# Graw 0.1 design report
# (Widget for Graph Drawing)

**Julien Jeany**
**Julien Roussel**

julien.jeany@epita.fr
julien.roussel@epita.fr

July 1, 2003

**Abstract**

Graw stands for Widget for Graph Drawing. Originally initiated by Yann Regis-Gianas (yann@lrde.epita.fr), this project is now developped by 2 Epita (Ecole Pour l'Informatique et les Techniques Avancées) second-year students : Julien Jeany (jeany_j@epita.fr) and Julien Roussel (rousse_l@epita.fr).
This document is here to provide the documentation needed to conceive and implement a such project.

Ecole pour l'Informatique et les Techniques Avancées
14-16, rue Voltaire – F-94270 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 44 08 01 01 – Fax. +33 1 44 08 01 99
`info@epita.fr` – `http://www.epita.fr`

# Contents

# Chapter 1

# Introduction

Graw is a widget for graph drawing project. The main problems of this project will be to offer the final user the possibility to use the widget in his project, with a minimum of "genericity", concerning the widget system bind (like Gtk or Qt, for example), the algorithms (visual reorganization of the graph), and the graph representation. Another problem will be to have a good conception of the library, in order to get the possibility to evolve, to improve and to add some features.

# Chapter 2

# Needs

## 2.1 Actors

An actor represents a role played by an external entity (human user, hardware, software, ...) which interacts directly with the system. An actor can consult or modify directly the system's state, emmitting or receiving messages carriers of data. With Graw, we can define 3 main actors : a programmer, a program and the final user.

### 2.1.1 A programmer

In order to use the Graw library, a programmer should implements some specifics needed methods.

### 2.1.2 A program

An external program should use the Graw library giving it some specific messages.

### 2.1.3 A final user

The final user should be allowed to interact with a program, using the library. All the actions with the library should be transparent enough to the user to make it easy at utilisation.

## 2.2 System

## 2.3 Use cases

Some concepts (like node, or edge) will be defined later, but are used here.

