

MetaUML: A Manual and Test Suite

Copyright ©2005-2019 Ovidiu Gheorghies. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

This page is intentionally left blank.

MetaUML: A Manual and Test Suite

Ovidiu Gheorghies

February 2, 2019

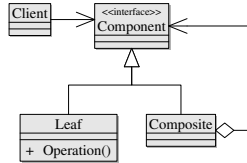
Abstract

MetaUML is a MetaPost [1] library for creating UML [2] diagrams by means of a textual notation. While presenting the inner workings of MetaUML, this manual doubles as a step-by-step tutorial. More importantly, its source code contains many useful examples of diagrams, ranging from the very basic to the more advanced and customized.

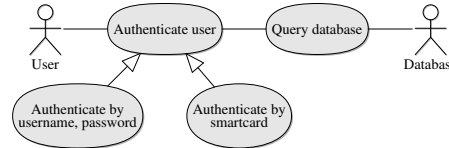
1 Introduction

Here is a quick MetaUML showcase:

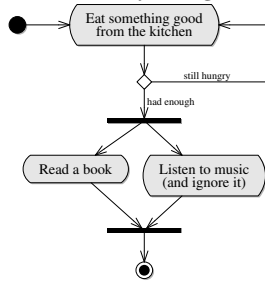
A Class Diagram



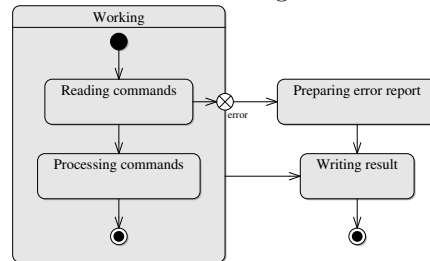
D Use Case Diagram



B Activity Diagram



E State Machine Diagram



C Notes



F Package Diagram



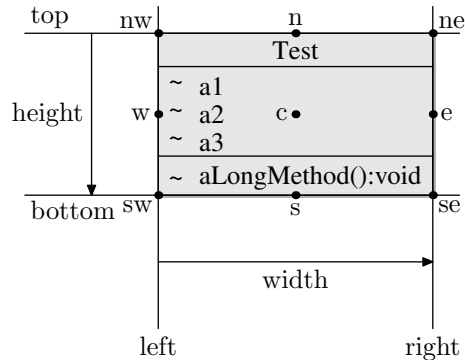
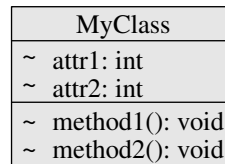


Figure 1: Layout properties of MetaUML objects. Here, a `Class` object is depicted.

The code that generates these diagrams is quite straightforward, combining a natural object-oriented parlance with the power of MetaPost equation solving. For example, a UML class is drawn as follows:

```
Class.A("MyClass")
  ("attr1: int", "attr2: int")
  ("method1(): void",
   "method2(): void");
```



```
A.nw = (0, 0); % optional, implied
drawObject(A);
```

This code creates a visual object, referenced by its name `A`, of the MetaUML-defined type `Class`. Object `A` has the following content properties: a name (`MyClass`), a list of attributes (`attr1`, `attr2`) and a list of methods (`method1`, `method2`). To set the object’s location, we assign a value to the so-called “north-west” point of the encompassing rectangle, `A.nw` — a point which in actual fact references the upper-left corner.

Every MetaUML visual object has the layout properties shown in figure 1. These properties may be used to set the location of any given object, either by assigning to them absolute values, or by linking them relatively to other objects via equations.

The following example demonstrates, respectively, the use of absolute and relative positioning for two classes, `A` and `B`.

```
A.nw = (0,0);
B.w = A.e + (20, 0);
```



After the objects have been drawn, it becomes possible to attach links to them. In a class diagram, inheritance or association relations are meaningful

links between classes, while in a state machine diagram, transitions between states can be used. Here is the general pattern used by MetaUML for drawing links:

```
link(<how-to-draw-information>(<path-to-draw>);
```

The “how-to-draw-information” is an object which defines the style of the line (e.g. solid, dashed) and the appearance of the heads (e.g. nothing, arrow, diamond). One such object, appropriately called `inheritance`, defines a solid line style and a white triangle head. The other parameter, the “path-to-draw”, is simply a MetaPost path.

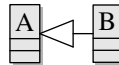
For example, the following call draws an inheritance relation from class B to class A.

```
link(inheritance)(B.e -- A.w);
```

The direction of the path is important, as MetaUML uses it to determine the type of adornment to attach to the link ends (if applicable). In our example, a white triangle, denoting inheritance, points towards the end of the path, that is towards class A.

Let us sum up with a diagram typical for MetaUML use. Firstly, we define the objects that we want to include in our diagram. Secondly, we position these objects relative to each other. Thirdly, we draw the objects. Finally, we draw the links, by referencing the layout properties of the previously drawn objects. Note that in our example the positioning of A need not be set explicitly because “floating” objects are automatically positioned at (0,0) by their draw method.

```
input metauml;
beginfig(1);
  Class.A("A")();
  Class.B("B")();
  B.w = A.e + (20, 0);
  drawObjects(A, B);
  link(inheritance)(B.w -- A.e);
endfig;
end
```



```
% 1. Define the objects
% 2. Position the objects
% 3. Draw the objects
% 4. Draw links between objects
```

As far as a user is concerned, this is all there is to MetaUML. With a reference describing how the UML elements are created, arbitrarily complex diagrams can be crafted.

2 Class Diagrams

A class is created as follows:

```
Class.<name>(<class-name>)
  (<list-of-attributes>)
  (<list-of-methods>);
```

The suffix `<name>` specifies an identifier for the newly created `Class` object (which, of course, represents a UML class). The name of the UML class is a string given by `<class-name>`; the attributes and methods are given as list of strings, `<list-of-attributes>` and `<list-of-methods>` respectively. The list of attributes and the list of methods may be void.

An attribute or a method string may begin with a visibility marker: “+” for public, “#” for protected, “-” for private, and “~” for package private. The default visibility is package private.

```
Class.A("Point")
  ("#x:int", "#y:int")
  ("+set(x:int, y:int)",
   "+getX():int",
   "+getY():int",
   "-debug():void",
   "test():void");
drawObject(A);
```

Point
x:int
y:int
+ set(x:int, y:int)
+ getX():int
+ getY():int
- debug():void
~ test():void

To disable showing the visibility markers, use `Class_noVisibilityMarkers`, as shown below:

```
Class.A("Point")
  ("#x:int", "#y:int")
  ("+toString():String");
Class_noVisibilityMarkers.A;

drawObject(A);
```

Point
x:int
y:int
toString():String

2.1 Stereotypes

After a class is created, but before it is drawn, its stereotypes may be specified by using `Class_stereotypes`:

```
Class_stereotypes.<name>(<list-of-stereotypes>);
```

Here, `<name>` is the object name of a previously created class and `<list-of-stereotypes>` is a comma-separated list of strings. Here is an example:

```
Class.A("User") ();
Class_stereotypes.A("<<interface>>", "<<home>>");

drawObject(A);
```

<<interface>>
<<home>>
User

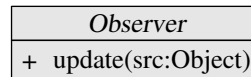
2.2 Interfaces and Abstract Classes

At times it is preferred to write the name of an interface in an oblique font, rather than using the “interface” stereotype. This can be easily achieved by using the macro `Interface`:

```
Interface.name(class-name)
  (list-of-methods);
```

Here is an example:

```
Interface.A("Observer")
  ("update(src:Object)");
```



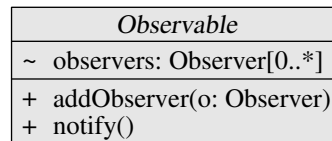
```
drawObject(A);
```

Since internally `Interface` treated as a special kind of `Class`, the code above is equivalent to:

```
EClass.A(iInterface)("Observer")()
  ("update(src:Object)");
```

Abstract classes can be drawn similarly using the `iAbstractClass` style:

```
EClass.A(iAbstractClass)("Observable")
  ("observers: Observer[0..*]"
   "+addObserver(o: Observer)",
   "+notify()");
```



```
drawObject(A);
```

If you prefer, you can use equivalent construct:

```
AbstractClass.A("Observable")
  ("observers: Observer[0..*]"
   "+addObserver(o: Observer)",
   "+notify()");
```

2.3 Displaying Class Name Only

If you want the empty methods and attributes compartments in a class not being displayed, one way is to set the spacing at their top and the bottom to 0:

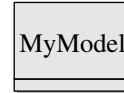
```
Class.A("MyModel")()();
A.info.iName.top := 10;
A.info.iName.bottom := 10;
A.info.iAttributeStack.top := 0;
A.info.iAttributeStack.bottom := 0;
A.info.iMethodStack.top := 0; 7
A.info.iMethodStack.bottom := 0;
```



```
drawObject(A);
```

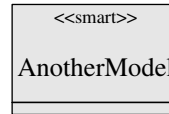
The same effect can be achieved by using the formatting information object `iClassNameOnly` or the `ClassName` macro:

```
EClass.A(iClassNameOnly("MyModel"))();
ClassName.B("AnotherModel");
Class_stereotypes.B("<<smart>>");
```



```
topToBottom(20)(A, B);
```

```
drawObjects(A, B);
```



To customize the space around the class name globally, you can set the values of `iClassNameOnly.iName.top` and `iClassNameOnly.iName.bottom`. Individually, for a given object, say `B`, the attributes `B.info.iName.top` and `B.info.iName.bottom` can be used.

