UML Layout – documentation and tutorial

1. General description

Program *layout* reads a file describing relations (associations) among classes and their inheritance (how the classes are derived from one another). It generates the full geometry of the corresponding UML class diagram.

To generate a pleasing diagram is not an easy task, because its objective is not to optimize a mathematical formula, and any attempt to formalize the task leads to NP complexity. The program must solve the following problems:

(1) It must decide which class or classes should be used as root of the diagram.
(2) It must place the boxes representing the classes on the screen, while anticipating the connections between the boxes and the size of labels both inside the boxes and on the connecting lines.
(3) It must route the connecting lines in a pleasing manner while preventing overlaps.
(4) It must add arrows, multiplicity symbols and labels to the connecting lines.

The program does not generate the diagram itself, but it has two options:

- The default option generates file display.log which breaks the resulting graphics into three basic geometries – line, box, and text label. A special Java program, *JavaDisplay*, then generates the picture.
- If you run *layout* with the –s option, the program generates file display.svg which you can view with any web browser.
-

At the moment, both options produce an identical diagram. The only advantage of the current *JavaDisplay* is that it has a smaller frame and therefore can accommodate slightly larger diagrams. The *layout* program attempts to place the entire diagram within the boundaries of the screen and uses various measures and scaling to achieve this objective. However, if the resulting diagram would become unreadable (font too small or overlapping labels), the resulting picture may expand beyond the boundaries of the screen and its right or bottom section may not be visible.

As this program will evolve, the two options may eventually provide different diagrams or additional features such as SAVE or ZOOM or, if one proves to be clearly superior we may drop the other option.

The current version does not work with the XMI format – neither for the input nor for the output. The input format is compact (one line per association), human readable, and using the syntax from *IN_CODE modelling* which makes the link between the two programs natural and simple. However, we recognize that XMI format would allow a cooperation of not only *layout* but also of *IN_CODE modelling* with other UML tools, and we plan to add the XMI interfaces in the near future.

Unfortunately, this version does not provide any simple mechanism for saving the resulting diagram (whether produced with –s option or by *JavaDisplay*). The only way to save it is to remember the entire screen with Alt+PrtScr and then paste it into a graphical tool such as Adobe Photoshop with cntrV

All these problems are only shortcomings of the first version and will be removed soon.

SYNTAX: `layout [-s] paramFileName inpFileName`

where paramFile is a 3-line file describing the size of the screen and the requested font of the labels in the diagram. This file can either be manually coded, or generated automatically by invoking *GenParamFile*. Typically, you create this file only once when you are installing *layout* on your computer.

The *layout* program is in a single source file, layout.cpp. It has been coded in C++ and it is available as Open Source through SourceForge. It uses only basic features of the C++ language and uses neither templates nor external libraries; it should compile on any platform. The penalty for this is that, in spite of generous comments, the program is not easy to read or maintain. We are planning to replace its raw pointer- and array-based data structures by associations from the IN_CODE library (alib), and that should significantly improve its internal organization and clarity.

2. Example

The first example in the tutorial for IN_LINE modelling deals with 3 classes: Department, Employee, and Manager. When the IN_LINE code generator *codegen* finds the following segments of the code

```
class Manager : public Employee {
    ...
  };
  Association LinkeList1<Department,Employee> empl;
  Association LinkedList1<Department,Department> dHier;
  Association SingleLink<Department,Manager> boss;
  Association Name<Employee> eName;
```

it generates file layout.inp which, as the name suggests, is the input for the *layout* program:

```
u1-* LinkedList1 Department Employee empl ;
u1-* LinkedList1 Department Department dHier ;
u1-1 SingleLink Department Manager boss ;
Name Employee eName ;
Inherits Manager Employee ;
```

