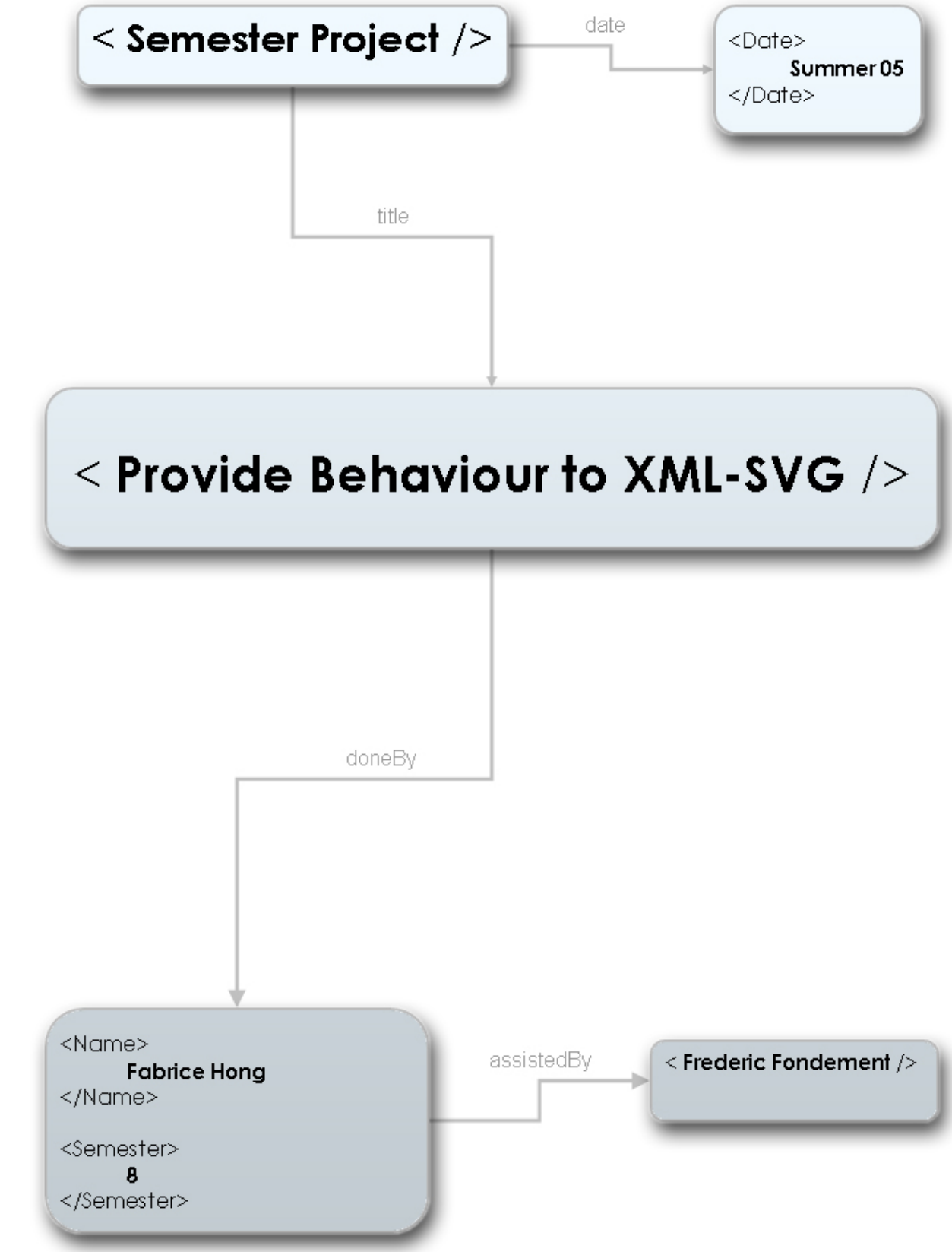

< Semester Project />
date
<Date>
Summer 05
</Date>
title
< Provide Behaviour to XML-SVG />
doneBy
<Name>
Fabrice Hong
</Name>
<Semester>
8
</Semester>
assistedBy
< Frederic Fondement />

Table of Contents

Introduction

The aim of this project is to give facilities to represent graphical concrete syntaxes. For this, the project can be separated in two main aspects. In first hand, SVG which is a XML dialect used to define vector graphics, will be added features for representing concepts and relations between them. On the other hand, extended SVG graphics will be given behaviours thanks to a Java toolkit named DoPIDOM, so the user will be able the interact with the components. The project have been temporary called **ProBXS** which means "**Provide Behaviour to XML/SVG**".
This report is divided in six parts. First of all, the **technologies involved in the project** will be presented and it will be explained why they've been chosen. Then, the problematic will present the **context** in which ProBXS is created and what problem is the latter going to solve. Will follow the description the general architecture used to handle different interactions. After that, a **user guide** will present in details how to use java classes, the constraints that have to be respected and the extentions added to SVG. A lot of graphical examples will also be shown in order to illustrate the aspect of what have been done. Then different aspects about the **current state of the project** such as an analysis and corrections to do will highlight key concepts for those who will continue the project. Finally, the report will be concluded by showing which problems have been encountered and how the project contribute to his context.

1. State of the art

1.1 XML

XML is a W3C standard and means e**X**tensible **M**arkup **L**anguage. It is a very simple and flexible text format and is used as a base to create specialized languages. XML language synthax is based on markups that forms structured data and can be used to represent any kind of concepts.

Objective and utility

The initial objective is to give facilities to exchange structured data on the Web. The main goal is to make a separation between data and its representations. Thus XML store documents data and increase portability between systems thanks to their format which is in UNICODE. There exist many tools on the Web that can load, import, manipulate and save XML.

XML also provide a support to define new languages in an easy way. Nowadays there exist many of these derivated languages that we call XML dialects. They follow formals syntaxes which are defined by DTD (Document Type Definition) or an XML Schema.

Functioning

The character encoding is defined in the first statement of the document, by default UTF-8 is used, which is a particular transcription of UNICODE.
The document is structured in Elements that are defined by means of start and end mark-ups. Elements can nest other elements. The whole set of elements in the document are contained in a unique element called “root”. Apart of elements, XML documents contain different kind of data:

- Comments (that are not part of the data)
- Processing instructions
- “Character calls” (to represent characters that doesn’t exist in the used encoding
- “entity calls” (kind of text macros)

1.2 SVG

SVG is a XML dialect to represent advanced vector graphics. It is used to display vector graphic on the Web and have more less the same capacities of definitions than Macromedia Flash.

Advantages

- open source
- resolution indepandant / bandwidth
- text based
- support transparency
- can be read by many tools
- standard

We will use this standard because he is well known, and that means that a lot of free resources can be found giving the possibility to edit, display and interact with it. One of the main characteristic that differentiate ProBXS from other diagram tools is that the representations are fully customized by the user. So with SVG, the user will be able to design his own component with the tool that he wants. Moreover, whereas vector graphics like flash store its documents in binary, flash store them in human readable/modifiable texts.

1.3 DOM

To parse and manipulate the XML structure of SVG, an advanced tool is needed. Basically, two known toolkit are well known: SAX and DOM. The latter offer an easy handlable tree interface that represents XML structure. For this project, the use of a tree structure to represent the SVG components and their hierarchy is clearly unavoidable. For this reason and because of the numbered advantage that DOM offer over SAX, the project will be partly based on DOM.

Using DOM

At first the application has to load SVG document using the parser. This latter will build a tree of elements called nodes. There exist many different kind of nodes, for example :

- Text nodes
- Attribute nodes
- Element nodes
- Comments nodes

The nodes are given methods by means of interfaces in order to be accessed, or to navigate through the tree. This allows applications to handle XML quite easily.

Advantages of DOM

- Robust and complete API for manipulating the tree.
- Relatively simple to modify the data structure and extract data.
- Isolate DOM nodes we want to deal with are isolable and can be used a vector of information between classes.
- Each component of the tree is represented by classes. This allows to subclass and therefore to wraps DOM nodes to run other mechanisms on the tree structure.

1.4 DoPIDOM

DoPIDOM is a java implementation of the DPI model based on DOM and SVG. The aim of the DPI model (Documents, Presentations, Instruments) is to provide an alternative to current application-centred environments by introducing a model based on documents and interaction instruments.
DPI makes it possible to edit a document through multiple simultaneous presentations. The same instrument can edit different types of content, facilitating interaction and reducing the user's cognitive load. DPI includes a functional model, aimed at the user interface designer, that describes implementation principles in terms of properties, services and representations. The DPI model offers a first but essential stage in designing and implementing a new

generation of document-centered environments based on a new interaction paradigm.
DoPIDOM propose an architecture in which Components and behaviours are defined separately.

DoPIDOM and SVG

SVG is a good base to work on, but the problem is that it's far not sufficient just to describe and to work with graphical concrete synthax that represent instances. Interactive and dynamics components are needed, but SVG is static !
It is possible to define some simple interactions with the SVG standard, but it will not provide the complex one requiered by the graphical edition of a language.
DoPIDOM wrapps XML/SVG elements represented in the DOM tree, in a new kind of component wich can consume or produce actions or queries that will modify or get informations.

1.5 Batik

Batik is a java toolkit for applications or applet that want to use images SVG format for various purpose, such viewing, generation or manipulation.
In our case we will need the parsing functionalities, and the renderer module which use a swing derivated canvas that can display SVG components.
There is some other functionalities like a SVG generator that translate the usual graphic interface of swing in SVG language, or even a browser, that is like a little application to load SVG documents.
When actions modify the DOM tree, Batik update the display immediately.

2. Problematic

2.1 Context : Language development

Nowadays the best approach for development is to chose a specific language for each requierement. But currently, developers are seriously constrained by the limitations of programming languages. With the Language-Driven development, the key thing is that developers can design their own languages which tend to focus on implementation level abstractions.

2.2 Aim of the project

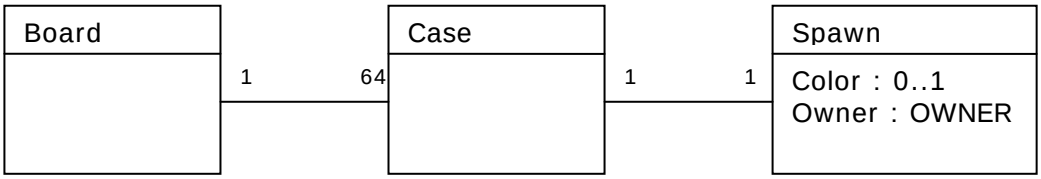
When we create a language, we have to define abstract and concrete synthaxe. Abstract syntax is the metamodel of the language and concrete syntax is the way to modify the instances. This concrete syntax usually is the vocabulary of the language, but here we are interested in a graphical way of programming. The aim of the project is to provide facilities to create such languages. ProBXS only help on the graphical part of this language modeling, more precisely, on the behaviours and components that could be needed to represent graphical instances. The representations would be based on SVG to let the user design his own language. To give language modeling abilities to SVG, this latter is extended so the user must be able to have some interactions with his components such as :

- Translation
- Attachement
- Ability to show relations between them
- and more

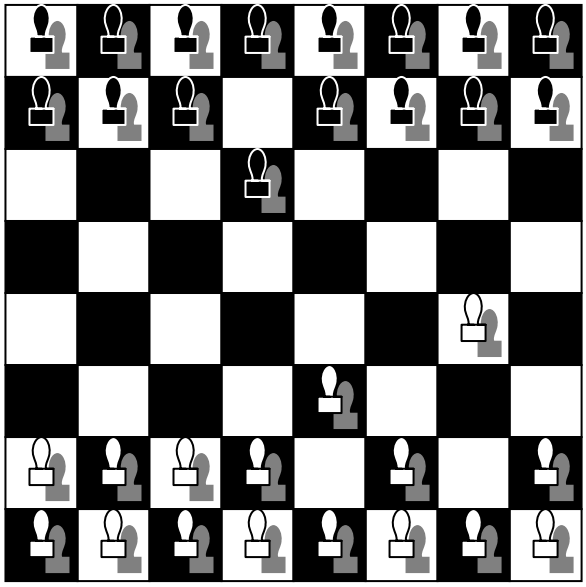
2.3 Examples of abstract and concrete syntax

The chessboard

Abstract syntax (metamodel) :