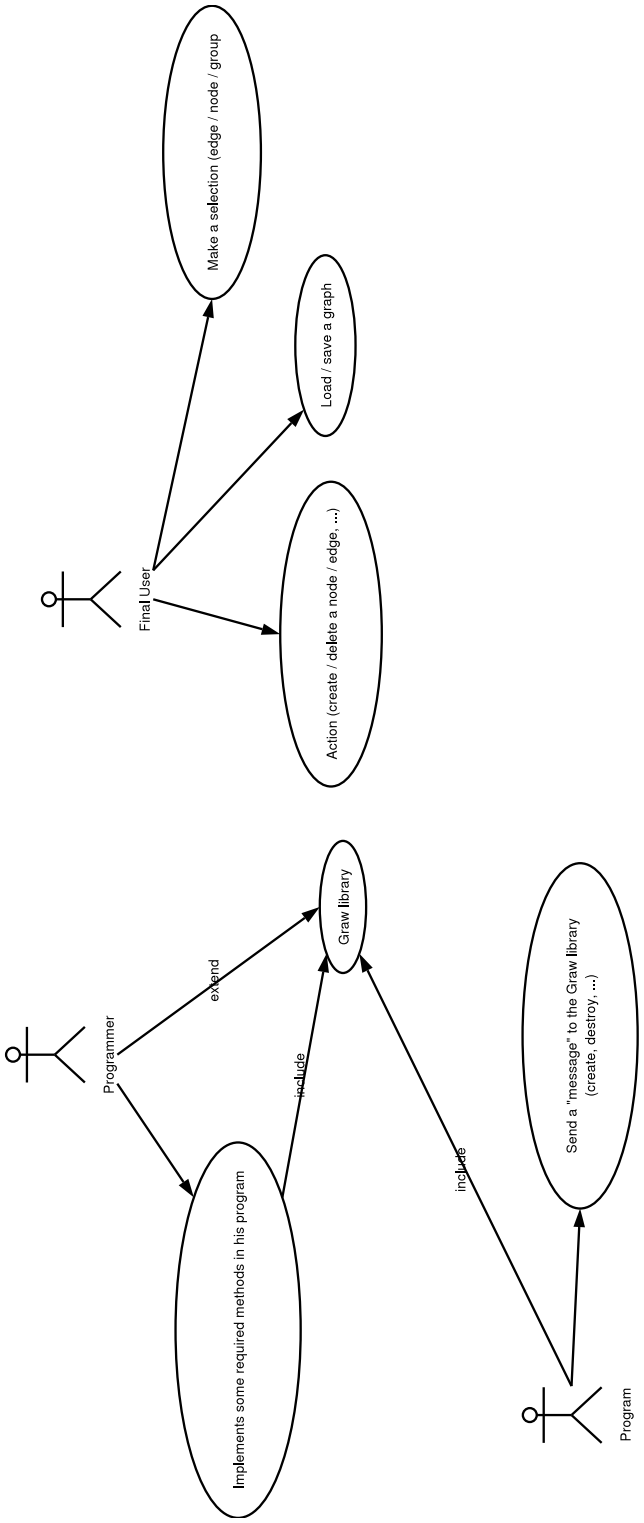
### 2.3.1 Actions

- A simple left-click on a button / menu will action / open it.
- Clicking on a button will make the lib to wait for the user to do the specific action. For example : if the user clicks on the "create node" button, then he should click on the drawing form to perform the action.

### 2.3.2 Selection

- A simple left-click on the draw form will cancel the selection.
- A simple left-click on an object (node, edge) will select it.
- Left-click and drag will draw a box. Each node / edge inside will be selected.
- Ctrl+Left-click will select a non-selected object (node, edge) and add it to the previous selection.
- Ctrl+Left-click will remove an already-selected object from the current selection.
- When a single object is selected, a right-click on it will open a window containing his properties.
- When a multiple (group) selection is engaged, a right-click on it will open a window containing some properties for the group.

## 2.4 Diagrams


Make a selection (edge / node / group
Load / save a graph
Action (create / delete a node / edge, ...)
Final User
Graw library
extend
include
include
Programmer
Implements some required methods in his program
Send a "message" to the Graw library
(create, destroy, ...)
Program

# Chapter 3

# Analysis

## 3.1 Concepts of the system

The system has the concepts to provide a library, which any user could use easily. Only few functions to implement in code to make graw widget available ! Consequently we have a powerfull tool, easy to integrate, and easy to use.

- As powerfull tools is associated the disponibility of many algorithms on graphs and the easy way for graw authors to add new ones.
  As many people use graph, but in so many different ways, no one has the same graph representation, maybe because of many different use. Graw intends to be able to display any graph, independant from the representation. This is possible by asking the final user to provide minimal functions that retrieve the necessary information to draw a graph. As the visualisation system is based on vectorial purpose, this should be easier for everybody to see a graph with good rendering.

- As easy to integrate is associated the possibility to integrate graw (the widget) into applications like other widget of the concerned gui whith only few line (associates to the insertion and the code provided by user to interfer with the graph).
  The User has a special integrated widget for each gui he wants to implement, and each of those widgets is specifically designed for this gui (i.e the Qt widget has been developped the Qt way to be really integrated into the widget suite).

- As easy to use is associated the visual aspect of the widget, especially designed to be easy and simple (mouse clicks, drag and drop, align objects on a grid, ...). Finally, three concepts are mainly exposed: graph genericity, gui genericity and easy to use. Many tricks are given to the end user to manipulate his graph in and out of graw.The main idea is to provide a generic way to let anybody the capability to use graw (even with the most weird graph implementation) and a specialized way to provide a much more optimized access to data (need a preliminar conversion).