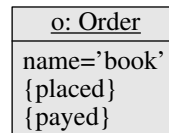
2.4 Objects (or Class Instances)

A UML object (or class instance) is created as follows:

```
Instance.name(object-name)
(list-of-attributes);
```

The suffix `name` gives a name to the `Instance` object. The name of the object (given by `object-name`) is typeset underlined. The attributes are given as a comma-separated list of strings, `list-of-attributes`.

```
Instance.order("o: Order")
("name='book'", "{placed}", "{payed}");
drawObject(order);
```



2.5 Parametrized Classes (Templates)

The most convenient way of typesetting a class template in MetaUML is to use the macro `ClassTemplate`. This macro creates a visual object which is appropriately positioned near the class object it adorns.

```
ClassTemplate.name(list-of-templates)
(class-object);
```

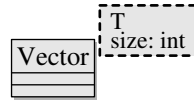
The `name` is the name of the template object, `list-of-templates` is a comma-separated list of strings and the `class-object` is the name of a class object.

Here is an example:


```

Class.A("Vector") ();
ClassTemplate.T("T", "size: int")(A);
drawObjects(A, T);

```



The macro `Template` can also be used to create a template object, but this time the resulting object can be positioned freely.

```

Template.name(list-of-templates);

```

Of course, it is possible to specify both stereotypes and template parameters for a given class.

2.6 Types of Links

In this section we enumerate the relations that can be drawn between classes by means of MetaUML macros. Suppose that we have the declared two points, A (on the left) and B (on the right):

```

pair A, B;
A = (0,0);
B = (50,0);

```

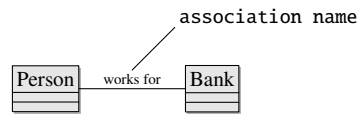
<code>link(association)(X.e -- Y.w)</code>	
<code>link(associationUni)(X.e -- Y.w)</code>	
<code>link(inheritance)(X.e -- Y.w)</code>	
<code>link(realization)(X.e -- Y.w)</code>	
<code>link(aggregation)(X.e -- Y.w)</code>	
<code>link(aggregationUni)(X.e -- Y.w)</code>	
<code>link(composition)(X.e -- Y.w)</code>	
<code>link(compositionUni)(X.e -- Y.w)</code>	
<code>link(dependency)(X.e -- Y.w)</code>	

2.7 Associations

In UML an association typically has two of association ends and may have a name specified for it. In turn, each association end may specify a multiplicity, a role, a visibility, an ordering. These entities are treated in MetaUML as pictures having specific drawing information (spacings, font).

The first method of creating association “items” is by giving them explicit names. Having a name for an association item comes in handy when referring to its properties is later needed (see the non UML-compliant diagram below). The last parameter of the macro `item` is an equation which may use the item’s name to perform positioning.

```
Class.P("Person")();
Class.C("Company")();
% drawing code omitted
```



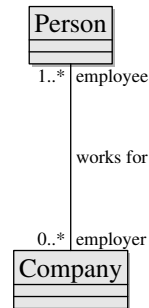
```
item.aName(iAssoc)("works for")
    (aName.s = .5[P.w, C.w]);
draw aName.n -- (aName.n + (20,20));
label.urt("association name" infont "tyxtt",
    aName.n + (20,20));
```

However, giving names to every association item may become an annoying burden (especially when there are many of them). Because of this, MetaUML also allows for “anonymous items”. In this case, the positioning is set by an equation which refers to the anonymous item as `obj`.

```
% P and C defined as in the previous example
```

```
item(iAssoc)("employee")(obj.nw = P.s);
item(iAssoc)("1..*")(obj.ne = P.s);
```

```
% other items are drawn similarly
```



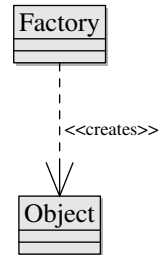
2.8 Dependencies and Stereotypes

Stereotypes are frequently used with dependencies. Below is an example.

```
Class.F("Factory")();
Class.O("Object")();
```

```
O.n = F.s - (0, 50);
drawObjects(F, 0);
```

```
clink(dependency)(F, O);
item(iStereo)("<<creates>>")(obj.w = .5[F.s,O.n])
```



3 Notes

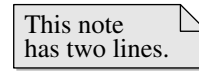
A note is created as follows:

```
Note.name(list-of-lines);
```

The suffix `name` is the name of the `Note` object. The contents of the note is given by a comma-separated list of strings, `list-of-lines`, gives the text contents of the note object, each string being drawn on its own line.

Here is an example:

```
Note.A("This note", "has two lines.");
drawObject(A);
```



3.1 Attaching notes to diagram elements

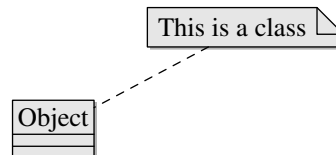
Notes can be attached to diagram elements by using a link of type `dashedLink`.

```
Note.A("This is a class");
Class.C("Object")();
```

```
A.sw = C.ne + (20, 20);
```

```
drawObject(A, C);
```

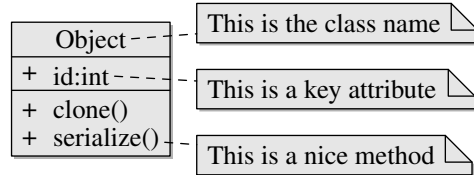
```
clink(dashedLink)(A, C);
```



Now let us see a more complex example, which demonstrates the ability of accessing sub-elements in a MetaUML diagram.

```
Note.nA("This is the class name");
Note.nB("This is a key attribute");
Note.nC("This is a nice method");

Class.C("Object")("+id:int")
      ("+clone()", "+serialize()");
```



```
topToBottom.left(10)(nA, nB, nC);
leftToRight(10)(C, nB);

drawObjects(C, nA, nB, nC);

click(dashedLink)(C.namePict, nA);
click(dashedLink)(C.attributeStack.pict[0], nB);
click(dashedLink)(C.methodStack.pict[1], nC);
```

Macros like `leftToRight` and `topToBottom` are presented in section 10.

3.2 Using mathematical formulae

MetaUML notes can contain mathematical formulae written in TeX [3]. Regrettably, LaTeX [4] support for formulae is **not** available. Limited as it may be, this feature is considered experimental, as it is not always straightforward to use. In the example below, note that the MetaPost package `TEX` is imported.

```
input metauml;
input TEX;
```

This class implements the formula:
 $\sum_1^n f(x) \cdot dx$

```
beginfig(1);
  Note.A("This class implements the formula:",
        TEX("$\sum_1^n f(x) \cdot dx$"));
  drawObjects(A);
endfig;

end
```

For taller formulae, you must be prepared to do some advanced stunts. Remark: `"aaa" & "bbb"` is MetaPost's way to concatenate the strings into `"aaabbb"`; the string containing the formula was split in two for layout reasons.

```
Note.A("Can you do it?",
      TEX("$\sum_1^n f(x) \cdot dx " &
        "\over \sum_1^m g(y) \cdot dy$"));
A.stack.info.spacing := 30;
A.stack.pict[1].info.ignoreNegativeBase := 0;

drawObject(A);
```

Can you do it?
 $\frac{\sum_1^n f(x) \cdot dx}{\sum_1^m g(y) \cdot dy}$

Alas, this trick does not entirely solve the problem: a third line in the note would be badly aligned. Therefore, until MetaUML's `Note` class is upgraded to better support this scenario, you may want to limit yourself to two lines per note — at least when tall formulae are involved.

4 Packages

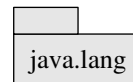
MetaUML allows for the creation of packages in various forms. Firstly, we have the option of writing the package name in the middle of the main box. Secondly, we can write the name on the tiny box above the main box, leaving the main box empty. Lastly, we can write the package name as in the second case, but the main box can have an arbitrary contents: classes, other packages, or even other UML items.

The macro that creates a package has the following synopsis:

```
Package.name(package-name)(subitems-list);
```

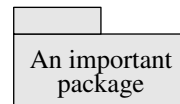
The parameter `package-name` is a string or a list of comma-separated strings representing the package's name. The `subitems-list` parameter is used to specify the subitems (typically classes or packages) of this package; its form is as a comma-separated list of objects, which can be void.

```
Package.P("java.lang")();  
drawObject(P);
```



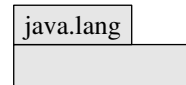
Below is another example:

```
Package.P("An important", "package")();  
drawObject(P);
```



If you wish to leave the main box empty, you can use the following code:

```
Package.P("java.lang")();  
P.info.forceEmptyContent := 1;  
drawObject(P);
```



The same effect as above can be achieved globally by doing:

```
iPackage.forceEmptyContent := 1;
```

More information on MetaUML's way of managing global and per-object configuration data can be found in [section 11](#) and [section 13](#).

Here is an example involving items contained in a package.