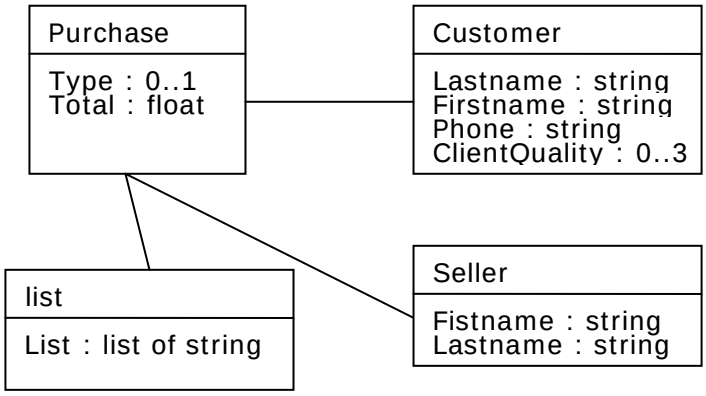


Concrete syntax (instance) :

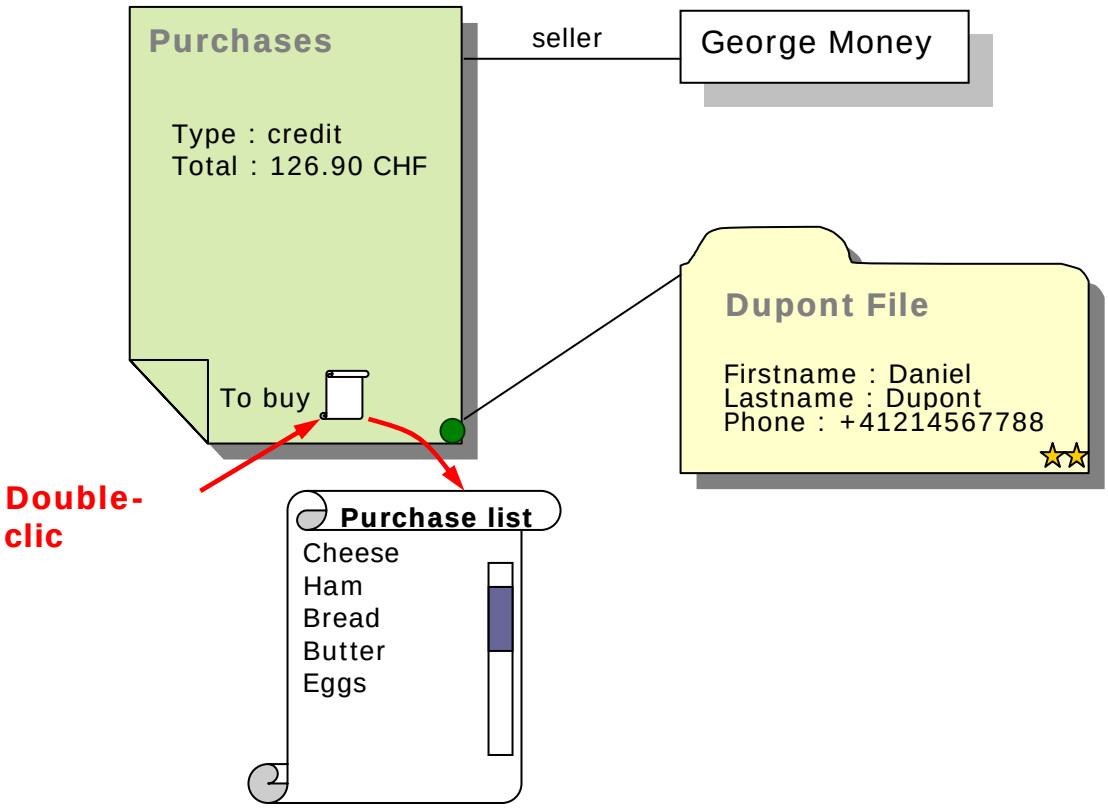


The purchase list

Abstract syntax :

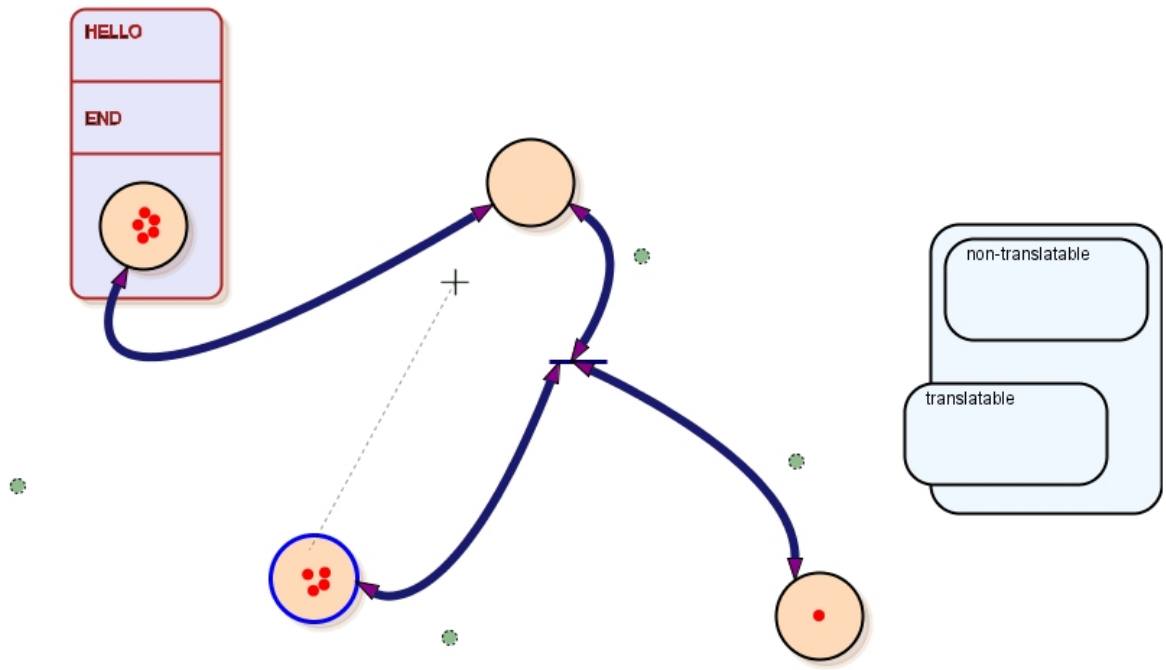


Concrete syntax :



Petri networks

Concrete syntax (screenshot) :

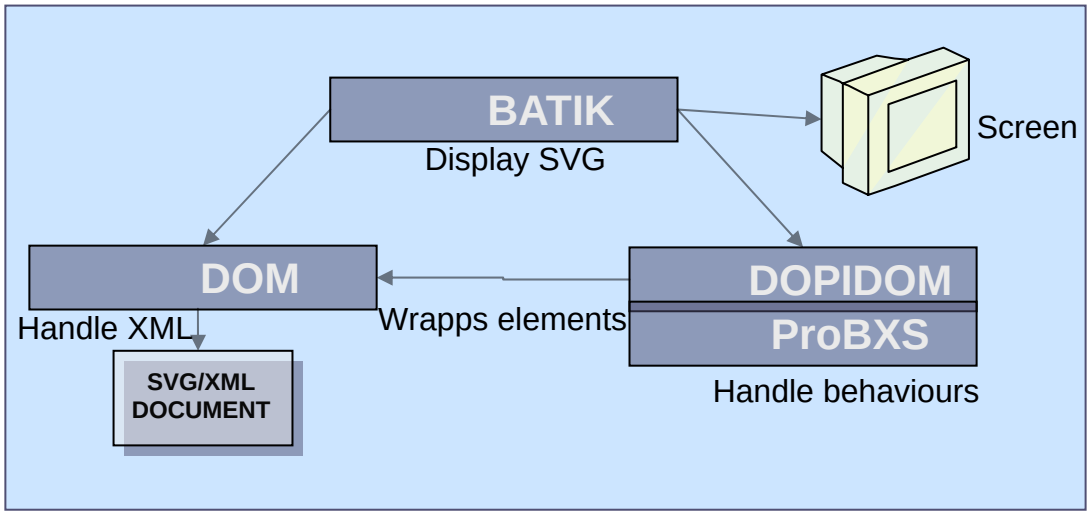


3. Architecture

3.1 General architecture

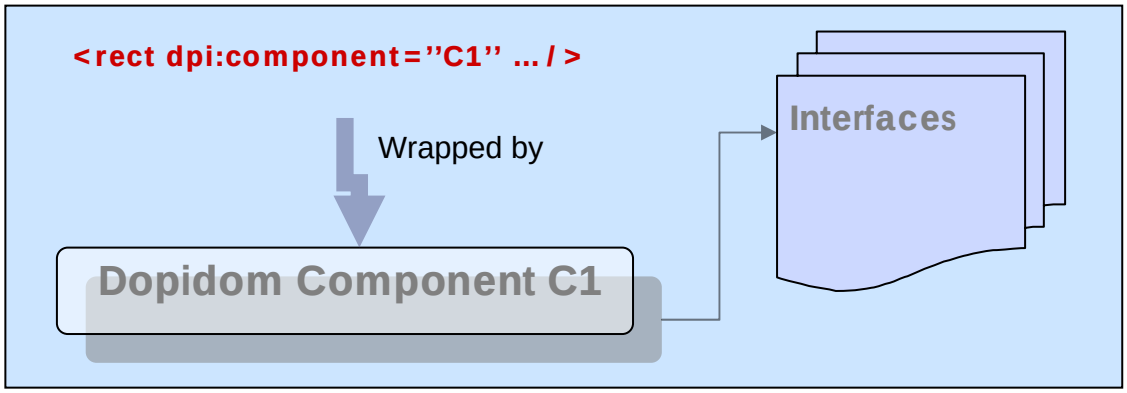
To program the behaviour of the component we need an object oriented programming language. For this we chosed to develop the project in a Java 1.5 environement.
ProBXS is not a toolkit itself but complete the DoPIDOM proposed architecture, that is to say, it adds features and new components. Three main modules run together to make the interaction with SVG components possible :

- Batik to display SVG graphics
- Dom to handle XML/SVG
- DoPIDOM to handle interactions with SVG



3.2 DoPIDOM architecture

Here will be presented the basic functionning of DoPIDOM and additional features and concepts that have been added.
The way to create behaviour for SVG element is quite simple with DoPIDOM. First of all, a subclass C1 of *Component* have to be created. In the constructor of this class, interfaces that the component must compose are declarated.
Finally the SVG element must be assigned the wanted class (possible package must apprear) with the attribute : *dpi:component*.



Components

In the modeling language context, every concepts is represented by SVG elements that can be wrapped by *Component*s to be given behaviours. We can point out two kind of behaviours for the components :

- The sharable behaviours : the Actions and Queries (consumed by interfaces)
- The behaviours inherent to the function of the component that are the methods of the component class

Components data :

The state of a DoPIDOM component must be able to be completely defined by SVG. Once a component is loaded, the interactions with the components change the datas, moreover additional datas have to be taken in account.