## 3.2 Concepts used in use cases

### 3.2.1 Programmer

- Implement some required methods: As any Graw user may have a different data structure to represent graphs, graw must provide a way to draw it without structure restriction. Then the programmer implements few functions adapted to his data structure, those provide necessary information for graw to display a graph. Then we still have genericity on graph structure.

### 3.2.2 Program

- Sends a message to the widget: The graw widget has some exported function like hiding, refresh, ..., depending on the final gui. That permits to include a graw widget in many different gui like Qt, Gtk, Aqua, ....
- Node Representation: The representation for those nodes could be simple shape (circle, rectangle) or complex vectorial picture (computers network representation) with a label.
- Edge Representation: the representation for a typical edge will be a single line with a label.

### 3.2.3 Final user

- Node action: Users can add, delete node, group of nodes. Then can also add datas on nodes and edges.
- Node selection: Users can do selection to handle actions on many nodes or edges.
- Graph loading/saving: Because the work must be saved, graw provide a way to save and load graphs in XML format to be standard compliant.

## 3.3 Static and dynamic point of view

## 3.4 Events stream

# Chapter 4

# Design

## 4.1 Entity, control, interface classes definition

### 4.1.1 Entity classes

**Graph**

This class represents the class structure, already prepared to handle actions as adding/deleting nodes/edges, and is generic enough on structure to permit the final user to use his own. It contains a node list and an edge list. The full graph also have a scale attribute to permit global zoom on the graph.

**Node**

A node is a part of a graph, it contains a name, a label and a position. A basic class is coded. (simple circle with strings in it)

**Edge**

An edge is, like a node, a part of a graph, it contains a name, a label and a position. it links two nodes. A minimal class is implemented to represents the most basic edge type.

**Group**

The usage is one of the most important feature in graw, then a group class has been defined, enabling to group nodes and applying functions on them. A scale parameter can be used to specify zoom on groups.

**Primitive**

To draw a complex graphic, primitive drawing is needed. Drawing a full picture will conclude in composing primitive drawings.

### 4.1.2 Control classes

**Bind**

The bind class controls the communication between the library and the widget because as a library, graw has to be generic to produce many widgets for many GUIs. It also handle zoom and grid capabilities.
This class is an abstract class. the widget coder will derivate a class from bind to code in it all connections with his gui : ability to draw a line whatever the gui (Qt, Gtk, ...).

This class also produce the opposite communication: this class provides also a connection from the widget to the library like GUI events (on mouse click, double click, focus, ...).

#### Register

This class is a singleton to access the canvas anywhere in the application.

#### Loader

This is an abstract class of different loader types. It is used in loading/saving graphs.