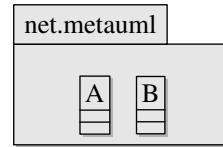
```

Class.A("A")();
Class.B("B")();
Package.P("net.metauml")(A, B);

leftToRight(10)(A, B);

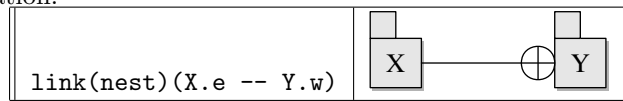
drawObject(P);

```



4.1 Types of Links

The nesting relation between packages is created by using the `nest` link information.



5 Component Diagrams

A component is created by the macro `Component`:

```

Component.name(component-name)
(subitems-list)

```

The parameter `component-name` is a string representing the component's name. The `subitems-list` parameter is used to specify the subitems of this component (possibly classes, packages or other components); its form is as a comma-separated list of objects, which can be void.

```

Component.C("Business Logic");
drawObject(C);

```



Here is an example involving subitems in a component:

```

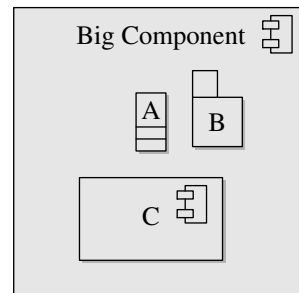
Class.A("A")();
Package.B("B");
Component.C("C");

Component.BigC("Big Component")(A, B, C);

leftToRight(10)(A, B);
topToBottom(10)(A, C);

drawObject(BigC);

```

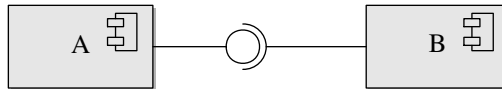


5.1 Types of Links

<code>link(requiredInterface)(A.e -- .5[A.e, B.w]);</code>	
<code>link(providedInterface)(.5[A.e, B.w] -- B.w);</code>	

The `requiredInterface` and `providedInterface` visual constructs can be easily combined, as shown in the following example:

```
Component.A("A")();
Component.B("B")();
```



```
leftToRight(80)(A, B);
```

```
drawObjects(A, B);
```

```
link(providedInterface)( A.e -- .5[A.e, B.w] );
link(requiredInterface)( B.w -- .5[A.e, B.w] );
```

6 Use Case Diagrams

6.1 Use Cases

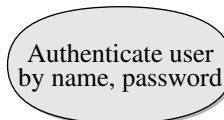
An use case is created by the macro `Usecase`:

```
Usecase.name(list-of-lines);
```

The `list-of-lines` is a comma-separated list of strings. These strings are placed on top of each other, centered and surrounded by the appropriate visual UML notation.

Here is an use case example:

```
Usecase.U("Authenticate user",
          "by name, password");
drawObject(U);
```



6.2 Actors

An actor is created by the macro `Actor`:

```
Actor.name(list-of-lines);
```

Here, `list-of-lines` represents the actor's name. For convenience, the name may be given as a list of strings which are placed on top of each other, to provide support for the situations when the role is quite long. Otherwise, giving a single string as an argument to the Actor constructor is perfectly fine.

Here is an actor example:

```
Actor.A("User");  
drawObject(A);
```



Sometimes it may be preferable to draw diagram relations positioned relatively to the visual representation of an actor (the “human”) rather than relatively to the whole actor object (which also includes the text). Because of that, MetaUML provides access to the “human” of every actor object `actor` by means of the sub-object `actor.human`.

```
Actor.A("Administrator");  
drawObject(A);  
draw objectBox(A);  
draw objectBox(A.human);
```



In MetaUML, `objectBox(X)` is equivalent to `X.nw -- X.ne -- X.se -- X.sw -- cycle` for every object `X`. `A.human` is considered a MetaUML object, so you can use expressions like `A.human.n` or `A.human.midx`.

6.3 Types of Links

Some of the types of links defined for class diagrams (such as inheritance, association etc.) can be used with similar semantics within use case diagrams.

7 Activity Diagrams

7.1 Begin, End and Flow End

The begin and the end of an activity diagram can be marked by using the macros `Begin` and `End` or `FlowFinal`, respectively. The constructors of these visual objects take no parameters:

```
Begin.beginName;  
End.endName;
```

Below is an example:


```
Begin.b;  
End.e;  
FlowFinal.f;
```



```
leftToRight(20)(b, e, f);
```

```
drawObjects(b, e, f);
```

7.2 Activity


An activity is constructed as follows:

```
Activity.name(list-of-strings);
```

The parameter `list-of-strings` is a comma-separated list of strings. These strings are centered on top of each other to allow for the accommodation of a longer activity description within a reasonable space.

An example is given below:

```
Activity.A("Learn MetaUML -",  
          "the MetaPost UML library");  
drawObject(A);
```



7.3 Fork and Join

A fork or join is created by the macro:

```
Fork.name(type, length);
```

The parameter `type` is a string and can be either of "h", "horiz", "horizontal" for horizontal bars, and either of "v", "vert", "vertical" for vertical bars. The `length` gives the bar's length.

```
Fork.forkA("h", 100);  
Fork.forkB("v", 20);  
  
leftToRight(10)(forkA, forkB);  
  
drawObject(forkA, forkB);
```



7.4 Branch

A branch is created by the macro:

```
Branch.name;
```

Here is an example:

```
Branch.testA;
```



```
drawObject(testA);
```

7.5 Types of Links

In activity diagrams, transitions between activities are needed. They are typeset as in the example below. In section 8.1 such a transition is showed. This type of link is also used for state machine diagrams.

```
link(transition)( pointA -- pointB );
```

8 State Diagrams

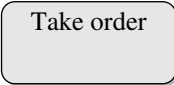
The constructor of a state allows for aggregated sub-states:

```
State.name(state-name)(substates-list);
```

The parameter `state-name` is a string or a list of comma-separated strings representing the state's name or description. The `substates-list` parameter is used to specify the substates of this state as a comma-separated list of objects; this list may be void.

An example of a simple state:

```
State.s("Take order")();  
drawObject(s);
```

A rectangular button with rounded corners and a light gray background, containing the text "Take order" in a dark gray font.

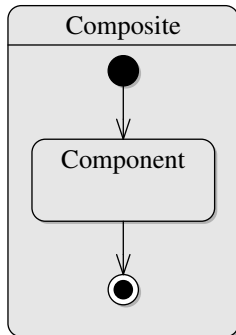
8.1 Composite States

A composite state is defined by enumerating at the end of its constructor the inner states. Interestingly enough, the composite state takes care of drawing the sub-states it contains. The transitions must be drawn after the composite state, as seen in the next example:

```
Begin.b;  
End.e;                               link(transition)(b.s -- c.n);  
State.c("Component")();             link(transition)(c.s -- e.n);  
State.composite("Composite")(b, e, c);
```

```
b.midx = e.midx = c.midx;  
c.top = b.bottom - 20;  
e.top = c.bottom - 20;
```

```
composite.info.drawNameLine := 1;  
drawObject(composite);
```



8.2 Internal Transitions

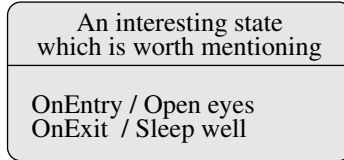
Internal transitions can be specified by using the macro:

```
stateTransitions.name(list-transitions);
```

Identifier `name` gives the state object whose internal transitions are being set, and parameter `list-transitions` is a comma-separated string list.

An example is given below:

```
State.s("An interesting state",
        "which is worth mentioning")();
stateTransitions.s(
    "OnEntry / Open eyes",
    "OnExit / Sleep well");
s.info.drawNameLine := 1;
```



```
drawObject(s);
```

8.3 Special States

Similarly to the usage of `Begin` and `End` macros, one can define history states, exit/entry point states and terminate pseudo-states, by using the following constructors.

```
History.nameA;
ExitPoint.nameB;
EntryPoint.nameC;
Terminate.nameD;
```

9 Drawing Paths

The `link` macro is powerful enough to draw relations following arbitrary paths:

```

path cool;
cool := A.e .. A.e+(20,10) ..
       B.s+(20,-40) .. B.s+(-10,-30)
       -- B.s;
link(inheritance)(cool);

link(aggregationUni)
(A.n ..(30,30)..B.w);

```



Amusing as it may be, this feature gets old soon. When typesetting UML diagrams in good style, rectangular paths are usually preferred. It is for this kind of paths that MetaUML offers extensive support, by means of “syntactic sugar” constructs which are not only self-documenting, but reduce the amount of typing and thinking required.