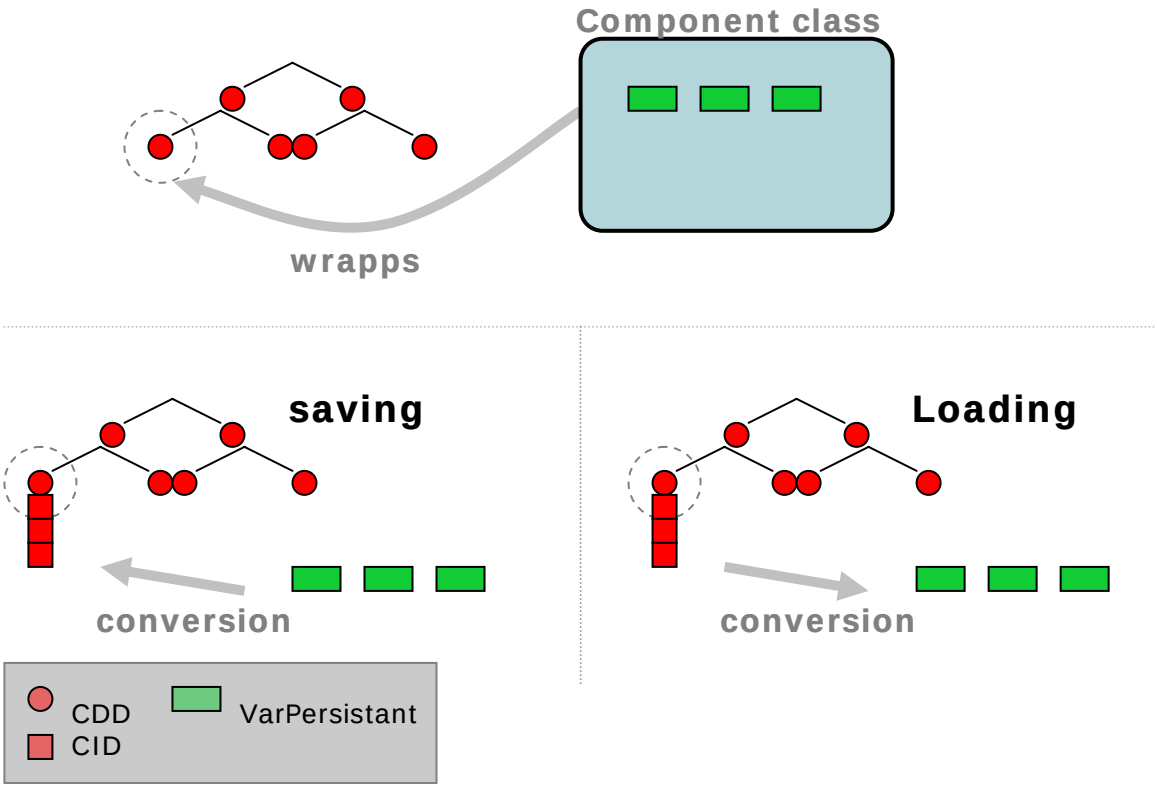
Afterwards, the entire informations about the component have to be stored in SVG. Two kind of data can be distinguished :

- The Component Definition Data
  Mandatory data about the properties of the object or about the initial state. For instance, the color, the width, the other object that it will use as sub parts (ex: Link Arrow)
- The Class Instance Data
  Optional data that represent the state of some class variable. For instance a number, a list of attached component, the component on which an arrow is sliding.

In most cases Class Instance Data overlapp Component Definition Data. If it's the case, Class Instance Data always have priority.

The Class Instance Data are handled by *VarPersistant*s that will be explained in detail in the section.

Here is shown the role of VarPersistant during saving and Loading :



Classical structure of components :

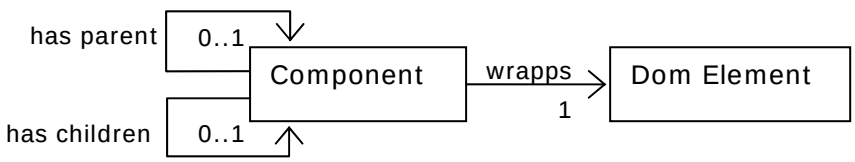
First of all, varPersistants are declared and will load possible Class Instance Data.

Then in the constructor, three important parts should be present :

- Declarations of interfaces, that is to say, the specification of which actions and queries component will be able to consume (such as Translatable, ColorGettable, Locatable)
- getParameters() method call. Used to get Component Definition Data
- checkInterfaces method call. Used to check if all mandatory interfaces are present

Class Diagram :

Because Components wrapps Dom SVG Elements, they keep the structure of this latter and thus components can access their hierarchical parent or children.



Interfaces

Interfaces, actions and queries :
Interface consume actions or queries. Interface class define the behaviour whereas actions or queries class are just used as container to send data or get data from interfaces. The execution of an interface is done by calling the method *doAction()* or *doQuery()* with the action or query as parameter.
More than one interface can consume the same action/query, but an Interface can consume only one of those.
Until now, only one interface has been assigned by action

Here are the list of the Interfaces associated with their consumable actions or queries:

ActionInterfaces :

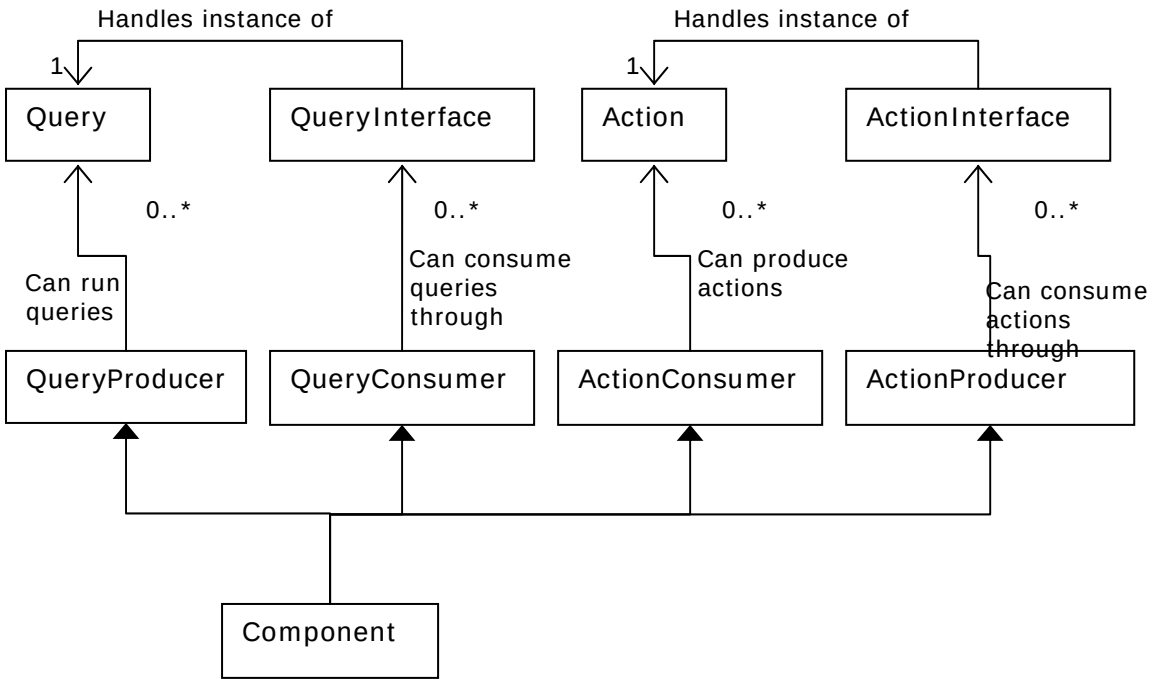
Interface	Associated Action
BorderSlidable	BorderSlide
DirectionAdjustable	AdjustDirection
LinePullable	PullLine
Positionable	Position
Stickable	Stick
Translatable	Translate

QueryInterfaces :

Interface	Associated Query
BorderFindable	GetBorder
OriginGettable	GetOrigin

Utility and use of each interfaces will be explained in the user guide

Classes Diagram



Interactions

In the continuation of ProBXS project, toolbars will be present somewhere in the scene to allow users to chose which kind of cursor they would like to use. To each type of cursor would be associated a certain role, in other words, the ability to trigger specifics actions or queries on consumer components.

SVGInteractions :
A SVGInteraction is a class that associate production of queries of actions to mouse triggers such as *mousePressed(), mouseReleased(),* etc. The cursors are related to the SVGInteractions the same way as components are related to interfaces. In the constructor of *SVGCleanToolInteractor*s (cursor), possible SVG interactions are defined.

Until now, the work has been concentrated on creating different kind of component more than on diversifying possible interactions; the only cursor available is named *Pointer*. All possible actions are triggered by it. For this reason, the interactions are currently used only to separate in logical groups actions and queries triggerable by the *Pointer* :

- Attacher (make relations between objects)
- CaretManager (text edition)
- Loader (Scene Loader)
- Saver (Scene saver)
- Selector (select objects)

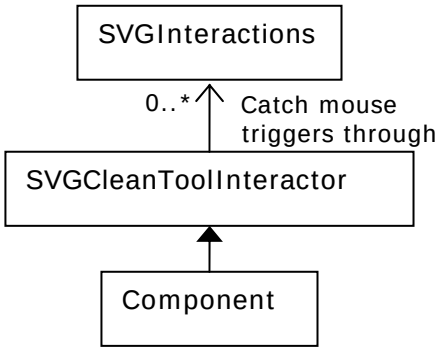
- Slider (slide arrows on borders)
- Translater (Translate objects)

Ideal set of cursors :

An ideal set of SVGInteractions would correspond to a set of cursor such as :

- The selecter
  - Select multiple objects by mean of temporary selection rectangle
  - Select multiple objects one by one
  - Contextual menu for selection
- The Translater
  - Move selected components
  - Move free arrows
  - Slide arrows on border of components
  - Move line points or handles
- The attacher
  - Stick components together
  - Unstick components
- The Creator
  - Create new Components from models
  - Delete unwanted component
- The Editor
  - Edit text
  - Edit lists
  - Edit GraphicContainers
- Resizer
  - Scale components
  - Rotate component

Classes Diagram :



Cursors are *SVGCleanToolInteractors*

4. User Guide

Color code :
Diagrams :

- Grey : abstract methods
- Purple : important variables (like varPersistants)
- Blue : statics methods
- Back : public methods

SVG definitions :

- Black : standard SVG
- Red : PROBXS attribute (extention to SVG)
- Orange : optional attribute

4.1 Interfaces

BorderSlidable

Type : Action

Utility :
Allow a component be translated along the border of a BorderFindable Component. Normally used for arrows.

Input (action content) :

- List of point representing the border *b*
- Coordinates of a point *pt*

Behaviour :
Put the wrapped element on the border *b* at nearest point from *pt*.

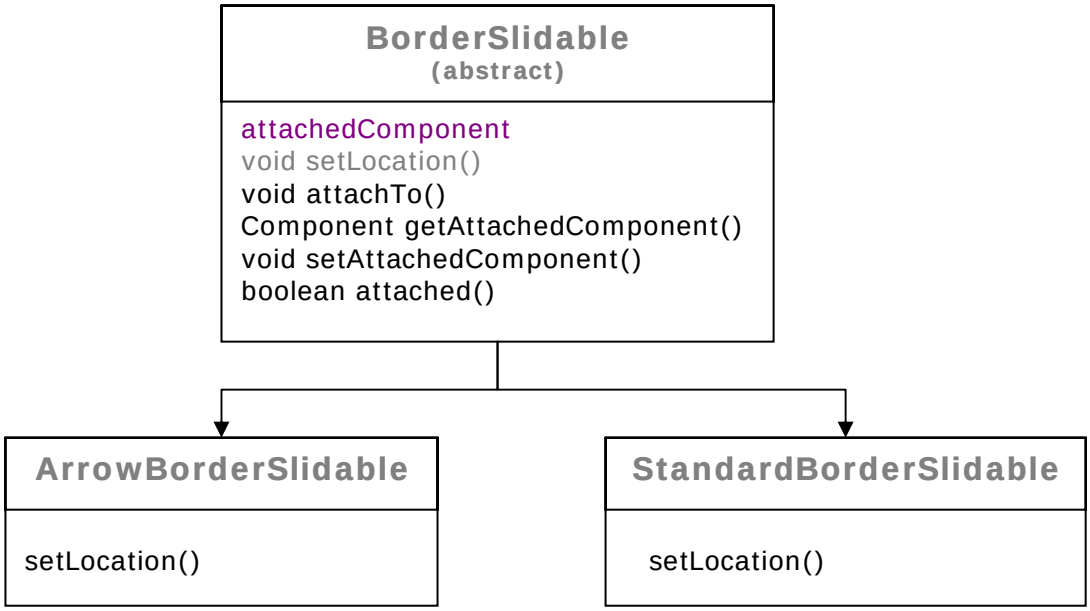
VarPersistants :
attachedComponent : BorderFindable Component on which the wrapped element is sliding.
Parameter=”BorderSlidable(attachedComponent, *componentID*)”

Dependent interfaces :
BorderFindable (Query Interface), GetBorder (Query)

Subclasses ArrowBorderSlidable :
Dependancies : Arrow
setLocation() : Place the arrow of a constructed Link on the border and adjust its orientation.