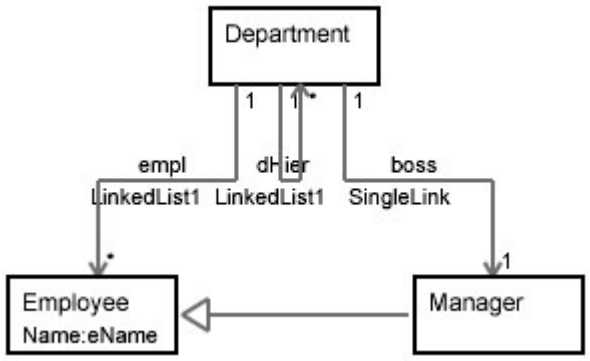
The lines specifying the associations are practically the same as they were in the original source, except the first group of characters, which *codegen* retrieved from the *registry* file in alib/lib. The Name relation is more an attribute than as true association and is therefore

treated differently – it does not have the leading 4 characters. The inheritance record is retrieved directly from the definition of class Manager.

After running program layout with file layout.inp as its input,

```
layout –s param.txt layout.inp
```

we get file display.svg, and when we can view from the Internet Explorer, and we get this picture



If we run program layout without the –s option,

```
layout param.txt layout.inp
```

we get file display.log which breaks the diagram into basic geometries:

```
layout.inp
A 1024 768 12 11
B 140 48 224 86
T 147 65 Department 12
L 153 134 84 134 0
T 104 130 empl 11
T 80 146 LinkedList1 11
L 175 134 185 134 0
T 163 130 dHier 11
T 142 146 LinkedList1 11
L 207 134 281 134 0
T 230 130 boss 11
T 209 146 SingleLink 11
L 153 134 153 86 0
T 157 98 1 11
L 175 134 175 86 0
T 179 98 1 11
L 185 134 185 86 1
T 189 98 * 11
L 207 134 207 86 0
T 211 98 1 11
L 84 134 84 182 1
T 88 178 * 11
```

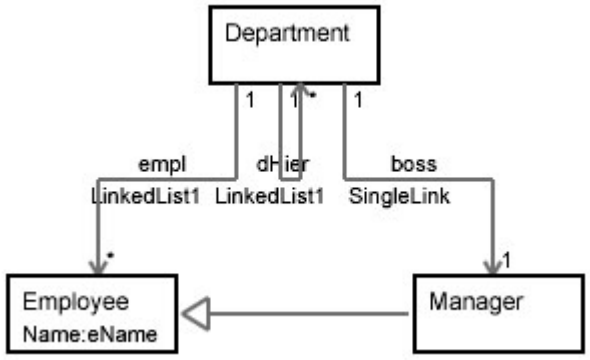
```
L 281 134 281 182 1
T 285 178 1 11
B 39 182 123 220
T 46 199 Employee 12
T 46 215  Name:eName 11
B 242 182 326 220
T 249 199 Manager 12
L 239 201 126 201 2
T 179 197 - 11
T 154 213 Inherits 11
T 130 197 - 11
T 228 197 - 11
```

In this file, character T specifies a text label, L a line, and B specifies a box. All coordinates are in pixels.

Then, when we call

```
java JavaDisplay display.log
```

we get the same diagram at the top-left corner of the screen:



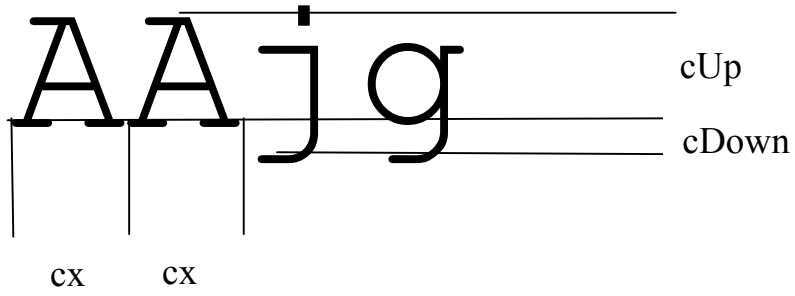
3. Parameter file

The parameter file has only three lines, for example:

```
1024 768     ... screen dimensions (x,y) in pixels
12 7 13 4    ... font description for class names
11 7 12 4    ... font description for all other labels
```

where each font description has 4 values: `fontSize,cx,cUp,cDown`
The assumed font type is always Courier New.