9.1 Manhattan Paths

The “Manhattan” path macros generate a path between two points consisting of one horizontal and one vertical segment. The macro `pathManhattanX` generates first a horizontal segment, while the macro `pathManhattanY` generates first a vertical segment. In MetaUML it also matters the direction of a path, so you can choose to reverse it by using `rpathManhattanX` and `rpathManhattanY` (note the prefix “r”):

```

pathManhattanX(A, B)
pathManhattanY(A, B)

rpathManhattanX(A, B)
rpathManhattanY(A, B)

```

Here is an example:

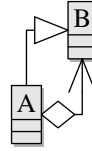
```

Class.A("A")();
Class.B("B")();

B.sw = A.ne + (10,10);
drawObjects(A, B);

link(aggregationUni)
  (rpathManhattanX(A.e, B.s));
link(inheritance)
  (pathManhattanY(A.n, B.w));

```



9.2 Stair Step Paths

These path macros generate stair-like paths between two points. The “stair” can “rise” first in the direction of Ox axis (`pathStepX`) or in the direction of Oy axis (`pathStepY`). How much should a step rise is given by an additional parameter, `delta`. Again, the macros prefixed with “r” reverse the direction of the path given by their unprefix counterparts.

```

pathStepX(A, B, delta)
pathStepY(A, B, delta)

rpathStepX(A, B, delta)
rpathStepY(A, B, delta)

```

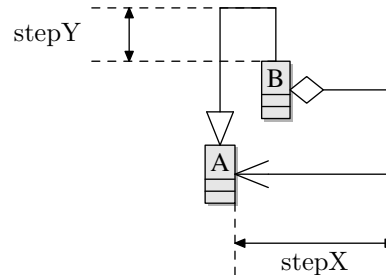
Here is an example:

```

stepX:=60;
link(aggregationUni)
  (pathStepX(A.e, B.e, stepX));

stepY:=20;
link(inheritance)
  (pathStepY(B.n, A.n, stepY));

```



9.3 Horizontal and Vertical Paths

There are times when drawing horizontal or vertical links is required, even when the objects are not properly aligned. To this aim, the following macros are useful:

```

pathHorizontal(pA, untilX)
pathVertical(pA, untilY)

```

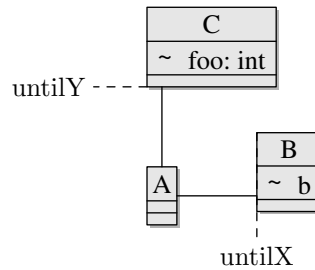
```
rpathHorizontal(pA, untilX)
rpathVertical(pA, untilY)
```

A path created by `pathHorizontal` starts from the point `pA` and continues horizontally until coordinate `untilX` is reached. The macro `pathVertical` constructs the path dually, working vertically. The prefix “r” reverses the direction of the path.

Usage example:

```
untilX := B.left;
link(association)
(pathHorizontal(A.e, untilX));

untilY:= C.bottom;
link(association)
(pathVertical(A.n, untilY));
```



9.4 Direct Paths

A direct path can be created with `directPath`. The call `directPath(A, B)` is equivalent to `A -- B`.

9.5 Paths between Objects

Using the constructs presented above, links between diagram objects are drawn easily like this:

```
link(transition)(directPath(objA.nw, objB.se));
```

There are times however when this direct approach may yield unsatisfactory visual results, especially when the object’s corners is round. To tackle these situations, MetaUML provides the macro `pathCut`, whose aim is to limit a given path exactly to the region outside the actual borders of the objects it connects. The macro’s synopsis is:

```
pathCut(thePath)(objectA, objectB)
```

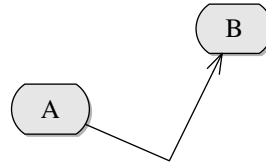
Here, `thePath` is a given MetaPost path and `objectA` and `objectB` are two MetaUML objects. By contract, each MetaUML object of type, say, `X` defines a macro `X.border` which returns the path that surrounds it. Because of that, `pathCut` can make the appropriate modifications to `thePath`.

The following code demonstrates the benefits of the `pathCut` macro:

```

z = A.se + (30, -10);
link(transition)
  (pathCut(A, B)(A.c--z--B.c));

```



9.5.1 Direct Paths between Centers

At times is quicker to just draw direct paths between the center of two objects, minding of course the object margins. The macro which does this is `clink`:

```

clink(how-to-draw-information)(objA, objB);

```

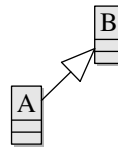
The parameter `how-to-draw-information` is the same as for the macro `link`; `objA` and `objB` are two MetaUML objects.

Below is an example which involves the inheritance relation:

```

clink(inheritance)(A, B);

```



10 Arranging Diagram Items

Using equations involving cardinal points, such as $A.nw = B.ne + (10,0)$, is good enough for achieving the desired results. However, programs are best to be written for human audience, rather than for compilers. It does become a bit tiresome to think all the time of cardinal points and figure out the direction of positive or negative offsets. Because of that, MetaUML offers syntactic sugar which allows for an easier understanding of the intent behind the positioning code.

Suppose that we have three classes, `A`, `B`, `C` and their base class `Base`. We want the base class to be at the top, and the derived classes to be on a line below. This code will do:

```

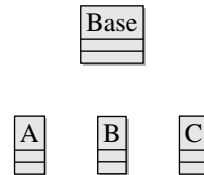
A.ne = B.nw + (20,0);
B.ne = C.nw + (20,0);
Base.s = B.n + (0,-20);

```

Unfortunately, writing code such as this makes it hard for fellow programmers to visualize its intent upon reading it. And “fellow programmers“ include the author, five minutes later.

Perhaps the next version of the code will drive home the point. The outcome is the same as before, but the layout is stated in a more human-friendly way. You might even infer by yourself that the numeric argument represents the distance between the objects.

```
leftToRight(20)(A, B, C);
topToBottom(20)(Base, B);
```



Below there are examples which show how these macros can be used. Suppose that we have the following definitions for objects X, Y, and Z; also, let's assume that `spacing` is a numeric variable set to 5.

```
Picture.X("a");
Picture.Y("...");
Picture.Z("Cyan");
```

<code>leftToRight.top(spacing)(X, Y, Z);</code>	
<code>leftToRight.midy(spacing)(X, Y, Z);</code>	
<code>leftToRight.bottom(spacing)(X, Y, Z);</code>	
<code>topToBottom.left(spacing)(X, Y, Z);</code>	
<code>topToBottom.midx(spacing)(X, Y, Z);</code>	
<code>topToBottom.right(spacing)(X, Y, Z);</code>	

To make things even easier, the following equivalent constructs are also allowed:

```
leftToRight.midy(spacing)(X, Y, Z);
leftToRight(spacing)(X, Y, Z);
```

```
topToBottom.midx(spacing)(X, Y, Z);
topToBottom(spacing)(X, Y, Z);
```

If you want to specify that some objects have a given property equal, while the distance between them is given elsewhere, you can use the macro `same`. This macro accepts a variable number of parameters, but at least two. The following table gives the interpretation of the macro for a simple example.

<code>same.top(X, Y, Z);</code>	<code>X.top = Y.top = Z.top;</code>
<code>same.midy(X, Y, Z);</code>	<code>X.midy = Y.midy = Z.midy;</code>
<code>same.bottom(X, Y, Z);</code>	<code>X.bottom = Y.bottom = Z.bottom;</code>
<code>same.left(X, Y, Z);</code>	<code>X.left = Y.left = Z.left;</code>
<code>same.midx(X, Y, Z);</code>	<code>X.midx = Y.midx = Z.midx;</code>
<code>same.right(X, Y, Z);</code>	<code>X.right = Y.right = Z.right;</code>

Relative positions of two points can be declared more easily using the macros `below`, `above`, `atright`, `atleft`. Let us assume that A and B are two points (objects of type `pair` in MetaPost). The following constructs are equivalent:

<code>B = A + (5,0);</code>	<code>B = atright(A, 5);</code>
<code>B = A - (5,0);</code>	<code>B = atleft(A, 5);</code>
<code>B = A + (0,5);</code>	<code>B = above(A, 5);</code>
<code>B = A - (0,5);</code>	<code>B = below(A, 5);</code>

11 The MetaUML Infrastructure

MetaPost is a macro language based on equation solving. Using it may seem quite tricky at first for a programmer accustomed to modern object-oriented languages. However, the great power of MetaPost consists in its versatility. Indeed, it is possible to write a system which mimics quite well object-oriented behavior. Along this line, METAOBJ [5] is a library worth mentioning: it provides a high-level objects infrastructure along with a battery of predefined objects.

Surprisingly enough, MetaUML does not use METAOBJ. Instead, it uses a custom written, lightweight object-oriented infrastructure, provisionally called “`util`”. METAOBJ’s facilities, although impressive, were perceived by me as being a bit too much for what was initially intended as a quick way of getting some UML diagrams layed out. Inspired by METAOBJ, “`util`” was designed to fulfill with minimal effort the specific tasks needed to comfortably position, align or group visual objects which include text.

Another library having some object-oriented traits is the `boxes` library, which comes with the standard MetaPost distribution. Early versions of MetaUML did use `boxes` as an infrastructure, but this approach had to be abandoned eventually. The main reason was that it was difficult to achieve good visual results when stacking texts (more on that further on). For all it’s worth, it did not fit well with the way in which MetaUML’s layout mechanism was shaping up at the time.