Subclasses StandardBorderSlidable :
setLocation() : Place the top left side of an object on the border

Diagram :



DirectionAdjustable

Type : Action

Utility :
Used to orient a component in the direction of another object (a point). Generally used to synchronize the *Arrow*'s orientation with the curve of their *Link*'s

Input (action content) :

- A *Point2D* that represent the relative vector for the new orientation.

Behaviour :
Put the wrapped element on the border *b* at nearest point from *pt*.

VarPersistants :
none

Dependent interfaces :
GetOrigin (Query)

Diagram :
Not relevant

LinePullable

Type : Action

Utility :
Generally used to draw a temporary line between two points, for example to make a relation

Input (action content) :
- Source point of the line
- Destination point of the line

Behaviour :
Important remark : doAction call must be done between *beginInteraction()* and *endInteraction()*. *beginInteraction()* will create a *SVGLineElement*, and *endInteraction()* will destroy it.
doAction() modify the coordinate of the *pulledLine* according to the given points.

VarPersistants :
none

Dependent interfaces :
none

Diagram :
Not relevant

Positionable

Type : Action

Utility :
Used to set the position of a component

Input (action content) :
- New coordinates of the component

Behaviour :
Set the component's position using the transform attribute

VarPersistants :
none

Dependent interfaces :
none

Diagram :

Not relevant

Stickable

Type : Action

Utility :
Allow to group the translations of several objects. The stickable component can have several children that will follow his movement.
Remark : the children displacement doesn't affect the parent.

Input (action content) :

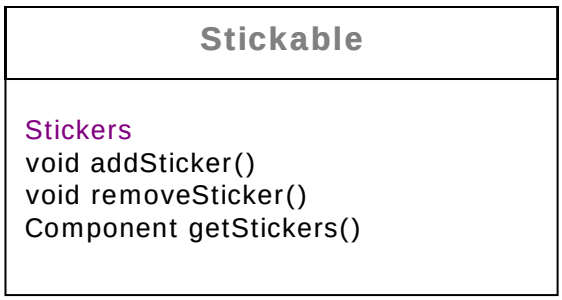
- Relative value of displacement

Behaviour :
Change the component's position using the transform attribute and forward the *translate* action to its children

VarPersistants :
Stickers : List of children elements to which the translation will be forwarded
parameter="Stickable(stickers, {*componentsID*, })"

Dependent interfaces :
Translatable (Interfaces) : Objects involved in Stickable must be translatable

Diagram :



Translatable

Type : Action

Utility :
Used to move a component to a given relative value

Input (action content) :

- Relative value of displacement

Behaviour :
Change the component's position using the transform attribute

VarPersistants :
none

Dependent interfaces :
none

Diagram :
Not relevant

BorderFindable

Type : Query

Utility :
Extract the contour from a SVG basic shape and return a list of points generally used by *BorderSlidable*

Output (Query content) :
ArrayList of *Point2D* that represent the border.

Behaviour :
Compute the points that represent the contour from those type of shapes :

- Polygon
- Rectangle
- Circle
- Line

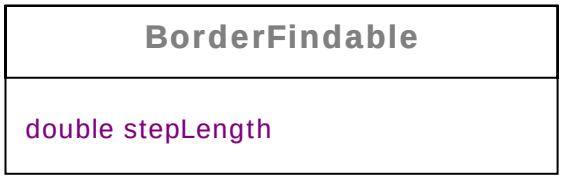
And return an *ArrayList* of *Point2D*

For the contour extraction of circles, the number of step is calculated in relation with the *stepLength* which is a constant variable of BorderFindable class.

VarPersistants :
none

Dependent interfaces :
none

Diagram :



OriginGettable

Type : Query

Utility :
Get the exact position of the wrapped element tenant compte the possible multiple transform attribut up in hierarchy. Moreover the query the position related to the component origin defined in SVG, not the top left bounding box of the object.

Output (Query content) :
Return a *Point2D* that represent absolute position of the wrapped Element's origin

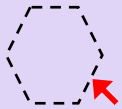
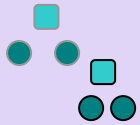
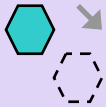
Query :
Get the absolute position of the wrapped Element's origin

VarPersistants :
none

Dependent interfaces :
none

Diagram :
Not relevant

4.2 Interfaces visual sumary

Interface			Interface	
BorderSlidable			Stickable	
DirectionAdjustable			Translatable	
LinePullable			BorderFindable	
Positionable			OriginGettable	

4.3 Components

AnchorPoint

Utility :
Component used to attach *Links*.

Use :
There is two general use of *AnchorPoint*. You can use it as a visible shape that can anchor the links, or you can also set it invisible (SVG attribute : visibility="hidden") in order to make only a part of another shape link attachable.

Characteristics :
Being BorderFindable, you can only define *Rectangles*, *Circles*, *Lines*, and *Polygons* as AnchorPoints.

VarPersistants :
Links : list of the links attached
parameter="AnchorPoint(links, {*componentsID*, })"

Mandatory Interfaces :
Stickable
BorderFindable

Example :

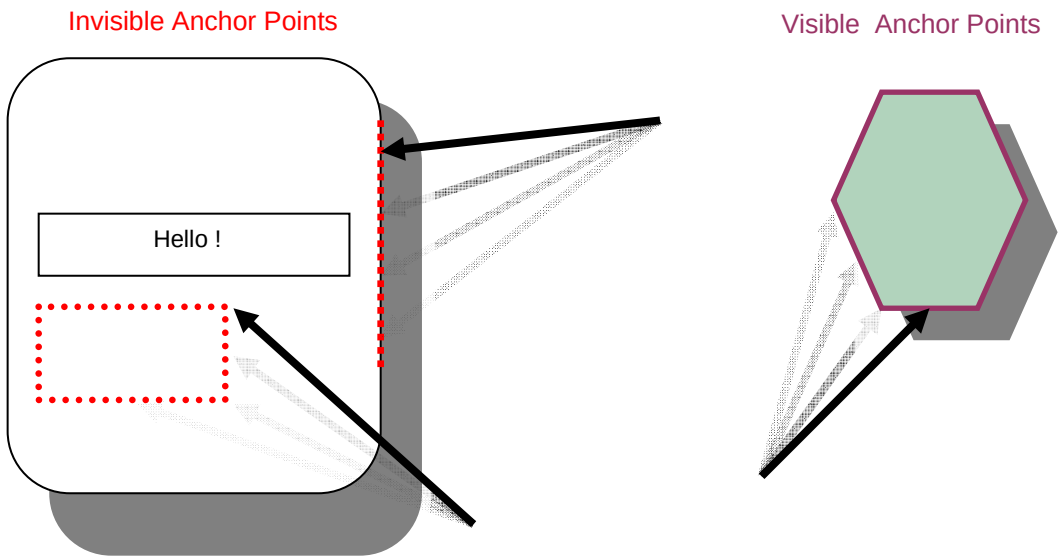
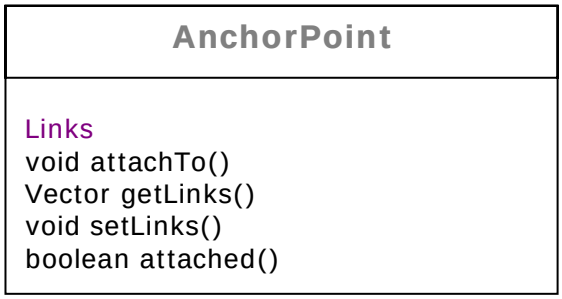


Diagram :



SVG definition :

Example for a circle *AnchorPoint* :
<circle dpi:component="AnchorPoint" linkType="#link1" r="30" .../>

Arrow

Utility :
Component used to handle and represent behaviour of Link's *arrowStart* and *arrowEnd* according to the body link's anchor points movement.

Use :
To make an arrow work properly, you must :

- Set its attached *Link*
- Set it as a *startArrow* or *endArrow*
- Use *adjustArrow* when the arrow have to be reoriented according to the *Link*'line or *AnchorPoint* movement

This procedure will be automatically done by the Link component during its construction

Characteristics :
For now, the arrow can only be polygons (to be changed !)

VarPersistants :
position : Integer specifying the position of the arrow on the Link (*startArrow* or *endArrow*)
parameter="Arrow(position, *positionInteger*)"

attachedLink : Component specifying the *Link* that the arrow belongs to
parameter="Arrow(attachedLink, *componentID*)"

Mandatory Interfaces :
ArrowBorderSlidable
Translatable
DirectionAdjustable

OriginGettable

Example :

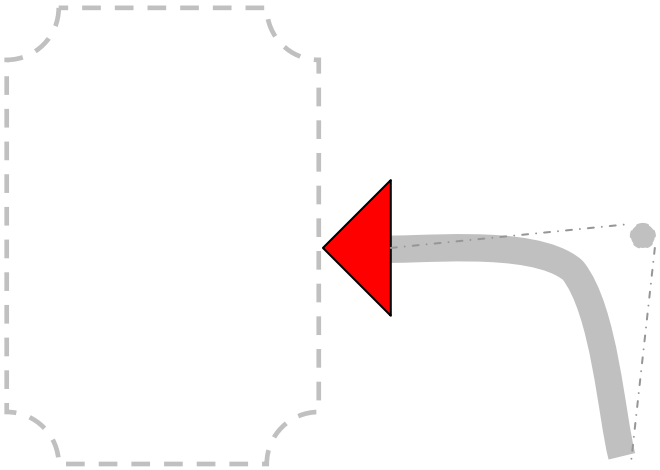
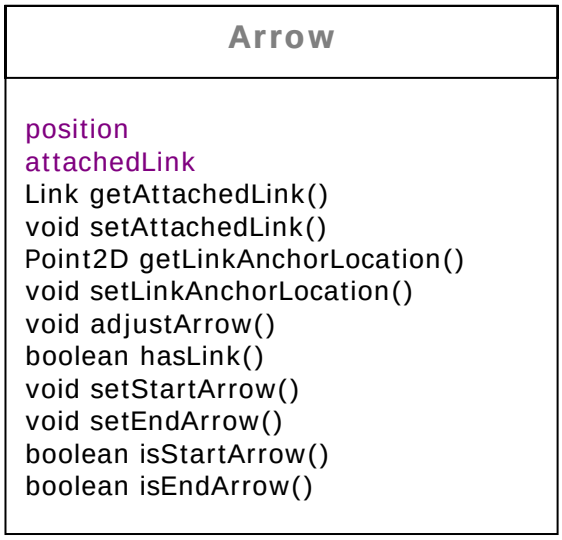


Diagram :



SVG definition :
<polygon id =”arrow” dpi:component=”Arrow” ax=”15” ay=”0” points=”...” .../>

ContainedGraphic

Utility :
Graphic that *GraphicContainer* can contains (see GraphicContainer)

Use :
Not relevant

Characteristics :

Generally automatically generated by the GraphicContainers. Shouldn't appear

VarPersistants :
none

Mandatory Interfaces :
Translatable
Positionable
OriginGettable
Locatable

Example :

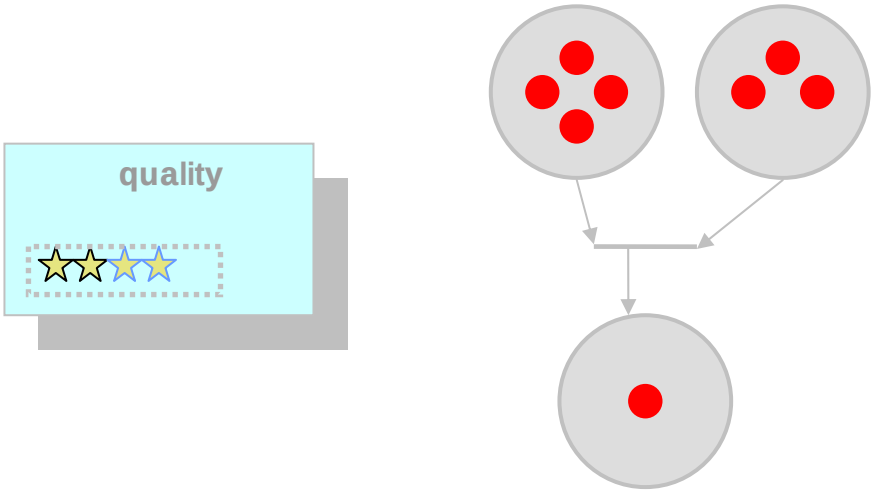


Diagram :
Not relevant

SVG definition :
Example of a circle ContainedGraphic:
<circle dpi:component="ContainedGraphic" id="cg1" cx="0" cy="0" r="4" fill="black"/>

CurvedLine

Utility :
Body of the *Link*. Manage the points and handle structures and synchronize them with the SVG path element.