#### Visitor

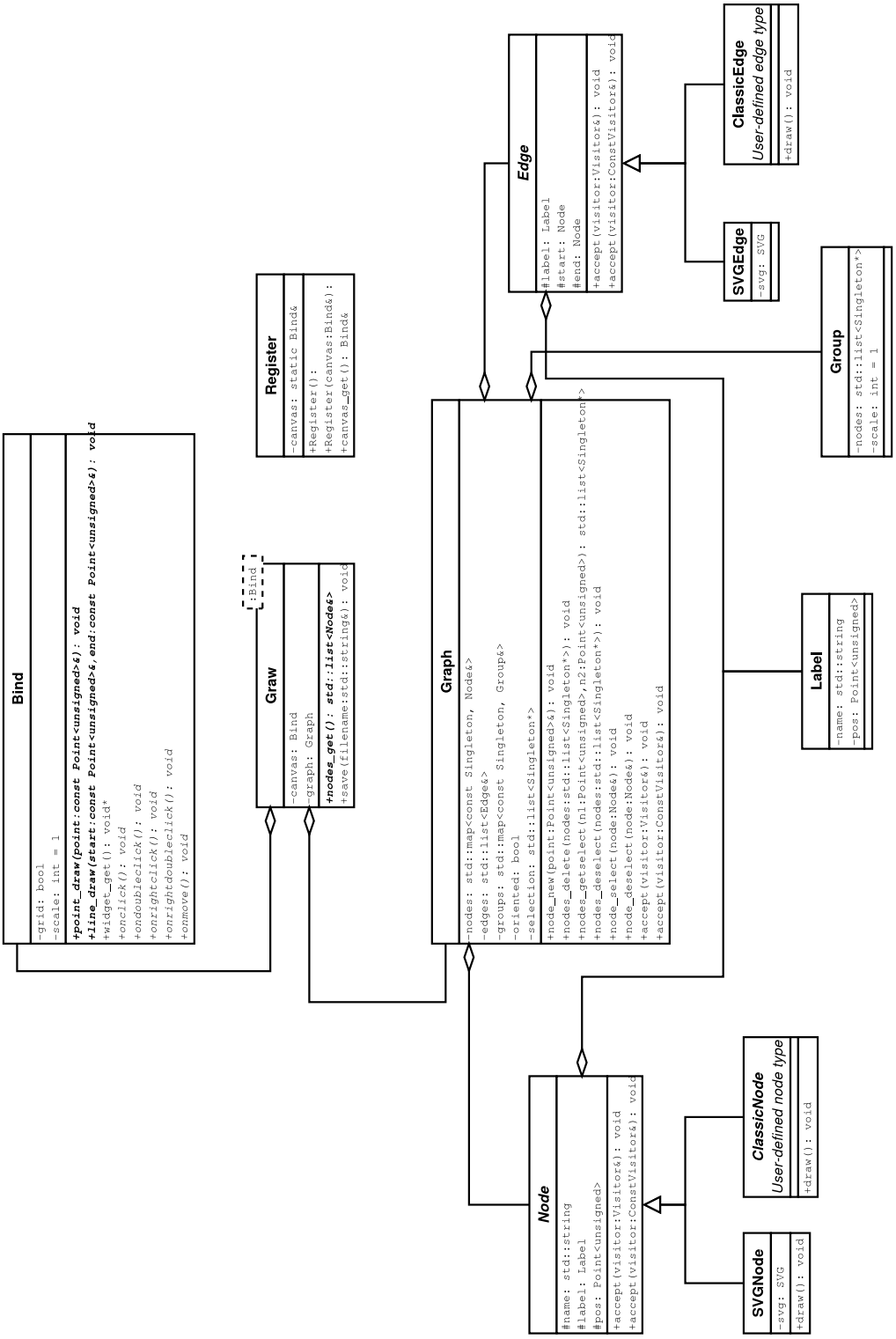
The typical visitor class to visit graph for algorithms.