where fontSize is the commonly used font number, and cx,cUp,cDown are the dimensions of one character in relation to the text line:



When you install the layout program, you either can create the parameter file with any text editor or you can generate it automatically by going to directory layout/javadisp and then typing

```
Java GenParamfile 12 11 > param.txt
```

where 12 and 11 are the two fonts, and param.txt (or any other name) is the name of the parameter file. You do not need to do it again, unless you switch to a different screen.

4. Input file

Typically, the input to the layout program is file layout.inp generated by IN_CODE code generator, *codegen*. It does not matter whether this file was created for C++ source (by alib/codegen) or for Java source (by jlib/codegen). This file which describes the logic of the UML class diagram is the same, and it has the same format.

The file consists of three types of records, which can be used in any order. However, the order in which the names of classes are introduced can sometimes influence the layout of the diagram. If the internal algorithm runs into the situation where several alternatives lead to the same quality of diagram, the algorithm tends to select the alternative which reflects the sequence of classes from the input. The reason behind this is that the human description may be based on some additional reasons which the algorithm cannot see, and even if this is not the case, it helps the user of the program if the picture resembles his/her mental picture of the architecture. Each record can run over several lines and must end with a semicolon.

The first record type describes an association:

```
code associationType partClass1 partClass2 assocName ;
```

for example

```
u1-* LinkedList1 Department Employee empl ;
```

where *code* is a four character code which expresses whether the association is uni-directional (u) or bi-directional (b), and the multiplicity of its ends. For example 1-* means one-to-many, 2-* means two-to-many, and 1-1 means one-to-one. Multiplicity 0

(for example in u1-0) is used for member-like structures not dependent on the application classes, such as Name or Property. Data structures which work with only one class such as plain ring or tree without a holding object must use '=' instead of '-', for example u1=1. For traditional many-to-many associations with an additional *relation* object, the code is R*2*, R*3*, etc depending on the number of entities forming the association.

The default associations usually used in the initial planning when the implementation does not matter have capital U or B. In the final diagram, such associations are displayed without the association type. Examples of such associations are *Uni1toX* or *Bi1to1*.

AssciationType is the type of the association such as *LinkedList1, Hash* or *Vector1*. It must be a type registered in the alib/lib/registry file (for C++) or in the jlib/lib/registry file (for Java).

PartClass1 and partClass2 are the participation classes in the order which corresponds to their roles. For example since Department is the parent which has multiple Employee children, Department is listed first.

AssocName is the name which you use as a reference when you invoke the association in your code. For example, in C++, empl::add(d,e) adds Employee e to Department d, and empl::remove(d,e) removes e from it. In Java, the syntax would be empl.add(d,e) and empl.remove(d,e).

The second record type specifies Name:

```
Name holdingClass assocName;
```

for example

```
Name Employee eName ;
```

where assocName is the name which you use as a reference when you invoke the association in your code. For example, in C++ and for char *s , eName::add(e,s) adds name to Employee e, and s=eName::get(e) gets it. In Java for String s, the syntax would be eName.add(e,s) and s=eName.get(e).

Name is a special type of association. It is a reference leading to a variable length string. The UML diagram shows Name as a member of the participating class, because one of its possible implementations is adding a member which is an object of class String.