Use :
It is possible to create the *CurvedLine* on the fly (for example during an on the fly Link creation) instead to define it in the SVG model file. For that using the LineDescription class can be used (see the related section for more details). If it's done so, after the *CurvedLine*

creation, the parentLink must be set, then points can be added (handles will automatically be added between the new points)

Characteristics :
Can be automatically generated by a *Link* component's creation.
The CurvedLine only wrapps SVG *path* elements. Moreover, you can only define curved lines.
The format in the 'd' attribute to define a curved line in svg is :
Mx, y {S{hx, hy, px, py}+}, where x and y are the first position of the line, px and py are the rest of the points, and hx and hy their corresponding handles
To work properly, the parentLink must be set before use, but this procedure will be automatically done by the Link component during its construction

VarPersistants :
parentLink : Component specifying the *Link* that the *CurvedLine* belongs to
parameter="CurvedLine(parentLink, *componentID*)"

Mandatory Interfaces :
none

Example :

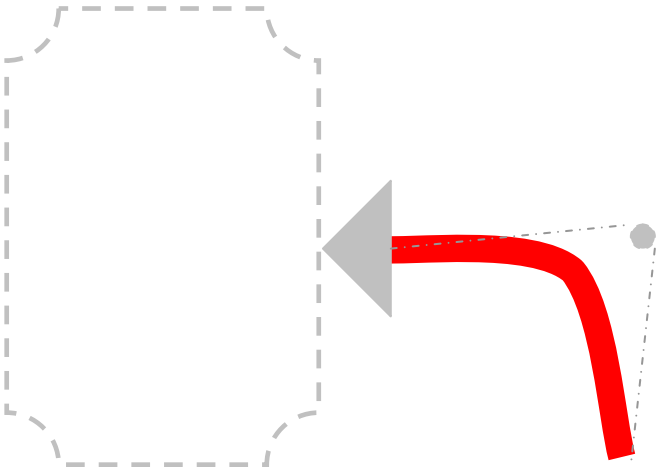
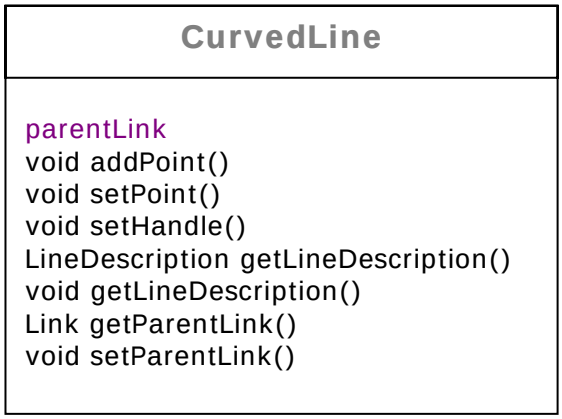


Diagram :



SVG definition :
Exemple of a circle *GraphicContainer* :
<path dpi:component="CurvedLine" d="M10, 10 S50, 50 70, 100" stroke-width="2" stroke="black"/>

GraphicContainer

Utility :
Used to represent numeric informations into graphical ones. Several methods of positionement should be available

Use :
At initialization de number of contained graphic is set to 0. This number can be changed with the provided methods, then the *draw()* method must be called so the positions of the elements can be calculated and drawn

Characteristics :
The positionement method (in-line or point centered) is defined by the svg element type. For now only *circle* and *rectangle* are available

VarPersistants :
components : list of the contained components
parameter="GraphicContainer(components, {*componentID*, })"

shape : Integer specifying the type of shape, therefore the type of positionement
parameter="GraphicContainer(shape, *shapeInteger*)"

Mandatory Interfaces :
none

Example :

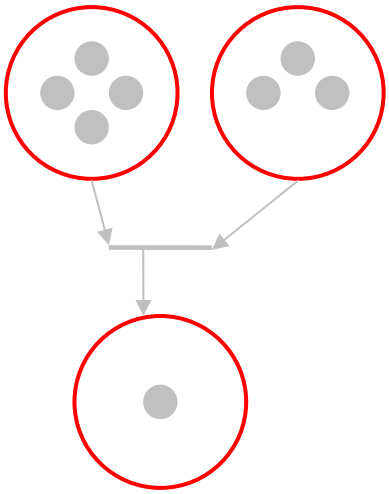
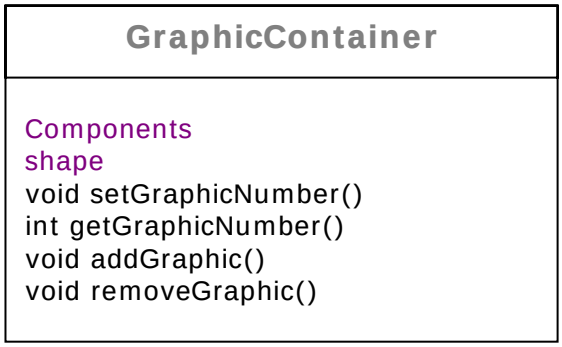


Diagram :



SVG definition :
Exemple of a circle *GraphicContainer* :
<circle dpi:component="GraphicContainer" cx="0" cy="0" r="30" stroke-width="2" stroke="black" fill="white" graphic="#cg1"/>

LineHandle

Utility :
Used to deform *CurvedLines*. They are synchronized with the handles of a SVG curved line path

Use :
LineHandle are generated automatically when points are added on a Link. The visual aspect of handles are defined when they are created with the class *HandleDescritption*

Characteristics :
As they are automatically generated, *LineHandles* shouldn't appear in a SVG model (except in saved ones).
Handles are identified on a CurvedLine by their *hIndex*

VarPersistants :
attachedLine : Component specifying the *Link* that the arrow belongs to
parameter=”LineHandle(attachedLine, *componentID*)”

hIndex : Integer specifying the *LineHandle*’s position on a *CurvedLine*
parameter=”LineHandle(hIndex, *positionInteger*)”

Mandatory Interfaces :
Translatable
Handlable
Locatable

Example :

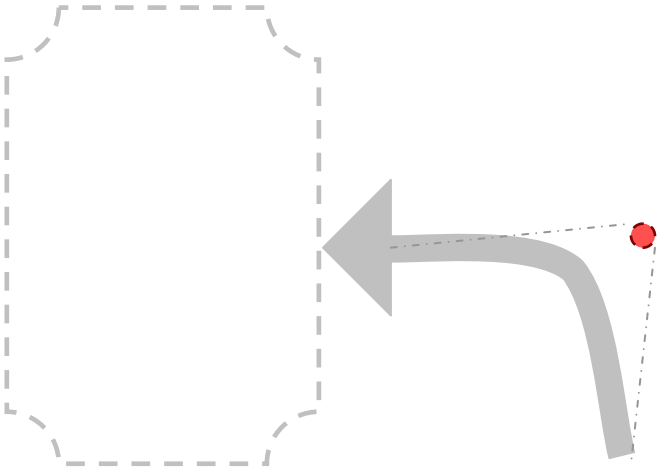
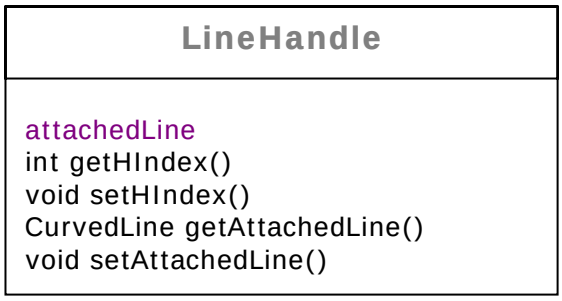


Diagram :



Link

Utility :

Represent the groups of informations. That group of informations allow to instanciate new links from a model and define the visual characteristics

Use :
There is a possibility to define a Link without path, in this case an empty *CurvedLine* will be automatically created with the attributes specified in the SVG node of the Link (color, witdh, etc). If the path attribute exists, those attributes are taken from this latter. When a Link component is created, it automatically calls methods in their *Arrow* and *CurvedLine* children so they can work properly. After the *Link* instanciation, the start then the end location of the *Link* must be set. Those function automatically add points if necessary.

Characteristics :
Must be defined if a SVG group.

VarPersistants :
curvedLine : Component specifying the *CurvedLine* children
parameter="Link(curvedLine, *componentID*)"

startArrow : Component specifying the start *Arrow* children
parameter="Link(startArrow, *componentID*)"

endArrow : Component specifying the start *Arrow* children
parameter="Link(endArrow, *componentID*)"

Mandatory Interfaces :
none

Example :

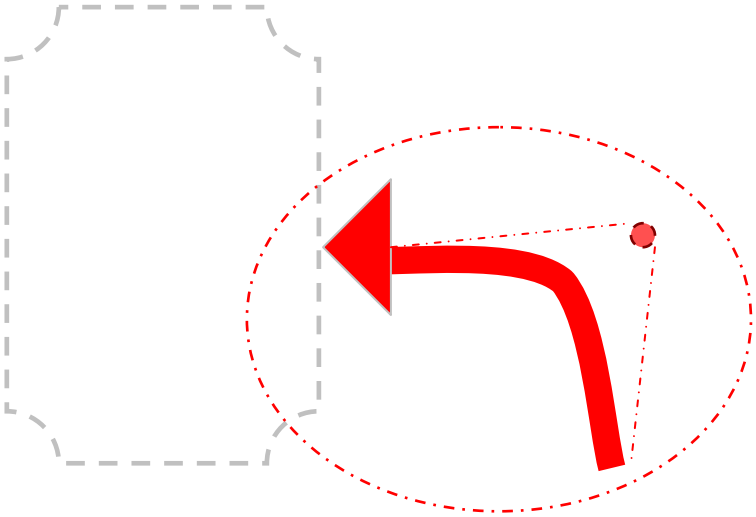
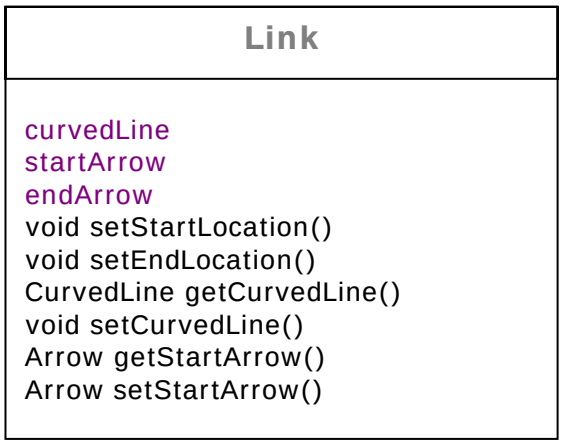


Diagram :



SVG definition :

```
<g id="link1" dpi:component="Link" arrowStart="#arrow1"
   arrowEnd="#arrow2" path="#path1" stroke="blue" stroke-
width="2"/>
```

MovableElement

Utility :

Used to make an SVG component *Translatable* by the *Pointer*. It's usefull to assign it to a SVG group, so every Translate action received by its children will be catched by the group and therefore will move the entire group with its children

Use :

Not relevant

Characteristics :

Not relevant

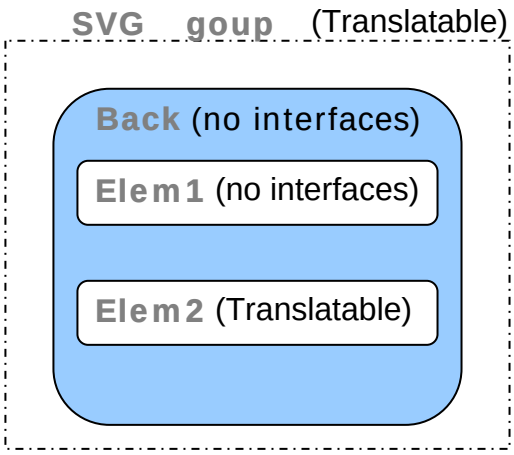
VarPersistants :

none

Mandatory Interfaces :