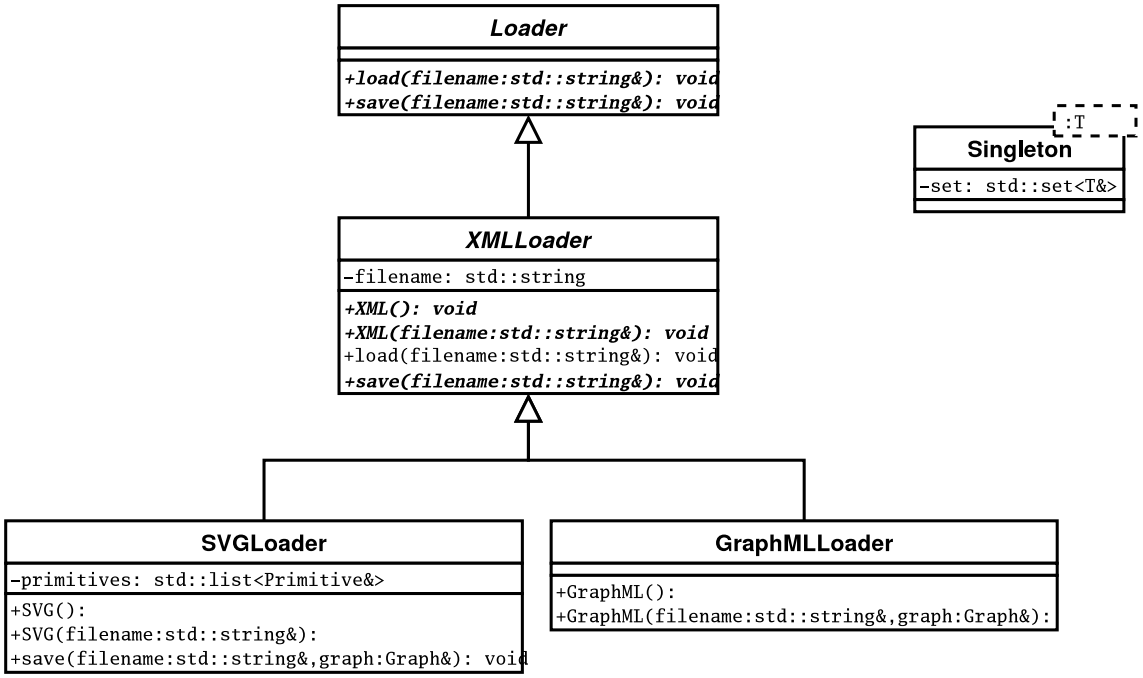
### 4.1.3 Interface classes

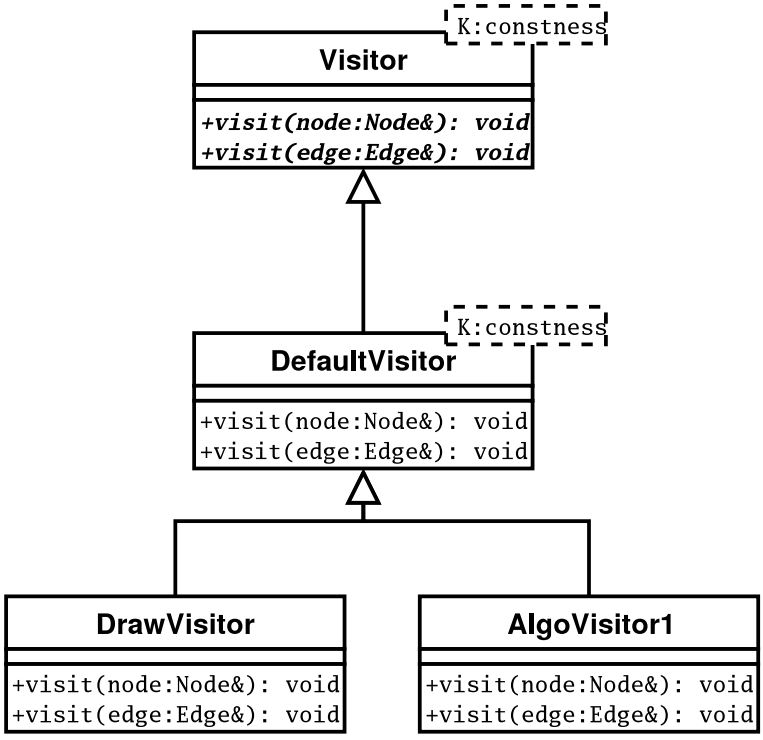
#### Graw

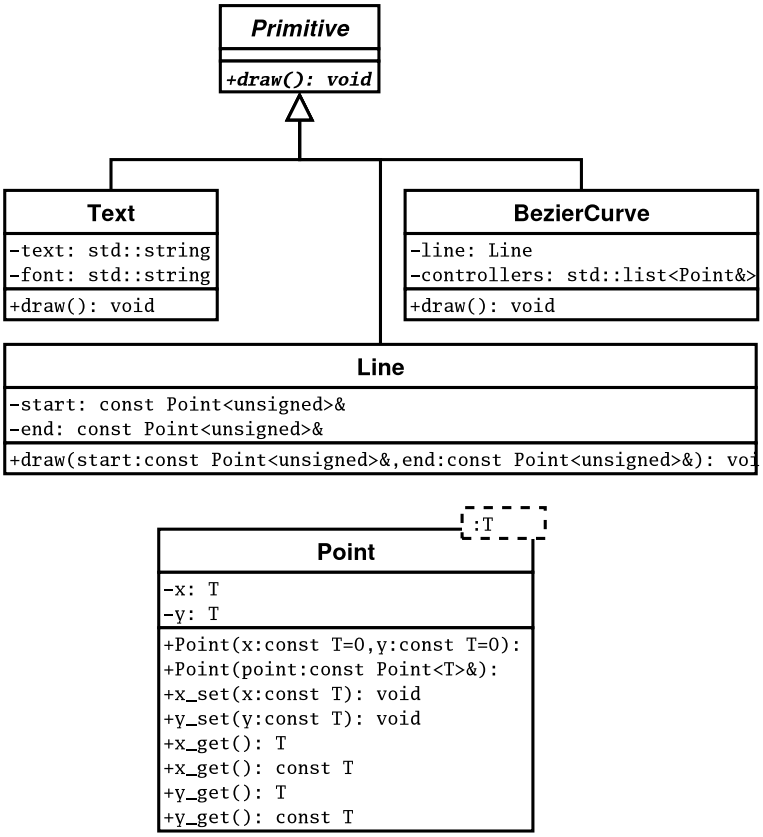
This is the main class, the only one the final user will work with. It produce an easy utilisation with only one class to handle.
To permit the user to communicate his graph to the library, the public get_nodes() has been created. Then the final user will instanciate a class from Graw and redefine get_nodes() to code the interaction between the two different graph structures.

## 4.2 Diagrams


Bind
-grid: bool
-scale: int = 1
+point_draw(point:const Point<unsigned>&): void
+line_draw(start:const Point<unsigned>&,end:const Point<unsigned>&): void
+widget_get(): void*
+onclick(): void
+ondoubleclick(): void
+onrightclick(): void
+onrightdoubleclick(): void
+onmove(): void
Register
-canvas: static Bind&
+Register():
+Register(canvas:Bind&):
+canvas_get(): Bind&
:Bind
Graw
-canvas: Bind
-graph: Graph
+nodes_get(): std::list<Node&>
+save(filename:std::string&): void
Graph
-nodes: std::map<const Singleton, Node&>
-edges: std::list<Edge&>
-groups: std::map<const Singleton, Group&>
-oriented: bool
-selection: std::list<Singleton*>
+node_new(point:Point<unsigned>&): void
+nodes_delete(nodes:std::list<Singleton*>): void
+nodes_getselect(n1:Point<unsigned>,n2:Point<unsigned>): std::list<Singleton*>
+nodes_deselect(nodes:std::list<Singleton*>): void
+node_select(node:Node&): void
+node_deselect(node:Node&): void
+accept(visitor:Visitor&): void
+accept(visitor:ConstVisitor&): void
Edge
#label: Label
#start: Node
#end: Node
+accept(visitor:Visitor&): void
+accept(visitor:ConstVisitor&): void
ClassicEdge
User-defined edge type
+draw(): void
SVGEdge
-svg: SVG
Group
-nodes: std::list<Singleton*>
-scale: int = 1
Label
-name: std::string
-pos: Point<unsigned>
Node
#name: std::string
#label: Label
#pos: Point<unsigned>
+accept(visitor:Visitor&): void
+accept(visitor:ConstVisitor&): void
ClassicNode
User-defined node type
+draw(): void
SVGNode
-svg: SVG
+draw(): void