11.1 Motivation

Suppose that we want to typeset two texts with their bottom lines aligned, using `boxit`:

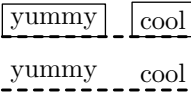
```

boxit.a ("yummy");
boxit.b ("cool");

a.nw = (0,0); b.sw = a.se + (10,0);

drawboxed (a, b); % or drawunboxed(a,b)
draw a.sw -- b.se dashed evenly
  withpen pencircle scaled 1.1;

```



Note that, despite supposedly having their bottom lines aligned, “yummy” *looks* slightly higher than “cool”. This would be unacceptable in a UML class diagram, when roles are placed at the ends of a horizontal association. Regardless of the default spacing being smaller in the `util` library, the very same unfortunate misalignment effect rears its ugly head:

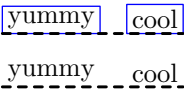
```

Picture.a("yummy");
Picture.b("cool");
% comment next line for unboxed
a.info.boxed := b.info.boxed := 1;

b.sw = a.se + (10,0);

drawObjects(a, b);

```



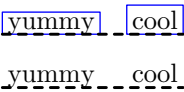
However, the strong point of `util` is that we have a recourse to this problem:

```

iPict.ignoreNegativeBase := 1;

Picture.a("yummy");
Picture.b("cool");
% the rest the same as above
drawObjects(a, b);

```



11.2 The Picture Macro

We have seen previously the line `iPict.ignoreNegativeBase := 1`. Who is `iPict` and what is it doing in our program? MetaUML aims at separating the “business logic” (what to draw) from the “interface” (how to draw). In order to achieve this, it records the “how to draw” information within the so-called `Info` structures. The object `iPict` is an instance of `PictureInfo` structure, which has the following properties (or attributes):

```

left, right, top, bottom
ignoreNegativeBase
boxed, borderColor

```

The first four attributes specify how much space should be left around the actual item to be drawn. The marvelous effect of `ignoreNegativeBase` has just been shown (off), while the last two attributes control whether the border should be drawn (when `boxed=1`) and if drawn, in which color.

There's one more thing: the font to typeset the text in. This is specified in a `FontInfo` structure which has two attributes: the font name and the font scale. This information is kept within the `PictureInfo` structure as a contained attribute `iFont`. Both `FontInfo` and `PictureInfo` have "copy constructors" which can be used to make copies. We have already the effect of these copy constructors at work, when we used:

```
Picture.a("yummy");
a.info.boxed := 1;
```

A copy of the default info for a picture, `iPict`, has been made within the object `a` and can be accessed as `a.info`. Having a copy of the info in each object may seem like an overkill, but it allows for a fine grained control of the drawing mode of each individual object. This feature comes in very handy when working with a large number of settings, as it is the case for MetaUML.

Let us imagine for a moment that we have two types of text to write: one with a small font and a small margin and one with a big font and a big margin. We could in theory configure each individual object or set back and forth global parameters, but this is far from convenient. It is preferable to have two sets of settings and specify them explicitly when they are needed. The following code could be placed somewhere in a configuration file and loaded before any `beginfig` macro:

```
PictureInfoCopy.iBig(iPict);
iBig.left := iBig.right := 20;
iBig.top := 10;
iBig.bottom := 1;
iBig.boxed := 1;
iBig.ignoreNegativeBase := 1;
iBig.iFont.name := defaultfont;
iBig.iFont.scale := 3;

PictureInfoCopy.iSmall(iPict);
iSmall.boxed := 1;
iSmall.borderColor := green;
```

Below is an usage example of these definitions. Note the name of the macro: `EPicture`. The prefix comes from "explicit" and it's used to acknowledge that the "how to draw" information is given explicitly — as a parameter, rather than defaulted to what's recorded in `iPict`, as with the `Picture` macro. Having predefined configurations yields short, convenient code.

```

EPicture.a(iBig)("yummy");
EPicture.b(iSmall)("cool");
% you can still modify a.info, b.info

b.sw = a.se + (10,0);

drawObjects(a, b);

```



11.2.1 Fixed Sizes

By default, the size of a `Picture` object is set by its contents. However, it is possible to specify fixed dimensions both the width and the height, independently. This can be done by setting the `info`'s attributes `fixedWidth` and `fixedHeight` to values greater than 0. If any of these attributes is left to its default value, `-1`, then for the corresponding axis the dimension is set according to the dimension of the content. Nevertheless, the fixed dimensions are enforced, even though the contained object would have needed additional space.

```

PictureInfoCopy.myFixed(iPict);
myFixed.ignoreNegativeBase := 1;
myFixed.fixedWidth := 15;
myFixed.fixedHeight := 20;
myFixed.boxed := 1;

```



```

EPicture.a(myFixed)("a");
EPicture.b(myFixed)(".-");
EPicture.c(myFixed)("toolong");

```

```

leftToRight.bottom(10)(a, b, c);

```

```

drawObjects(a, b, c);

```

11.2.2 Content alignment

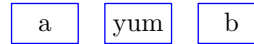
When fixed dimensions are used, one most likely would prefer a centered alignment of the contents in the `Picture` box. This option can be expressed independently for each of the axes, by setting the `info`'s attributes `valign` and `halign` to descriptive string values. For horizontal alignment, `halign` can be set to `"left"` or `"center"`, and for vertical alignment, `valign` can be set to `"bottom"` or `"center"`. The default values for these attributes are `"left"` and `"bottom"`, respectively.

The next example uses horizontal centered alignment and a bottom alignment with a 4.5 base offset, for vertical alignment. This vertical alignment gives a better visual result than the centered one, at least for the situations in which there are texts to be placed horizontally.

```

PictureInfoCopy.myFixed(iPict);
myFixed.ignoreNegativeBase := 1;
myFixed.bottom := 4.5;
myFixed.valign := "bottom";
myFixed.halign := "center";
myFixed.fixedWidth := 25;
myFixed.fixedHeight := 15;
myFixed.boxed := 1;

```



```

EPicture.a(myFixed)("a");
EPicture.b(myFixed)("yum");
EPicture.c(myFixed)("b");

```

```
leftToRight.bottom(10)(a, b, c);
```

```
drawObjects(a, b, c);
```

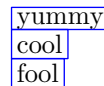
11.3 Stacking Objects

It is possible to stack objects, much in the style of `setboxjoin` from `boxes` library.

```

Picture.a0("yummy");
Picture.a1("cool");
Picture.a2("fool");

```



```

setObjectJoin(pa.sw = pb.nw);
joinObjects(scantokens listArray(a)(3));

```

```

drawObjects(scantokens listArray(a)(3));
% or drawObjects (a0, a1, a2);

```

The `listArray` macro provides here a shortcut for writing `a0`, `a1`, `a2`. This macro is particularly useful for generic code which does not know beforehand the number of elements to be drawn. Having to write the `scantokens` keyword is admittedly a nuisance, but this is required.

11.4 The Group Macro

It is possible to group objects in MetaUML. This feature is the cornerstone of MetaUML, allowing for the easy development of complex objects, such as composite states in state machine diagrams.

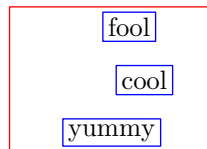
Similarly to the macro `Picture`, the structure `GroupInfo` is used for specifying group properties; its default instantiation is `iGroup`. Furthermore, the macro `EGroup` explicitly sets the layout information.

Here is an example:

```

iGroup.left:=20;
iGroup.right:=15;
iGroup.boxed:=1;
iPicture.boxed:=1;

```



```

Picture.a("yummy");
Picture.b("cool");
Picture.c("fool");

```

```

b.nw = a.nw + (20,20); % A
c.nw = a.nw + (15, 40); % B

```

```

Group.g(a, b, c);
g.nw = (10,10); % C

```

```

drawObject(g);

```

After some objects are grouped, they can only be drawn by invoking the `drawObject` macro on the group that aggregates them, and not individually. Conveniently, once the relative positioning of objects within a group is set (line A and B), the whole group can be “moved” to the desired position (line C), and all the contained objects will move along.

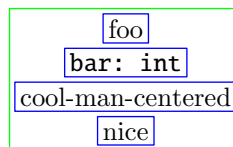
11.5 The PictureStack Macro

The `PictureStack` macro is a syntactic sugar for a set of pictures, stacked according to predefined equations and grouped together.

```

iStack.boxed := 1;
iStack.iPict.boxed := 1;
PictureStack.myStack("foo",
  "bar: int" infont "tyxtt",
  "nicely-centered" infont defaultfont,
  "nice")("vcenter");

```



```

drawObject(myStack);

```

Note the last parameter of the macro `PictureStack`, here `vcenter`. It is used to generate appropriate equations based on a descriptive name. The spacing between individual picture objects is set by the field `iStack.spacing`. Currently, the following alignment names are defined: `vleft`, `vright`, `vcenter`, `vleftbase`, `vrightbase`, `vcenterbase`. All these names refer to vertical alignment (the prefix “v”); alignment can be at left, right or centered. The variants having the suffix “base” align the pictures so that `iStack.spacing` refer to the distance between the bottom lines of the pictures. The unsuffixed variants use

`iStack.spacing` as the distance between one's bottom line and the next's top line.

The “**base**” alignment is particularly useful for stacking text, since it offers better visual appearance when `iPict.ignoreNegativeBase` is set to 1.

12 Components Design

Each MetaUML component (e.g. `Picture`, `PictureStack`, `Class`) is designed according to an established pattern. This section gives more insight on this.