Translatable

Example :



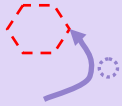
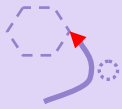
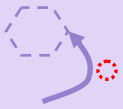
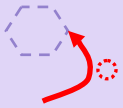
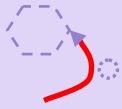

```
<g>
    <Back …/>
    <Elem1 …/>
    <Elem2 …/>
</g>
```

Incoming Translate Action on on…

- **Elem1** or **Back**
  All elements in the group move
- **Elem2**
  Only Elem2 moves

Diagram :
Not relevant

4.4 Components visual sumary

Component			Component	
AnchorPoint			GraphicContainer	
Arrow			LineHandle	
ContainedGraphic			Link	
CurvedLine				

4.5 Interactions

Interactions group methods in an Interface
The goal is to regroup methods related to the function of a certain cursor

Attacher

Utility :
Put in relation object that can to the PullLine Action (see LinePullable interation)

Execute Action / Query :
PullLine (Interface LinePullable)

Selector

Utility :
Used to select one or several objects

Execute Action / Query :
SelectAction (Interface Selectable)

Slider

Utility :
Used to slide *Arrow*s on the border of objects

Execute Action / Query :
GetBorder (Interface BorderFindable)
BorderSlide (Interface BorderSlidable)

Translater

Utility :
Used to move components and arrows in *freeSliding* (when an arrow is not attached to a border)

4.6 Utilities

varPersistant

Utility :
The goal is to be able to save crucials informations contained in classes to XML that are necessary to reconstruct a saved scene when it's loaded.
VarPersistant can handle variable of different types that will be synchronized with a specific attribute '*parameters*' in the related

SVG node. When the variable is modified, it codes the string equilavent and update the *parameters*. When a scene is saved all the dom tree is saved to XML, and therefore the attribute of the varPersistant. Also, when a SVG file is loaded the value of the variable is possibly filled according to the attribute content if found.

Use :
When a varPersistant is created, three importants informations must be passed to the constructor :

- The node in which varPersistant will create/modify the attribute *parameters* that usually is the *wrappedElement* of the component or interface
- A string description of the context, for example the class that belongs the *varPersistant*
- A string description of the role of the variable, that usually correspond to the name of the *varPersistant*

Then the *varPersistant* can be modified with the given methods

There is no restriction the user to set the parameters attribute in the SVG model in order to define preestablished setting such as Sticked elements, attached border/arrow, number of ContainedGraphics, and more.
Multiple variable can be defined in one *parameters* attribute, the format is the following :
Parameters=
'Context1=(VarName1, value1)***{***, ContextN=(VarNameN, valueN)***}***'

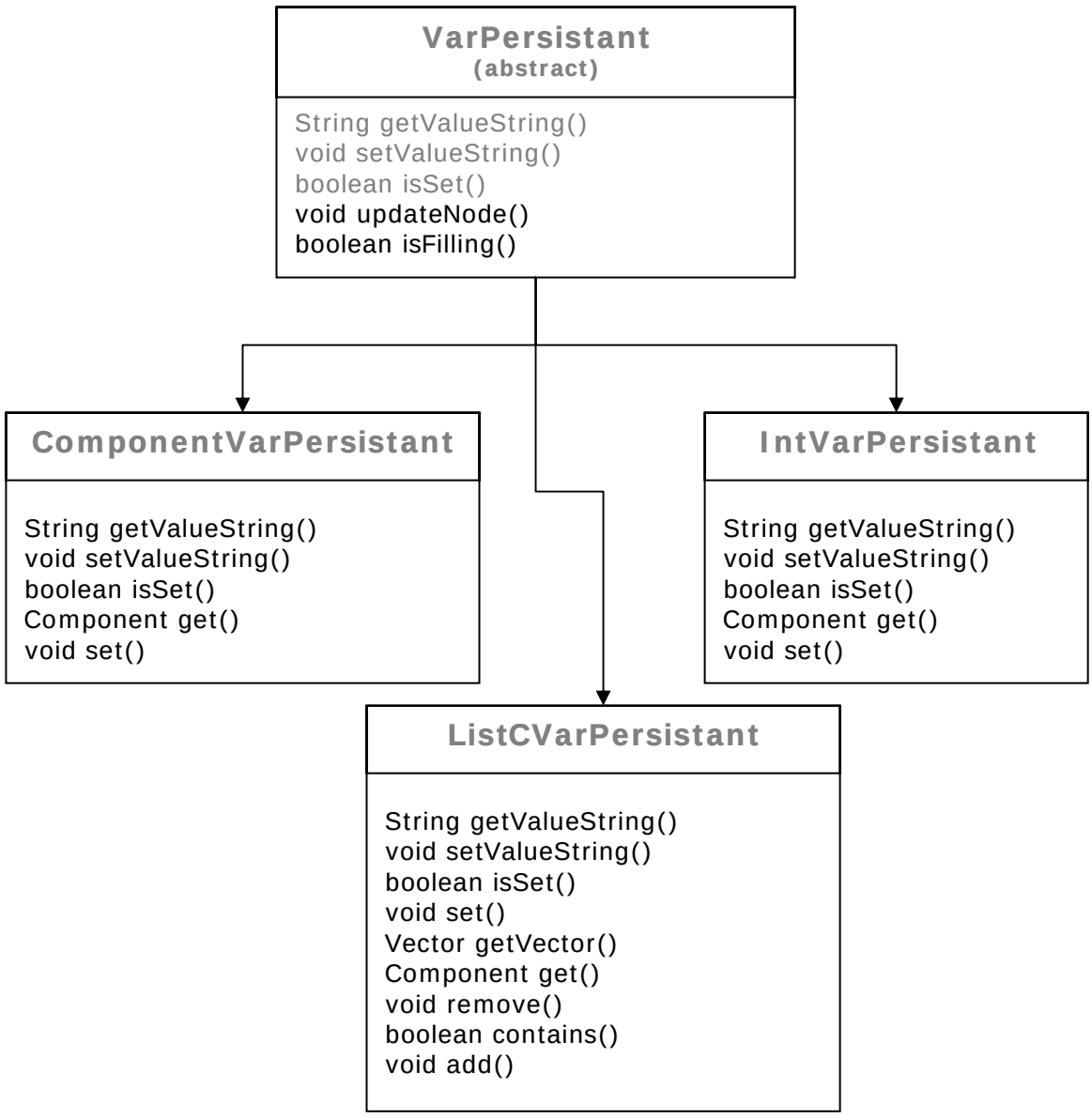
Subclass ComponentVarPersistant :
Variable Type : *Component* class
String format of the *value* : SVG Id of the SVG node of the component
When a scene is loading and a *ComponentVarPersistant* is created (for instance in the constructor of a *Component* class), it put back its initialization with the method *InvokeLater* in order to be sure that all the component have been loaded in the scene before trying to find the Component that correspond to the SVG Element
Format. Because of this, the value will not be available before the end of the loading. This availability can be checked using the method *isFilling()* that returns true is the variable is still not filled

Subclass ListCPersistant :
Variable Type : *Vector* class containing *Component*s
String format of the *value* :
'*componentNodeID1****{***, *componentNodeID1****}***
The same problem than *ComponentVarPersistant* occurs for the loading of the *Component*s

Subclass IntVarPersistant :

String format of the *value* : string representation of the integer

Diagram :



IdManager

Utility :

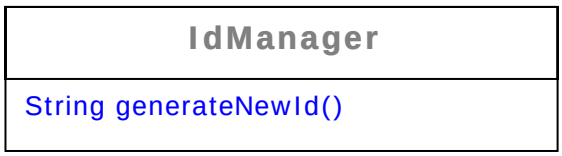
Used to create unic SVG ID's for the nodes. Could be very usefull to handle the id conflicts generated by the loading of a SVG file in the scene (instanciation new components). Here is an example :

- Group of components loaded and having crossed references
- The instanciation make copy of nodes. If one Id is in conflict with the existing scene, or if the file has already been loaded, some id must change, and all references in the loaded group.

This functionality could also be handled by the InstanceManager and is not yet implemented.

Use :
To generate an Id, a *Component* or an *SVGElement* must be passed. Both give the string representation of the corresponding *Component* class (toString()). Then the Id is randomly changed until it is unic.

Diagram :

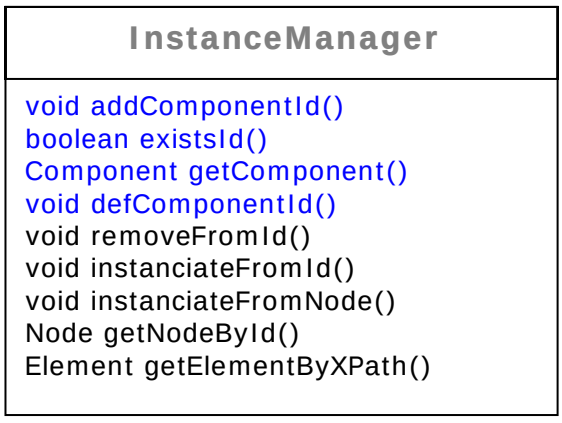


InstanceManager

Utility :
Group methods to extract, copy and remove SVG nodes by differents means. This class also could handle Id conflicts during the loading of a file in the existing scene. (see IdManager)

Use :
Not relevant

Diagram :



LineDescription

Utility :
Used to represent the description of a *curvedLinePath* and provide methods to modify the path and convert from text attribute to structure and vice-versa. Though it represent all important attribute

of a path, there is also a method to create a *CurvedLine* component from the *LineDescription*

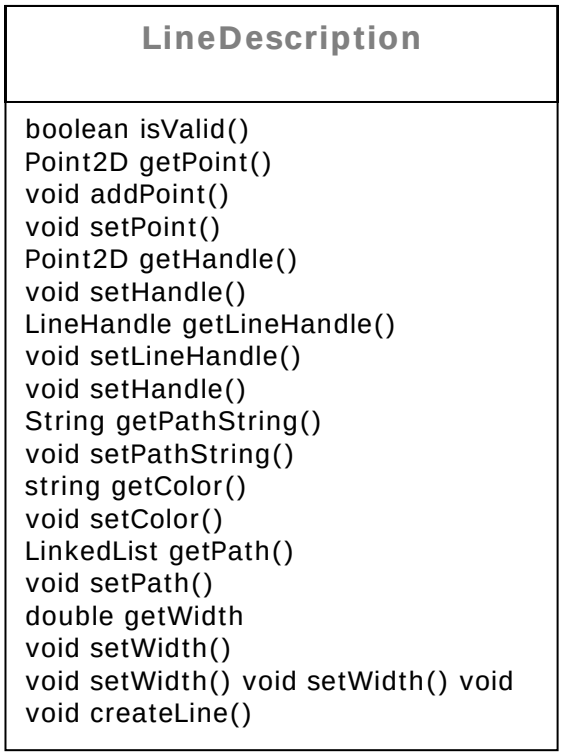
Use :
The lineDescription is used in two different ways :

- It is part of the CurvedLine class, and this latter use it to synchronize SVG text representation of the path and class datas
- It can be use to create a new curvedLine from a description (the LineDescription will be also used like the first way after *CurvedLine* creation)

To use it to create a CurvedLine, it's mandatory to fill the following properties before calling *createLine()* :

- Path
- Color
- Width

Diagram :



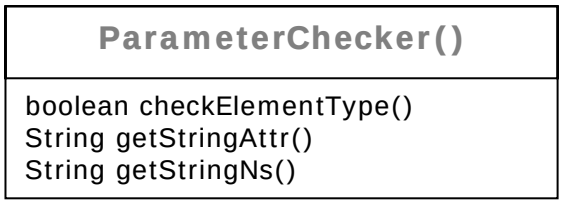
ParameterChecker

Utility :
Used to facilitate the usual checks and retrieval of attributes from a SVG node during its *Component* initialization (generally used in getParameters()) and show errors messages on the console

representation of the integer

Use :
Must be initialized the the node to be checked

Diagram :