Loader
+load(filename:std::string&): void
+save(filename:std::string&): void
:T
Singleton
-set: std::set<T&>
XMLLoader
-filename: std::string
+XML(): void
+XML(filename:std::string&): void
+load(filename:std::string&): void
+save(filename:std::string&): void
SVGLoader
-primitives: std::list<Primitive&>
+SVG():
+SVG(filename:std::string&):
+save(filename:std::string&,graph:Graph&): void
GraphMLLoader
+GraphML():
+GraphML(filename:std::string&,graph:Graph&):


K:constness
Visitor
+visit(node:Node&): void
+visit(edge:Edge&): void
K:constness
DefaultVisitor
+visit(node:Node&): void
+visit(edge:Edge&): void
DrawVisitor
+visit(node:Node&): void
+visit(edge:Edge&): void
AlgoVisitor1
+visit(node:Node&): void
+visit(edge:Edge&): void


Primitive
+draw(): void
Text
-text: std::string
-font: std::string
+draw(): void
BezierCurve
-line: Line
-controllers: std::list<Point&>
+draw(): void
Line
-start: const Point<unsigned>&
-end: const Point<unsigned>&
+draw(start:const Point<unsigned>&,end:const Point<unsigned>&): voi
:T
Point
-x: T
-y: T
+Point(x:const T=0,y:const T=0):
+Point(point:const Point<T>&):
+x_set(x:const T): void
+y_set(y:const T): void
+x_get(): T
+x_get(): const T
+y_get(): T
+y_get(): const T

# Chapter 5

# Implementation

## 5.1 Modules

A module system will be used in Graw implementation, allowing the developpers to get a better repartotion of the tasks to do. This should also allow an improvement of the design of the project, and some facilities to extend the library.

The main modules should be :

- Widget : the general widget library, with the differents binds.
- Graph : the differents representations of graphes.
- Algorithms : to manage all the algorithms on graphes.

## 5.2 Technical choices

### 5.2.1 XML

The entire project will be developped in C++. Why this choice ? C++ is a standard in the world of object-oriented languages, and has the advantage to be portable from an architecture to another, and to favorite the modularity and conception of a project. C++ got another strength, which is to make the work of the programmer really easier, with many tools or library already developped (like parsers, mathematical tools, ...).

libxml++ should be the right choice to parse our **XML** files (graphes and pictures). This library is easy to use and fast to incorporate in a project.

Although, **GraphML** will be used to represent the graphes in a XML format. GraphML has the advatages to be rich enough to permit every kind of fantasies on graph representation.

Graw will permit to export graphes as pictures. As this feature exists, Graw should be able to save graphes in many formats. The main format should be the **SVG** one. SVG is an XML file format, developped by the W3C, and used to describe vectorial pictures.

Graw will be binded on some widget-manager, such a **Qt** (for the first version). In a next version, Graw should be usable with a **Gtk** bind.

# Chapter 6

# Tests

The test period is in a graphical application case a difficult operation. After many research over the Internet, we do not have found anything. That's why we will stay in our position : use the "Monkey test".
This remains in a full-clicking way of test. By the way, this kind of tests could be realized by humans, or by a few softwares (not implemented on non-Windows systems). As we develop under the Linux system, we should use the Monkey system.
To test the algorithms, only a human verification could validate them because of their complexity.

# Chapter 7

# Conclusion

The core team expects Graw to be functionnal and utilizable enough to help some other projects. For example, a program using Graw should be able to draw automata, in order to offer a widget for visual-testing for some libraries, such as Vaucanson, or interface for some languages, such as MAL (Minimalist Automaton Language).