In order to draw a component, MetaUML categorizes the required information as follows:

- what to draw, or what are the elements of a component.
- how to draw, or how are the elements positioned in relation to each other within the component
- where to draw

For example, in order to draw a picture object we must know, respectively:

- what is the text or the native picture that needs to be drawn
- what are the margins that should be left around the contents
- where is the picture to be drawn

Why do we bother with these questions? Why don't we just simply draw the picture component as soon as it was created and get it over with? That is, why doesn't the following code just work?

```
Picture.pict("foo");
```

Well, although we have the answer to question 1 (what to draw), we still need to have question 3 answered. The code below becomes thus a necessity (actually, you are not forced to specify the positioning of an object, because its draw method positions it to (0,0) by default):

```
% question 1: what to draw
Picture.pict("foo");
```

```
% question 3: where to draw
pict.nw = (10,10);
```

```
% now we can draw
drawObject(pict);
```

How about question 2, how to draw? By default, this problem is addressed behind the scenes by the component. This means, for the `Picture` object, that a native picture is created from the given string, and around that picture certain margins are placed, by means of MetaPost equations. (The margins also come in handy when stacking `Picture` objects, so that the result doesn't look too cluttered.) If these equations were defined within the `Picture` constructor, then an usability problem would have appeared, because it wouldn't have been possible to modify the margins, as in the code below:

```
% question 1: what to draw
Picture.pict("foo");

% question 2: how to draw
pict.info.left := 10;
pict.info.boxed := 1;

% question 3: where to draw
pict.nw = (0,0);

% now we can draw
drawObject(pict);
```

To allow for this type of code, the equations that define the layout of the `Picture` object (here, what the margins are) must be defined somewhere after the constructor. This is done by a macro called `Picture_layout`. This macro defines all the equations which link the “what to draw” information to the “how to draw” information (which in our case is taken from the `info` member, a copy of `iPict`). Nevertheless, notice that `Picture_layouts` is not explicitly invoked. To the user's great relief, this is taken care of automatically within the `Picture_draw` macro.

There are times however, when explicitly invoking a macro like `Picture_layout` becomes a necessity. This is because, by contract, it is only after the `layout` macro is invoked that the final dimensions (width, height) of an object are definitely and permanently known. Imagine that we have a component whose job is to surround in a red-filled rectangle some other objects. This component needs to know what the dimensions of the contained objects are, in order to be able to set its own dimensions. At drawing time, the contained objects must not have been drawn already, because the red rectangle of the container would overwrite them. Therefore, the whole pseudo-code would be:

```
Create objects o1, o2, ... ok;
Create container c(o1, o2, ..., ok);
Optional: modify info-s for o1, o2, ... ok;
Optional: modify info for c;

layout c, requiring layout of o1, o2, ... ok;
establish where to draw c;
```



```
draw red rectangle defined by c;
draw components o1, o2, ...ok within c
```

A natural conclusion is that an object must not be laid out more than once, because otherwise inconsistent or superfluous equations would arise. To enforce this, by contract, any object must keep record of whether its layout method has already been invoked, and if the answer is affirmative, subsequent invocations of the layout macro would do nothing. It is very important to mention that after the `layout` macro is invoked over an object, modifying the `info` member of that object has no subsequent effect, since the layout equations are declared and interpreted only once.

12.1 Notes on the Implementation of Links

MetaUML considers edges in diagram graphs as links. A link is composed of a path and the heads (possible none, one or two). For example, since an association has no heads, it suffices to draw along the path with a solid pen; however, an unidirectional aggregation has, in addition to a solid path, two heads: one is an arrow and the other is a diamond.

The general algorithm for drawing a link is:

0. Reserve space for heads
1. Draw the path (except for the heads)
2. Draw head 1
3. Draw head 2

Each of the UML link types define how the drawing should be done, in each of the cases (1, 2 and 3). Consider the link type of unidirectional composition. Its “class” is declared as:

```
vardef CompositionUniInfo@# =
  LinkInfo@#;

  @#widthA      = defaultRelationHeadWidth;
  @#heightA     = defaultRelationHeadHeight;
  @#drawMethodA = "drawArrow";

  @#widthB      = defaultRelationHeadWidth;
  @#heightB     = defaultRelationHeadHeight;
  @#drawMethodB = "drawDiamondBlack";

  @#drawMethod = "drawLine";
enddef;
```

Using this definition, the actual description is created like this:

```
CompositionUniInfo.compositionUni;
```

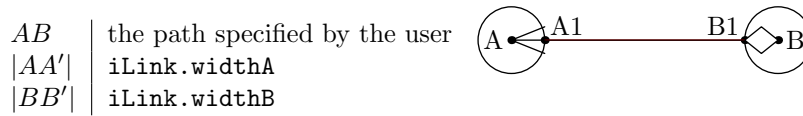


Figure 2: Details on how a link is drawn by MetaUML.

As shown previously, is is the macro `link` which performs the actual drawing, using the link description information which is given as parameter (generally called `iLink`). For example, we can use:

```
link(aggregationUni)((0,0)--(40,0));
```

Let us see now the inner workings of macro `link`. Its definition is:

```
vardef link(text iLink)(expr myPath)=
  LinkStructure.ls(myPath,
                  iLink.widthA, iLink.widthB);
  drawLinkStructure(ls)(iLink);
enddef;
```

First, space is reserved for heads, by “shortening” the given path `myPath` by `iLink.widthA` at the beginning and by `iLink.widthB` at the end. After that, the shortened path is drawn with the “method” given by `iLink.drawMethod` and the heads with the “methods” `iLink.drawMethodA` and `iLink.drawMethodB`, respectively (figure 2).

12.2 Object Definitions: Easier `generic_declare`

In MetaPost, if somebody wants to define something resembling a class in an object-oriented language, named, say, `Person`, he would do something like this:

```
vardef Person@#(expr _name, _age)=
  % @# prefix can be seen as ‘this’ pointer
  string @#name;
  numeric @#age;

  @#name := _name;
  @#age := _age;
enddef;
```

This allows for the creation of instances (or objects) of class `Person` by using declarations like:

```
Person.personA;
Person.personB;
```

However, if one also wants to be able to create indexed arrays of persons, such as `Person.student0`, `Person.student1` etc., the definition of class `Person` must read:

```
vardef Person@#(expr _name, _age)=
  _n_ := str @#;
  generic_declare(string) _n.name;
  generic_declare(numeric) _n.age;

  @#name := _name;
  @#age := _age;
enddef;
```

This construction is rather inelegant. MetaUML offers alternative macros to achieve the same effect, uncluttering the code by removing the need for the unaesthetic `_n_` and `_n`.

```
vardef Person@#(expr _name, _age)=
  attributes(@#);
  var(string) name;
  var(numeric) age;

  @#name := _name;
  @#age := _age;
enddef;
```

13 Customization in MetaUML: Examples

We have seen that in MetaUML the “how to draw” information is memorized into the so-called “Info” structures. For example, the default way in which a `Picture` object is to be drawn is recorded into an instance of `PictureInfo`, named `iPict`. In this section we present a case study involving the customization of `Class` objects. The customization of any other MetaUML objects works similarly. Here we cannot possibly present all the customization options for all kinds of MetaUML objects: this would take too long. Nevertheless, an interested reader can refer to the top of the appropriate MetaUML library file, where `Info` structures are defined. For example, class diagram related definitions are in `metauml_class.mp`, activity diagram definitions are in `metauml_activity.mp` etc.

13.1 Global settings

Let us assume that we do not particularly like the default foreground color of all classes, and wish to change it to something yellowish. In this scenario, one would most likely want to change the appropriate field in `iClass`:

```
iClass.foreColor := (.9, .9, 0);
```

After this, we can obtain the following result:

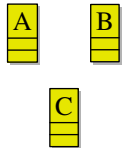
```

Class.A("A")();
Class.B("B")();
Class.C("C")();

B.w = A.e + (20,0);
C.n = .5[A.se, B.sw] + (0, -10);

drawObjects(A, B, C);

```



13.2 Individual settings

To modify the settings of one particular `Class` objects, another strategy is more appropriate. How about having class `C` stand out with a light blue foreground color, a bigger font size for the class name and a blue border?

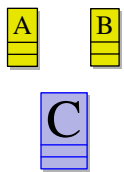
```

iPict.foreColor := (.9, .9, 0);

Class.A("A")();
Class.B("B")();
Class.C("C")();
C.info.foreColor := (.9, .7, .7);
C.info.borderColor := green;
C.info.iName.iFont.scale := 2;

% positioning code omitted
drawObjects(A, B, C);

```



As an aside, each `Class` object has an `info` member which is created as a copy of `iClass`; the actual drawing is performed using this copied information. Because of that, the `info` member can be safely modified after the object has been created, obtaining the expected results and not influencing other objects.

Another thing worth mentioning is that the `ClassInfo` structure contains the `iName` member, which is an instance of `PictureInfo`. In our example we do not want to modify the spacings around the `Picture` object, but the characteristics of the font its contents is typeset into. To do that, we modify the `iName.iFont` member, which by default is a copy of `iFont` (an instance of `FontInfo`, defined in `util_picture.mp`). If, for example, we want to change the font the class name is rendered into, we would set the attribute `iName.iFont.name` to a string representing a font name on our system (as used with the MetaPost `infont` operator).