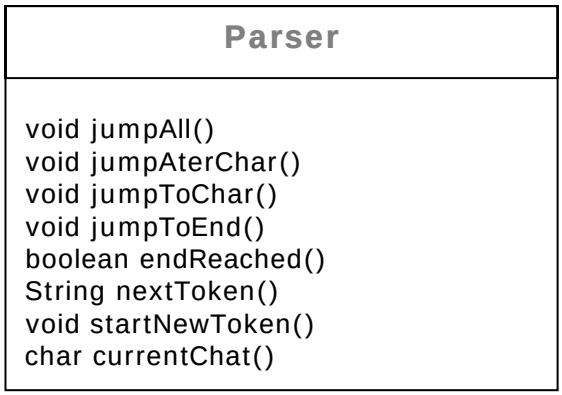


Parser

Utility :
Used to make simple parsing on strings

Use :
Used to parse value from SVG attributes such as *path*s.
The methods are commented

Diagram :



4.7 Tableau récapitulatifs

Interfaces

Interface	Utility	dependencies	varPersistant	comments
BorderSlidable		GetBorder	attachedComponent	one subclass for arrows, and one subclass for general components
DirectionAdjustable	orient according a point	GetOrigin		
LinePullable	draw temporary line			
Positionable	set position			
Stickable	group movement	Translate	stickers	
Translatable	move			
BorderFindable	extract border			only polygons, rectangles, circles, lines
OriginGettable	get origin taking in account SVG origin definition			

Components

Component	Utility	Interfaces	comments
AnchorPoint	Extract their border	BorderFindable Stickable	can only be basic shapes
Arrow	conexion between border and link	ArrowBorderSlidable Translatable DirectionAdjustable OriginGettable	can only be SVG polygons
ContainedGraphic	graphics for GraphicContainer	Translatable Positionable OriginGettable Locatable	
CurvedLine	Link body		can only by SVG curved path element
GraphicContainer	Graphically represent numerical informations		can only be SVG circle or rectangles
LineHandle	deform curvedLines	Translatable Handlable Locatable	automatically generated
Link	regroup informations for link instanciation		defined in a SVG group
MovableElement	just able to move	Translatable	

5. State of the project

5.1 Analysis

Here is a a detailed analysis of the important aspects of this project for those who would wish to develop it further.
After the implementation of ProBXS, the following concepts pointed out because they were sources of problems :

- **Interface dependancy problem**
- **Types of behaviours (useless interface or components)**
- **Data of components (dinctinction of data type)**
- **Classes of components (standard or helpers)**

For each sections of the analyse, solutions will be proposed to solve the problem.

Interface dependancy problem

There is two to opposed view to optimize the architecture of interfaces :

- To make them modular
- To make them independant

For instance the interface *DirectionAdjustable* is dependant to *OriginGettable*.
So when a user create a new component, that must be *DirectionAdjustable*, he must also include dependant Interface. The main bad consequence is that a knowledge about the interfaces dependencies is needed by the user to assign interfaces, which is a bad point for the usability of the project. Though when interfaces are chosen for a component, we don't care about possible dependant interfaces, the only interface needed is the one which serve the wanted behaviour

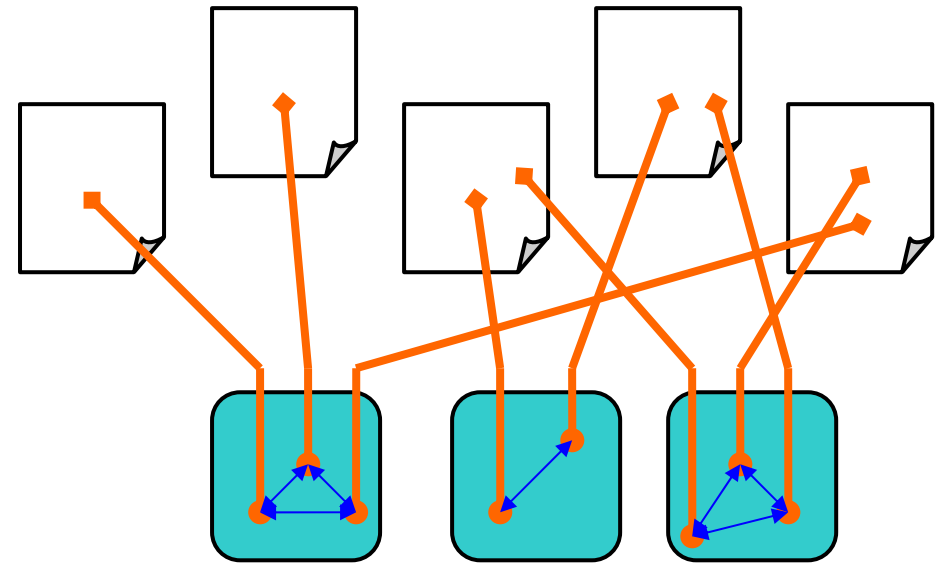
Factors that could solve the dependancy problem :

- Rewrite code
- Change place of functionalities

Rewriting the code :
For our example, instead of using OriginGettable interface, we could simply rewrite the four lines of code that allow to get the origin of the component.
The advantage is that it would ma ke totally independant interfaces.
The disavantage is that it would become a nightmare when code have to be changed.
This solution is clearly not usable !

Change the functionalities places :
To avoid the Interfaces to need eachother, all interactions between interfaces is done at the component level.

Functionalities pushed to components :



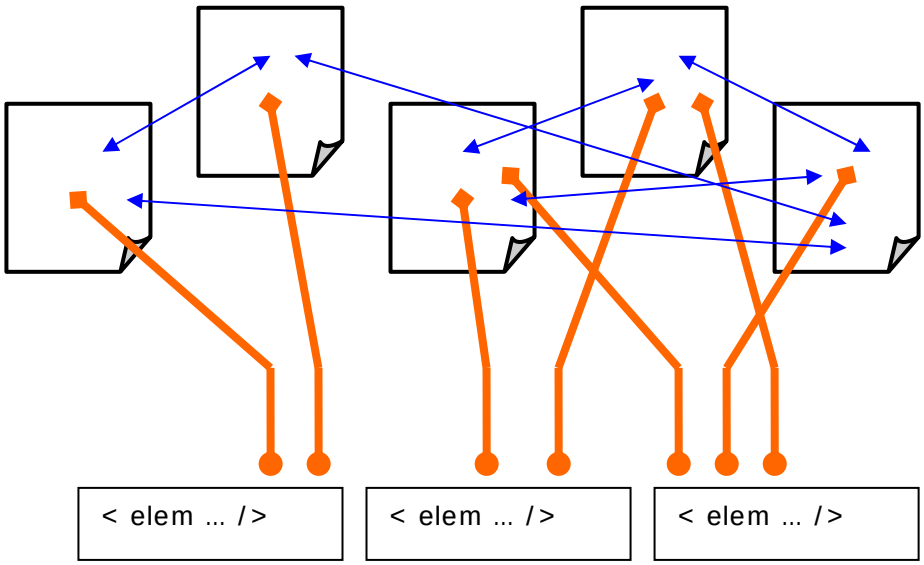
Interface classes

Component classes

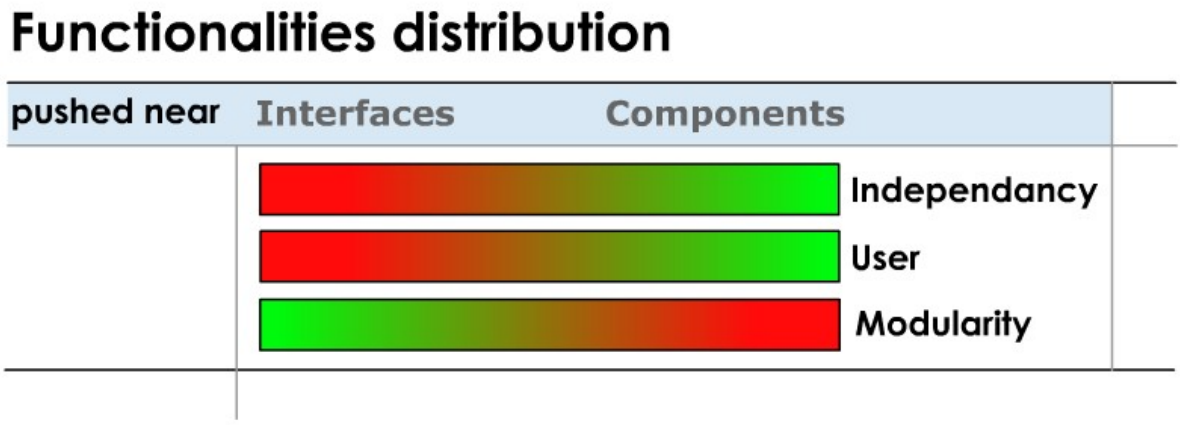
Here we can point out the importance of the presence of component. This situation wouldn't be possible if interfaces were directly assigned to SVG element. As said in the section Architecture/Component, *Component*s a necessary to put behaviours inherent to type of component.

Here is the other extreme that we'd like to avoid. It generate a lot of dependancies at the Interfaces level.

Functionalities pushed to interfaces :



Actually, none of those extreme are possible because the priority for the place of functionalities is given to the *Type of Behaviours* (see Architecture/Components)
As we can see, functionalities distribution as impact on dependancy, knowledge about interface dependancies by user and modularity (reusability of the code of the behaviours)



Possible solution : InterfaceManager :
Here is a solution that would solve the problem of the load of knowledge needed by user to use interface.

Each Interface would handle a InterfaceManager initialized with the names of dependant interfaces.

Interface can only call interfaces through the InterfaceManager, and only for thos which have been declared.
User wouldn't have to care about dependant Interfaces anymore when he create new Component.

Useless interface and components

There are two types of behaviours : sharable and non sharable (see *Architecture/Component*)
One of the goal of using an architecture based on interfaces consuming actions/querie is to permit some behaviours to be shared by many components. However, two kind of errors can be done :

- To make an Interface that will be usefull for only one component, in that case the functionallity should be implemented as a method in the Component class.
- To implement functionalities in a component class which could be used by other kind of component in the future

When the user create a new component or a new interface, he must keep in mind what kind of behaviour he is creating.
Those two errors have been done :

- The interface *Handlable* (ability to deform curvedLines) can only be used by the *LineHandle*s, moreover the user shouldn't care about this interface because the handles are generated automatically
- The *attachTo()* method in the component *Arrow* should belong to any component able to slide on a border, that is to say the interface *BorderSlidable*

All the architecture should be verified again with that principle in mind

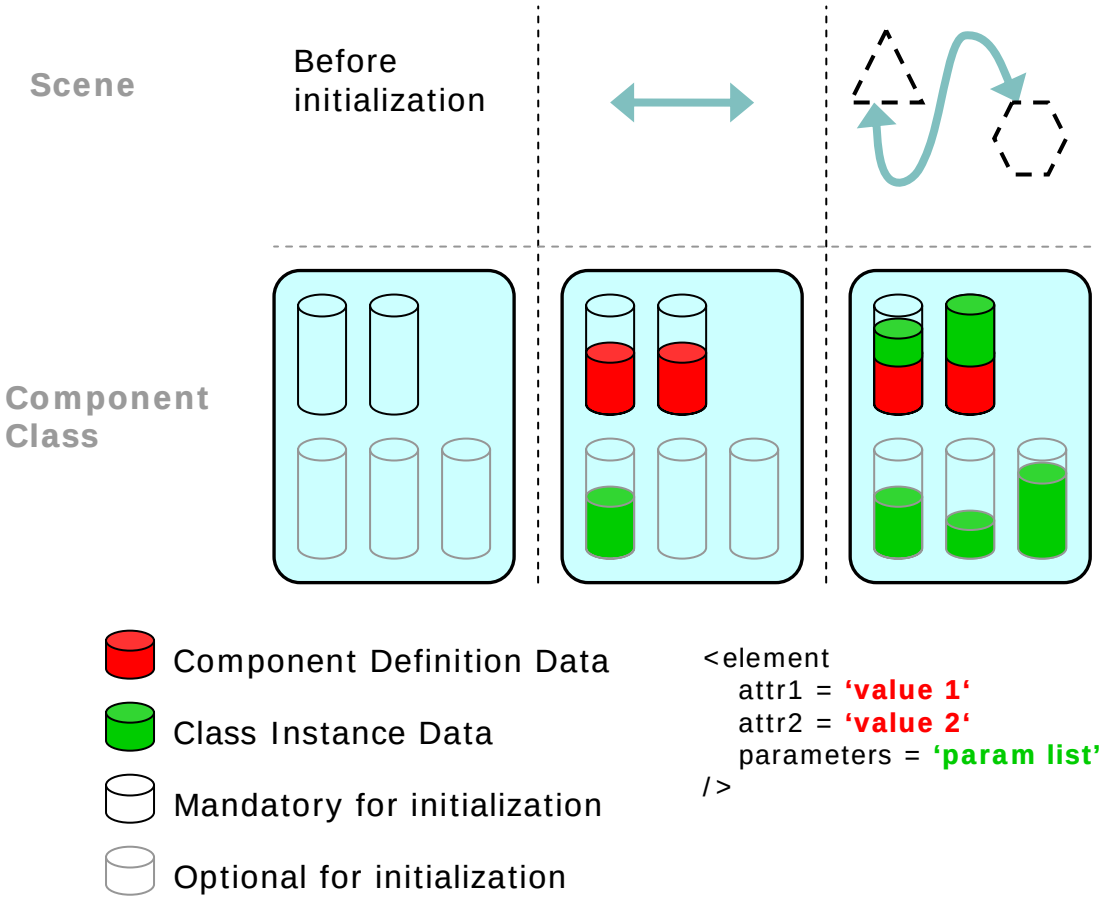
Clarify the distinction between CDD and CID