The third record type specifies inheritance. In C++, multiple inheritance is allowed:

```
Inherits derivedClass baseClass1 [baseClass2 baseClass3 ...] ;
```

for example

```
Inherits Manager Employee ;
```

In Java, two formats are possible:

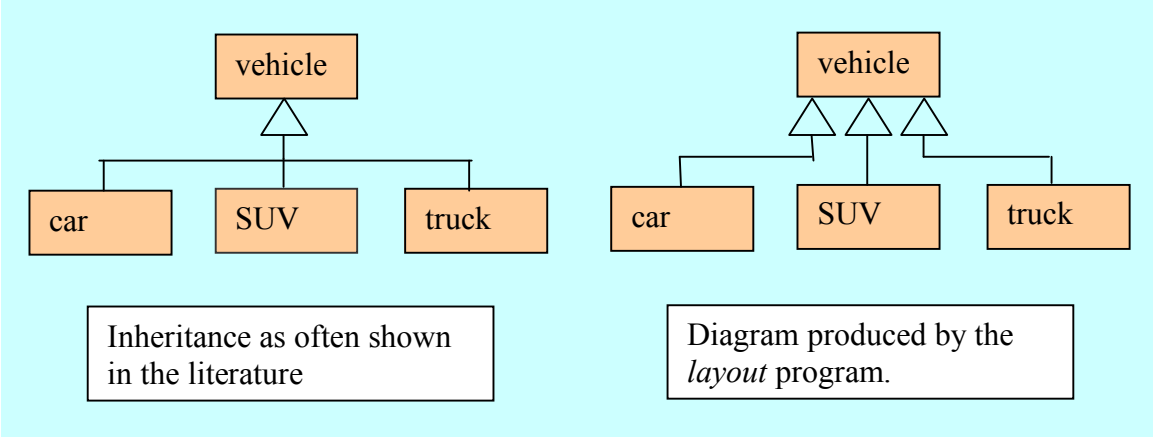
```
Inherits derivedClass baseClass [interface1 interface2 ...] ;
```

```
Implements derivedClass interface1 [interface2 ...        ] ;
```

for example

```
Inherits Paragraph Editable ;
```

Note that inheritance arrows do not merge as sometimes shown in the literature (e.g. "The Unified Modelling Language Reference Manual" by Rumbaugh, Jacobson and Booch). This is the display style used by most UML tools:



5. The log file

The purpose of the log file is to provide a simple and human readable format which could be easily converted to a diagram using any graphics library.

```
Line 1: name of the input file - to be used as the identifier
Line 2: A xScreen yScreen classNameFont otherLabelsFont
```

Remaining lines using arbitrarily formats starting with letters B, L, or T:

```
B xMin yMin xMax yMax
L x1 y1 x2 y2 arrow
T x y textString font
```

where all the x,y coordinates are in pixels. Note that the lines may be only horizontal or vertical and that they have a sense of direction. It means that x1>x2 or y1>y2 are valid coordinates.
The *arrow* code =0 for no arrows on the line,. =1 for regular arrow at the x2,y2 end, and =2 for the inheritance arrow assuming the base class or interface at the x2,y2 end.

For example:

```
layout.inp
A 1024 768 12 11
B 140 48 224 86
T 147 65 Department 12
```

```
L 153 134 84 134 0
T 104 130 empl 11
T 80 146 LinkedList1 11
L 175 134 185 134 0
T 163 130 dHier 11
T 179 98 1 11
L 185 134 185 86 1
  .......
```

6. Potential improvements

-- If the diagram has several natural roots, use them in the first row instead of selecting just a single root.
-- Allow to enhance the diagram by colours instead of the existing solid/dash lines.
-- Provide optional XMI output of the *layout* program.

7. Known problems

There are still some bugs which show in complex examples with many classes. For example:

(a) When using inp15, the connection between Net and Edge is missing, but the text is there.
(b) When using inp15, Net should be closer to Node than Track and Group. I thought the placement was well debugged, but maybe a bug crept into one of the sorting functions.
(c) When using inp17, connection between Net and Edge has missing vertical lines.
(d) Has the display of Names inside the class boxes disappeared or is it not visible as a selected option?