13.3 Predefined settings

This usage scenario is perhaps more interesting. Suppose that we have two types of classes which we want to draw differently. Making the setting adjustments

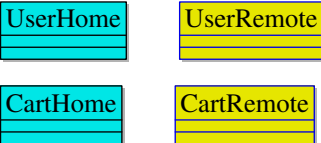
for each individual class object would soon become a nuisance. MetaUML's solution consists in the ability of using predefined "how to draw" Info objects. Let us create such objects:

```
ClassInfoCopy.iHome(iClass);
iHome.foreColor := (0, .9, .9);

ClassInfo.iRemote;
iRemote.foreColor := (.9, .9, 0);
iRemote.borderColor := green;
```

Object `iHome` is a copy of `iClass` (as it might have been set at the time of the macro call). Object `iRemote` is created just as `iClass` is originally created. We can now use these Info objects to easily set the "how to draw" information for classes. The result is depicted below, please note the "E" prefix in EClass:

```
EClass.A(iHome)("UserHome")();
EClass.B(iRemote)("UserRemote")();
EClass.C(iHome)("CartHome")();
EClass.D(iRemote)("CartRemote")();
```



13.4 Extreme customization

When another font (or font size) is used, it may become necessary to change the space between the baselines of attributes and methods. Figure below is the result of the (unlikely) code:

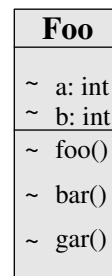
```
Class.A("Foo")
  ("a: int", "b: int")
  ("foo()", "bar()", "gar()");

A.info.iName.iFont.name := metauml_defaultFontBold;
A.info.iName.iFont.scale := 1.2;

A.info.iAttributeStack.iPict.iFont.scale := 0.8;
A.info.iAttributeStack.top := 10;
A.info.iAttributeStack.spacing := 11;

A.info.iMethodStack.iPict.iFont.scale := 2;
A.info.iMethodStack.spacing := 17;
A.info.iMethodStack.bottom := 10;

drawObject(A);
```



Both `iAttributeStack` and `iMethodStack` are instances of `PictureStackInfo`, which is used to control the display of `PictureStack` objects.

As font names, you can choose from the globally defined `metauml_defaultFont`, `metauml_defaultFontOblique`, `metauml_defaultFontBold`, `metauml_defaultFontBoldOblique`, or any other name of a font that is available on your system.

14 Alternatives to MetaUML

No software package is perfect, and for this MetaUML is a prime example. Here is a list of packages that may also be used to create UML diagrams for LaTeX work:

- `uml.sty` [6]
- `pst-uml` [7]
- `umldoc` [8]
- `TiKZ-UML` [9]

Do not ignore the possibility of creating your diagrams using a GUI program, and then exporting them into a LaTeX-friendly open format such as SVG [10].

15 Test Suite

15.1 Low-level

Test 1 —
nothing-shown-(intentionally)

Test 2 —
nothing-shown-(intentionally)

15.2 Fonts

Test 1 —

```
Font name: ( ) pcurr  
<<stereotype>> text with guillemets. ><>><<  
<<a>>, <<b>>, <<c>>  
[g uard] text with square brackets [].  
{c onstraint} text with curly brackets {}.
```

Test 2 —

```
Font name: ( ) tyxbtt  
<<stereotype>> text with guillemets. ><>><<  
<<a>>, «b», «c»  
[g uard] text with square brackets [].  
{c onstraint} text with curly brackets {}.
```

Test 3 —

```
assembleElementLocalMatrix(k: KeyType, mat: LocalMatrixType, a: AssembleAction)  
assembleElementLocalMatri(k: KeyType, mat: LocalMatrixType, a: AssembleAction)  
assembleElntLocalMatri(k: KeyType, mat: LocalMatrixType, a: AssembleAction)
```

15.3 Util library

15.3.1 Picture tests

Test 1 —
qux,-norf

```
xxx .yyy  
foo,-bar,-baz
```

Custom iPicture

Test 2 —

```

toof

root
bar
foo
Test 3 —
goof
-----
----- f: int goofy: int goot
a

|      |
|------|
| foo1 |
| bar1 |

foo

|       |
|-------|
| baz2  |
| norf2 |

bar
baz
norf
Test 4 —
goof Aoorian fpp f: int aa()
foo Bar baz qux f: int aa()
Test 5 —
<<foo>>
Test 6 —
x:int
      an-anonymous-item
foo-bar-baz
Test 7 —
a 

|     |
|-----|
| bar |
|-----|



|     |
|-----|
| .-. |
|-----|



|     |
|-----|
| baz |
|-----|



|     |
|-----|
| qux |
|-----|

 norf
Test 8 —
a 

|     |
|-----|
| bar |
|-----|



|     |
|-----|
| .-. |
|-----|



|     |
|-----|
| baz |
|-----|



|     |
|-----|
| qux |
|-----|

 norf
Test 9 —


|   |
|---|
| a |
|---|



|     |
|-----|
| bar |
|-----|



|     |
|-----|
| .-. |
|-----|



|     |
|-----|
| baz |
|-----|



|     |
|-----|
| qux |
|-----|

 norf
Test 10 —


|   |
|---|
| a |
|---|



|     |
|-----|
| bar |
|-----|



|     |
|-----|
| .-. |
|-----|



|     |
|-----|
| baz |
|-----|



|     |
|-----|
| qux |
|-----|

 norf

```

15.3.2 Picture tests - TeX rendering

```

Test 1 —


|                      |
|----------------------|
| Hello, world $x = 7$ |
|----------------------|



|                                                |
|------------------------------------------------|
| Hello, world!                                  |
| This is cool: $x = y$ .                        |
| But this is insane: $\sum_1^3 \frac{f(x)}{x}!$ |


```


15.3.3 Group tests

Test 1 —

p0
p1

Test 2 —

s s

□

Test-picture-in-group

15.3.4 PictureStack tests

Test 1 —

Test 2 —

foo

Test 3 —

foo

bar

Test 4 —

item-A
item-B-long
C

item-A
item-B-long
C

item-A
item-B-long
C

Test 5 —

BCA:ACB

Test 6 —

• go

.

.. further

... and-further

.... and-further-still

Test 7 —

a

b

c

d

e

15.3.5 Positioning tests

Test 1 —

a	...	XYZ
---	-----	-----

a	...	XYZ
---	-----	-----

Test 2 —

a	...	XYZ
---	-----	-----

a	...	XYZ
---	-----	-----

Test 3 —

a	...	XYZ
---	-----	-----

a	...	XYZ
---	-----	-----

Test 4 —

a
...
XYZ

a
...
XYZ

Test 5 —

a
...
XYZ

a
...
XYZ

Test 6 —

a
...
XYZ

a
...
XYZ

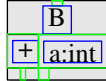
15.4 Class diagram

15.4.1 Class tests

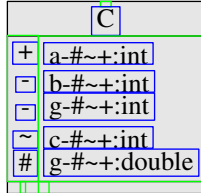
Test 1 —



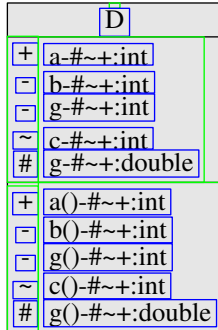
Test 2 —



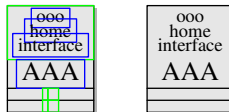
Test 3 —



Test 4 —



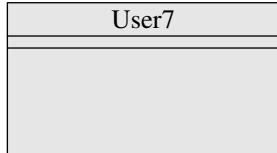
Test 5 —



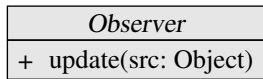
Test 6 —



Test 7 —



Test 8 —



Test 9 —

<i>Observer</i>
+ update(src: Object)

Test 10 —

<i>Observer</i>
+ update(src: Object)

Test 11 —

<i>AbstractClass</i>
~ []{}
+ update(src: Object)

Test 12 —

<i>AbstractClass</i>
~ []{}
+ update(src: Object)

Test 13 —

AClassWithNoCompartments

Test 14 —

AnotherClass

Test 15 —

<<interface>> <<remote>>
AnotherClass

Test 16 —

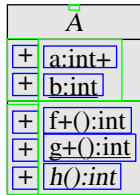
Foo
a: int b: int c: int d: int
x() y() z() t()

