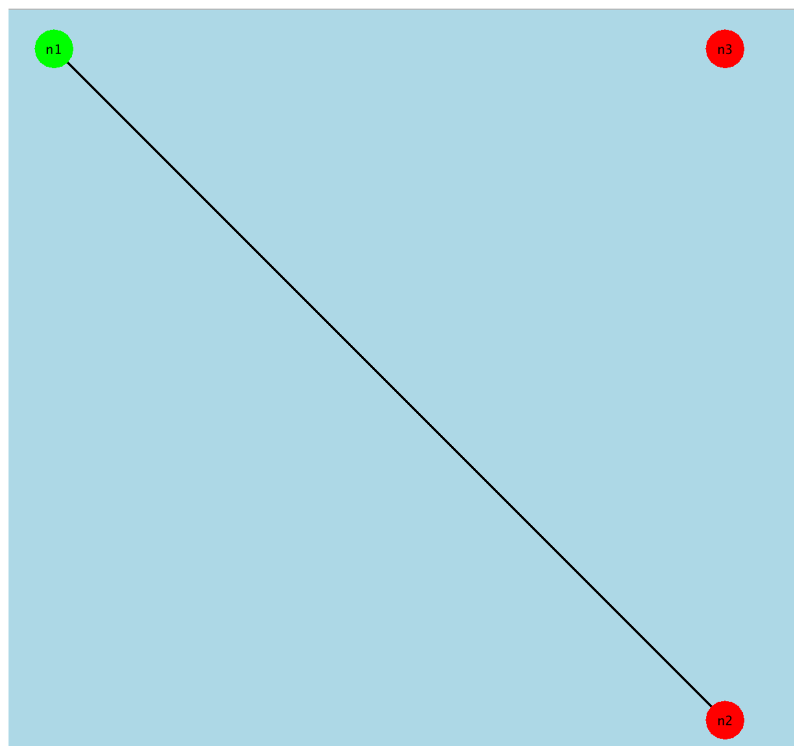
  → change the attribute "ui.style" of the edge which identifier is "e34". The new style specifies that edge will be red with a thickness equals to 3 pixels

- `cg ui.quality=true ui.antialias=true`

  → change the attributes "ui.quality" and "ui.antialias" of the graph. Both attributes are used for a better rendering of the display of the graph.

```
DGS003
st 0
cg ui.stylesheet="graph{fill-color:lightblue;} node{fill-color:red;} edge{size:2px;}"
st 1
an n1 x=10 y=20 ui.label="n1" ui.style="fill-color:  green;"
an n2 x=20 y=10 cost=38 ui.label="n2"
an n3 x=20 y=20 ui.label="n3"
st 3
ae e12 n1 n2
```

| | |
|---|---|
| DGS004 | DGS file format version 4 |
| null 0 0 | for retro-compatibility with older formats |
| st 0 | step 0 (not mandatory for static graphs) |
| cg ui.stylesheet="..." | change graph, attribute "ui.stylesheet" has value "..." |
| st 1 | step 1 |
| an n1 x=10 ... | add Node with identifier n1 and attribute "x" has value 10... |
| ae e12 n1 n2 | add Edge which identifier is e12 linking nodes n1 and n2 |



### 4.2 Reading a Graph from a File

For creating a graph from a file, and if your graph is a static graph, you can read it at once using only one line of code surrounded by statement for catching exceptions.

```
try {
     myGraph.read("path/to/dgsfile.dgs");
} catch(Exception e) { }
```

### 4.3 Writing a Graph to a File

As for reading, writing a graph to a dgs file requires only **one** line of code:

```
graph.write("path/to/file.dgs");
```

Well, actually it also need to be surrounded by a `try { ... } catch(IOException ioe) {...}` statement

### 4.4 Graph-Selfie

When you work with graphs, it is sometimes interesting to save the current layout of the graph as an image, this can be done thanks to the method below.

```
public void screenshot(SingleGraph graph, String pathToImage) {
   if(graph != null) if(graph.getNodeCount() > 0) {
      FileSinkImages fsi = new FileSinkImages(
          FileSinkImages.OutputType.PNG,
          FileSinkImages.Resolutions.SVGA);
      fsi.setLayoutPolicy(
          FileSinkImages.LayoutPolicy.COMPUTED_FULLY_AT_NEW_IMAGE);
      try {
          fsi.writeAll(graph, pathToImage);
      } catch (IOException e) { e.printStackTrace(); }
   }
}
```

When manipulating dynamic graphs it is also possible to record videos from the evolution of the graph. This will be considered in a future lab session.

### 4.5 Exercise 3

Preliminaries:

- Save the following text in file `savedgraph.dgs`:

```
DGS003
"example" 0 0
an "A"
an "B"
an "C"
ae "AB" "A" "B"
ae "BC" "B" "C"
ae "CA" "C" "A"
ce "AB" label="AB"
ce "BC" label="BC"
ce "CA" label="CA"
```

- Create a directory ”dgs” into your project and move the file into that directory.
- Create a directory ”pictures” into your project, all the images will be stored in that directory.

todo:

1. write the lines of code for reading and displaying the graph.
2. Save an image of the graph into the file `"graphbeforemodifications.png"` (for instance)
3. Modify this graph by adding two vertices and two edges and by changing the style of the graph (background color for instance).
4. Save another image (`"graphaftermodifications.png"` for instance) of the new graph
5. Save that graph into the dgs format and store it as `"modifiedgraph.dgs` (for instance) in the ”dgs” directory

## 4.6 Exercise 4

Create a new java class `Tools.java` in the `iwocs` package with three methods which prototypes are:

- `public final static SingleGraph read(String);`

  → this method has one parameter: the dgs filename containing the dgs file description, and it returns an instance of `SingleGraph`.

- `public final static void write(SingleGraph, String);`

  → this method has two parameters, a non null instance of `SingleGraph` and the filename where the dgs description of the graph will be written.

- `public final static void screenshot(SingleGraph, String);`

  → this method has two parameters: a non null instance of `SingleGraph` and the filename of the image of the graph.

and modify the code of `TemplateGS.java` accordingly.