As already said, there is two kind of data a component should handle (see *Architecture/Component/Component Data*) :

- The Component Definition Data (CDD)
- The Class Instance Data (CID)

In this section we will analyse the necessity of this distinction.
The following question must be answered : Do we need to distinguish the mandatory data to initialize a component, and the optional ones that could have be added after a scene save ?
Advantage of making a distinction :
If there where no distinction, all parameters would be saved in attributes with different names.
If the original data is kept, and modified data is stored in the parameters attribute, it's easy to extract the original SVG model just by removing the *Parameters* attribute.

To understand better what represent CDD and CID, here is a graphical representation of the component class according to the scene :



Proposed solution :
The distinction seems to be usefull but instead of having a method for loading CDD (*getParameters()*), why not to to let CDD and CID to be handled by VarPersistants ? If CDD and CID must be differenciated, that is to say, the original data SVG model have to be kept, a differenciation would have to be considered inside the class VarPersistant. Then it would be better to handle conflicts.
Another advantage would be that each Variable directly represent the node it is attached to and the initialization of all variables would be automated just by putting the name of the attribute.
Probably then, VarPersistant system will have to be complexified to handle conflicts, dependencies, etc.

Differentiation normal components / helpers

It's possible to differentiate two kind of components :

- The components designed by the used that represent concepts of the graphical concrete syntax

- The components used to help the user to edit the components, such as *lineHandles*, *PulledLines*, selection rectangles, future combo boxes, tool boxes, etc.

The first ones, depends of the language of the user. In opposite, second ones can be reusable in any languages.

It has been decided to handle both kinds of components exactly in the same way to allow a maximum of flexibility. The only thing that can make their distinction is the file in which they are defined.

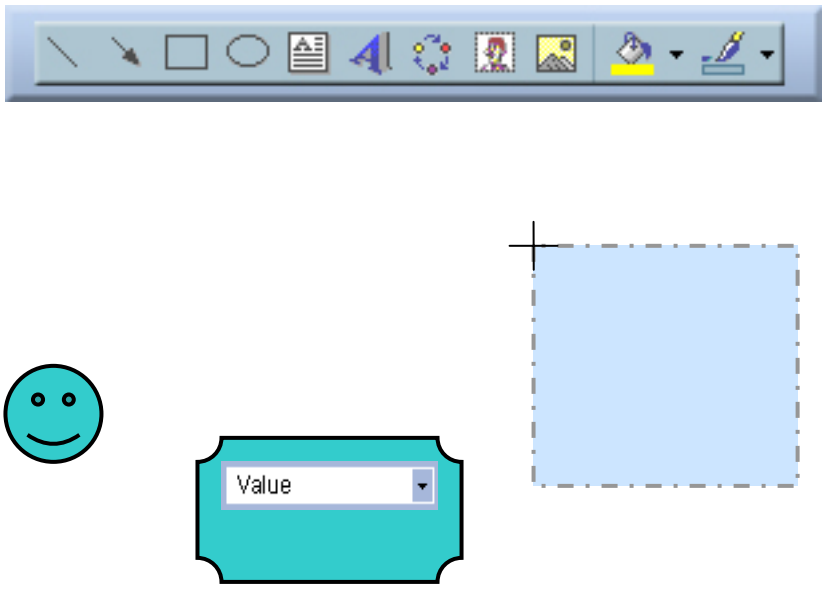
Maybe in the future of the project, it may be good to make something that distinguish them better (other class of component ? not defined with dpi:component attribute ?). It would permit to special considerations compared to normal components. For instance :

- Ability to appear only then certain component are activated or selected (LineHandle, comboBox, PulledLine, future selection rectangle).
- Ability to be instanciated from parametered templates and would not entirely be defined in SVG files. It would avoid to pollute SVG files with thousand of components.

Also the fact that those helpers component are saved exactly the same way as the others is not very “goodlooking”.

Another argument to differentiate those component is that the user only care about his graphical language, maybe he doesn’t want to define the shape of the helper component that should be always the same. There is probably a good solution that do a good compromise between flexibility and “user-friendlyness”.

Here is a representation of some reusable and non-reusable (cyan) component :



5.2 Corrections

In addition of the proposed correction in the Analysis section, here are some other ones to consider.

Architecture

Interfaces and hierarchy :
Some interface are partly the same each other. Interfaces should be putted in hierarchy in order to group those which are likely the same. For instance, Translatable do the same translation than BorderSlidable, expect that this latter has a constraint. The interactions should be verified again with that principle in mind

Interactions :
Basically, *SVGInteractions* have been added to the DoPIDOM architecture to group actions and queries that the cursor can trigger by kind of behaviours.
Finally, interactions are related to the Interactors (cursor) the same way than Interface are related to component. Every Interactor should be represented by a picked tool in a toolbox. The interactions should be verified again with that principle in mind

GraphicContainer :
- Make subclasses corresponding to each kind of positionement
- Reorganize Architecture and methods distribution over classes and subclasses

Limitations

AnchorPoints :
When an AnchorPoint is defined in SVG, the attribute linkType must be specified. However the AnchorPoints should be able to anchor any kind of links, and this attribute is useless

Arrows :
To only SVG element that can be an Arrow is the polygon element. All elements should be allowed to be arrows.

Minors corrections

Initialization :
When something is added to the scene, a listener is triggered to place the pointer at the end of the DOM tree so the cursor can always be visible. The initialization for this listener is done in the listener of the Pointer. A more appropriate place have to be found to initialize general things.

SVG
Every new attribute extending SVG should be placed in different names spaces

InstanceManager :
All methods should be turned static

Interactions :
The translations of arrows in the interaction *Translater* must be incorporated to the *TranslateRunner* (optimisation for the translation)

Bugs

BorderFindable :
When a border is not centered to zero in the SVG definition, the BorderSlidable doesn't slide properly

VarPersistant :
Each time a node is updated, the new parameter string is placed at the beginning of the parameter attribute instead of replacing the old value. This bug doesn't prevent the system to work because only the first matching parameter is read, but the bad point is that very long parameters are generated espacially for the GraphicsContainer. This bug is easy to correct.

General

Errors :

- For RuntimeExceptions, the error should be related to the Action/Query, and not the Interface
- In the beginning of the project, warning and fatal errors was both signaled in the *err* output stream of the system. Fatal errors should raise exceptions in order to be located easily

Correction sumary

- Types of behaviours
  - Reorganize place of functionalities
  - Delete useless interface
  - Delete useless components
- Datas of components
  - Clarify the needs (separation CDD/CID ?)
- Independancy problem
  - Reduce implicit knowledge of dependancies needed by user
- Classes of components
  - Dont pollute SVG files with helpers
- Organize Interfaces in hierarchy
  - Regroup interfaces by similarities
- SVGInteractions
  - An Interactor must represent a tool

5.3 What have to be added

Links :

- Possibility to add intermediate points on *CurvedLines* by mean of an interactor
- Possibility to move intermediate points on *CurvedLines* by mean of an interactor. For now, only handles can deform *CurvedLines*
- The handles should appear only when the links are selected
- Possibility to create Links between two components. For that new components have to be instanciated from models with *InstanceManager*
- Add the possibility to set different types of lines. For now only *curvedLines* can be chosen as a Link Body

Behaviour assignement :

Give the possibility to assign single interfaces to elements when defining the SVG scene instead of components (that are group of Interfaces with methods). For instance, that would allow to have some *AnchorPoints Translatable*, and others not

GraphicContainer :

- For the in-line positioning, add the possibility to align to left, right, top or bottom of the GraphicContainer

Other needed omponents :

- List
- ComboBox
- Better TextField

Selection :
A selection rectangle to select more than one component with only one mouse click.

Undo/redo :
Methods to handle undo and redo are provided in interfaces, but they're not been filled

6 Conclusion

6.1 Encountered problems

Helpers

The problem with the helpers (see corrections/architecture) was present from the beggining

Architecture

One of the advantage of DoPIDOM is to have a very modular architecture. It is important to try to keep this modularity. Difficult decision about the design of the architecture had to be done each time a new component or interface was created to find which behaviours it is possible to group. For instance, is it possible to associate the behaviour of a GraphicContainer with a List, or the translation of arrows on a border and the confinement of a component inside limits with the same system of constraints but in different dimensions ?
The future components to be designed will probably encounter this problem much more.

Independancy

An important dilemma is the problem of interface independency told in the section "*Analysis*". DoPIDOM has been made to be very modulable by sharing behaviours. The goal is to try to keep this modularity, but this latter is sometime in contradiction with the Independancy of interfaces.

6.2 Contribution to language driven development

After the final analyse of the project, a lot of concepts have been taken from mistakes, and then propositions have been made for the rest of the project. For sure, I don't pretend they are correct either and have to be reconsiderated by more experienced persons.
I hope that my work and my report can help in identifying the main problems that can be encountered with the design of components to represent graphical concrete syntaxes. Maybe it will facilitate the creation new features for DoPIDOM or a specialized toolkit to model modeling languages.

6.3 What I liked and regretted

I enjoyed working on that project, because it gave me a good experience on big project architecture. Even if didn't produce anything amazing, I learned a lot about design and found it important because students from Computer Science have a lack of knowledge on making reusable architectures.
For sure it was a pleasure and a real interest for me to work in cooperation with Frederic who I spent time with to solve daily headaches.

I regret having spent too much time on solving implementation problems than having designed interesting architecture for components and interfaces (hierarchy).

6.4 Acquired knowledge

- Working with Eclipse (a big discovery !)
- Some Patterns
- Concepts on nowadays Java programmation
- Better technics to work on big projects, that is to say :
    - Reflex of writing down problems and decisions tree for further analyse
    - Reflex of quoting and classifying ideas
    - The case study helps in identifying weaknesses
- Better aptitude to analyse
- Better aptitude to put concepts in hierarchy

6.5 Remarks

Remark about the number of uncorrected things

After three month of work on this project, I realize that an enormous part of the project should be refactored.

- Dependances between interfaces
- Interactions should have hierarchy (classes and subclasses). For instance Translatable and borderSlidable both translate objects, but one has constraints.

Moreover, there are many minor bugs or correction to do. Their quantity and the lack of time to finish this project had for consequence that it was much more usefull for the continuation of the project to list them all than to correct only few of them (and not have time to list them)

Remark about time and work balance

The amount of time required by the project was probably much more than 12 hours/week (3 months), for this reason it's unevitable not to have a complete work. Though I did my best and had to balance equitably my work time in two axes : In one hand, I had to try to respond to Frederic's demand and make a maximum of features (we needed even much more features), and in the other hand I wanted to make efforts on making a good code architecture and reusability. Unfortunalety the second part was underdone in order to assure a good synchronization with Frederic's work but can be seen as an advantage for the future work on the project : The more feature have been done the larger is the view to do a good refactoring general architecture.

Thanks

Thanks to my advisor Frederic Fondement who helped me with patience during the semester !