15.4.2 Class feature types tests

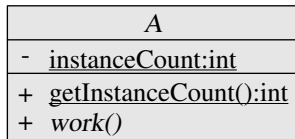
Test 1 —

Test 2 —

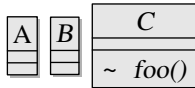
Test 3 —



Test 4 —

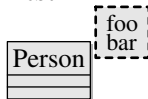


Test 5 —

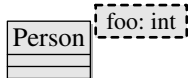


15.4.3 Class template tests

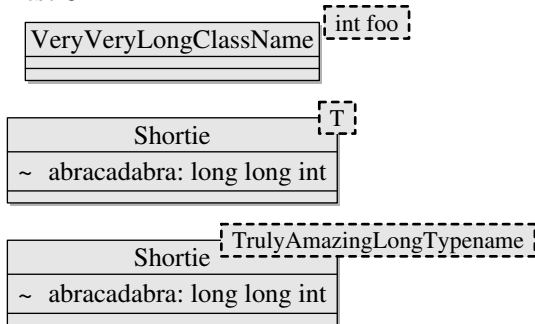
Test 1 —



Test 2 —

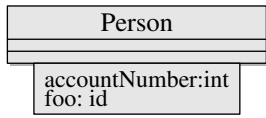


Test 3 —

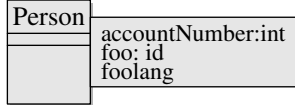


15.4.4 Qualified Association tests

Test 1 —



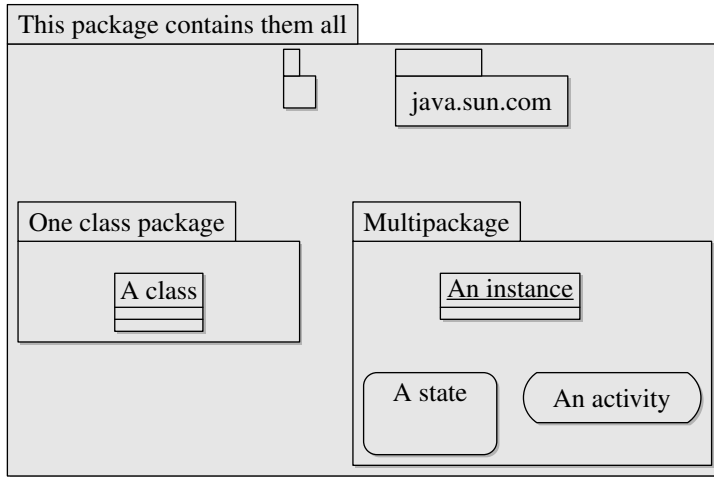
Test 2 —



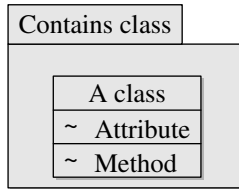
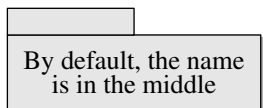
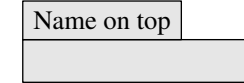
15.5 Package diagram

15.5.1 Package tests

Test 1 —



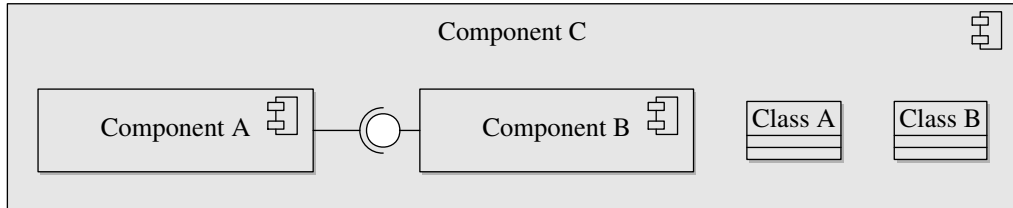
Test 2 —



15.6 Component diagram

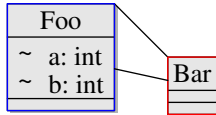
15.6.1 Component tests

Test 1 —

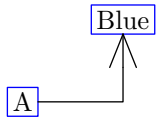


15.7 Paths

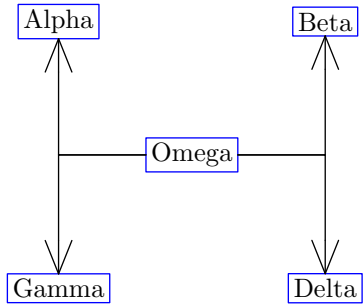
Test 1 —



Test 2 —



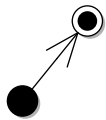
Test 3 —



15.8 Behavioral diagrams

15.8.1 Activity tests

Test 1 —

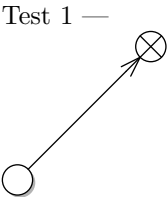


Test 2 —



go to school
while singing

15.8.2 State Machine tests



Test 2 —

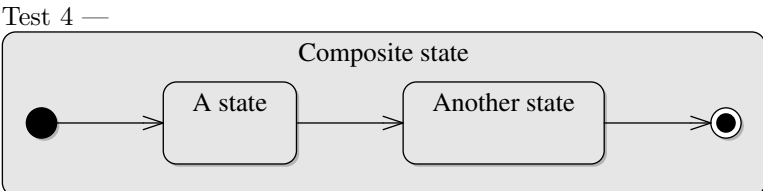
Another nice state

The light is
visibly on

Test 3 —

Interesting state

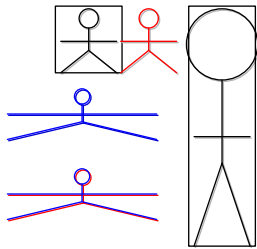
OnEntry / doVeryHappy
OnExit / doSomewhatSad



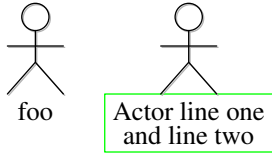
Test 5 —

15.8.3 Usecase tests

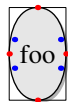
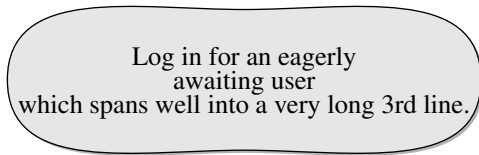
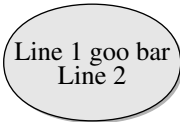
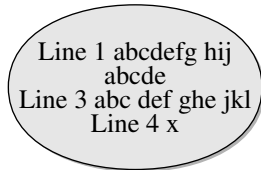
Test 1 —



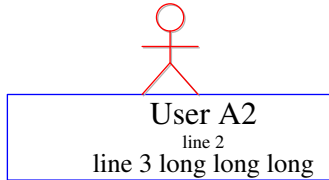
Test 2 —



Test 3 —




Test 4 —



Test 5 —


User A
Specifically configured

Test 6 —


User A
Globally configured

Test 7 —

A highly customizable
usecase. Foo bar!

Test 8 —

A highly customizable
usecase. Foo bar 2!

Test 9 —

A highly
customizable usecase.

Another very
customizable usecase.

15.9 Miscellaneous

15.9.1 Notes

Test 1 —

This is the first line
and this the second one.

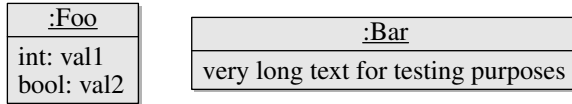
Test 2 —

Please take the other note
very seriously.

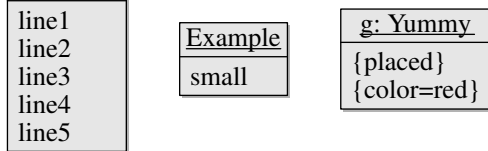
Please disregard this note.

15.9.2 Objects (Class Instances)

Test 1 —

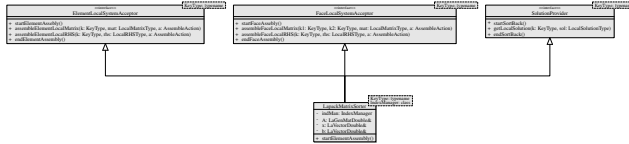


s: Student

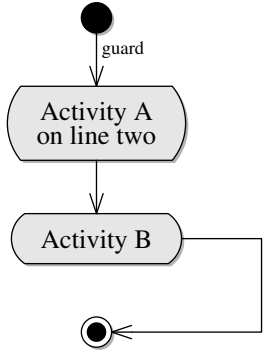


15.10 User requests

Test 1 —

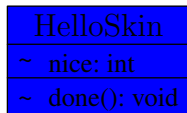


Test 2 —



15.11 Skins

Test 1 —



Test 1 —

HelloSkinGlobal
~ foo:int
~ bar():void

16 References

- [1] J. D. Hobby, *METAPOST A User's Manual*, 2018. [Online]. Available: <http://www.tug.org/tutorials/mp/mpman.pdf>.
- [2] *OMG® Unified Modeling Language® (OMG UML®)*, 2017. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1/>.
- [3] D. E. Knuth, *The T_EXbook*. Addison-Wesley Publishing Company, 1986.
- [4] L. Lamport, *L^AT_EX a Document Preparation System*. Addison-Wesley Publishing Company, 1994.
- [5] D. Roegel, *The METAOBJ tutorial and reference manual*, 2002. [Online]. Available: <http://texdoc.net/texmf-dist/doc/metapost/metaobj/momanual.pdf>.
- [6] E. F. Gjelstad, *Uml.sty, a package for writing UML diagrams in L^AT_EX*, 2010. [Online]. Available: <http://mirror.hmc.edu/ctan/graphics/pstricks/contrib/uml/uml.pdf>.
- [7] M. Diamantini, *Interface utilisateur du package pst-uml*, 2006. [Online]. Available: <http://mirrors.nxthost.com/ctan/graphics/pstricks/contrib/pst-uml/pst-uml-doc.pdf>.
- [8] D. Palmer, *The umldoc UML Documentation Package*, 1999. [Online]. Available: <https://www.charvolant.org/elements/umldoc.pdf>.
- [9] N. Kielbasiewicz, *The TikZ-UML package*, 2016. [Online]. Available: <http://perso.ensta-paristech.fr/~kielbasi/tikzuml/var/files/doc/tikzumlmanual.pdf>.
- [10] J. B. C. Engelen, *How to include an svg image in latex*. [Online]. Available: <http://tug.ctan.org/info/svg-inkscape/InkscapePDFLaTeX.pdf>.