# Part I

# Annex

## .1 Provided DTD

```
<!-- ============================================================--->
<!--Parameter entity for data content -->
<!--============================================================--->

<!ENTITY % GRAPHML.data.content "(#PCDATA)">

<!-- ============================================================--->
<!--Parameter entities for attribute list extensions -->
<!--============================================================--->

<!ENTITY % GRAPHML.graphml.attrib "">
<!ENTITY % GRAPHML.locator.attrib "">
<!ENTITY % GRAPHML.graph.attrib "">
<!ENTITY % GRAPHML.node.attrib "">
<!ENTITY % GRAPHML.port.attrib "">
<!ENTITY % GRAPHML.edge.attrib "">
<!ENTITY % GRAPHML.hyperedge.attrib "">
<!ENTITY % GRAPHML.endpoint.attrib "">
<!ENTITY % GRAPHML.key.attrib "">
<!ENTITY % GRAPHML.data.attrib "">
<!ENTITY % GRAPHML.default.attrib "">

<!--============================================================--->
<!--Attributes used by each GRAPHML element-->
<!--============================================================--->

<!ENTITY % GRAPHML.common.attrib "">

<!--============================================================--->
<!--the graphml elements-->
<!--============================================================--->

<!ELEMENT data  %GRAPHML.data.content;>
<!ATTLIST data
                key             IDREF   #REQUIRED
                id              ID      #IMPLIED
                %GRAPHML.data.attrib;
                %GRAPHML.common.attrib;
>

<!ELEMENT default  %GRAPHML.data.content;>
<!ATTLIST default
                %GRAPHML.default.attrib;
                %GRAPHML.common.attrib;
>

<!ELEMENT key (desc?,default?)>
<!ATTLIST key
              id  ID       #REQUIRED
              for (graph|node|edge|hyperedge|port|endpoint|all) "all"
```

```
              %GRAPHML.key.attrib;
              %GRAPHML.common.attrib;
>

<!ELEMENT graphml  (desc?,key*,(data|graph)*)>
<!ATTLIST graphml
                  %GRAPHML.graphml.attrib;
                  %GRAPHML.common.attrib;
>

<!ELEMENT graph    (desc?,(((data|node|edge|hyperedge)*)|locator))>
<!ATTLIST graph
                  id       ID           #IMPLIED
                  edgedefault (directed|undirected) #REQUIRED
                  %GRAPHML.graph.attrib;
                  %GRAPHML.common.attrib;
>

<!ELEMENT node   (desc?,((((data|port)*,graph?))|locator))>
<!ATTLIST node
                 id        ID      #REQUIRED
                 %GRAPHML.node.attrib;
                 %GRAPHML.common.attrib;
>

<!ELEMENT port (desc?,(data|port)*)>
<!ATTLIST port
               name    NMTOKEN  #REQUIRED
               %GRAPHML.port.attrib;
               %GRAPHML.common.attrib;
>

<!ELEMENT edge (desc?,data*,graph?)>
<!ATTLIST edge
               id         ID           #IMPLIED
               source     IDREF        #REQUIRED
               sourceport NMTOKEN      #IMPLIED
               target     IDREF        #REQUIRED
               targetport NMTOKEN      #IMPLIED
               directed   (true|false) #IMPLIED
               %GRAPHML.edge.attrib;
               %GRAPHML.common.attrib;
>

<!ELEMENT hyperedge  (desc?,(data|endpoint)*,graph?)>
<!ATTLIST hyperedge
                   id     ID      #IMPLIED
                   %GRAPHML.hyperedge.attrib;
                   %GRAPHML.common.attrib;
>

<!ELEMENT endpoint (desc?)>
```

```
<!ATTLIST endpoint
                  id    ID       #IMPLIED
                  node  IDREF    #REQUIRED
                  port  NMTOKEN  #IMPLIED
                  type  (in|out|undir) "undir"
                  %GRAPHML.endpoint.attrib;
                  %GRAPHML.common.attrib;
>

<!ELEMENT locator EMPTY>
<!ATTLIST locator
                  xmlns:xlink   CDATA   #FIXED  "http://www.w3.org/TR/2000/PR-xlink-
                  xlink:href     CDATA    #REQUIRED
                  xlink:type     (simple) #FIXED    "simple"
                  %GRAPHML.locator.attrib;
                  %GRAPHML.common.attrib;
>

<!ELEMENT desc (#PCDATA)>
<!ATTLIST desc %GRAPHML.common.attrib;>
```