

## MetaUML: Tutorial, Reference and Test Suite

Copyright ©2005-2006 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. A copy of the license is included in the section entitled "GNU Free Documentation License".

This page is intentionally left blank.

# MetaUML: Tutorial, Reference and Test Suite

Ovidiu Gheorghies

August 27, 2006

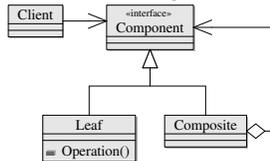
## Abstract

MetaUML is a GNU GPL MetaPost library for typesetting UML diagrams, using a human-friendly textual notation. MetaUML offers a highly customizable, object-oriented API, designed with the ease of use in mind. Apart from being a reference, this manual is also a tutorial but, more importantly, a living example. You can look at its source code, getting direct accounts on “how things are done”.

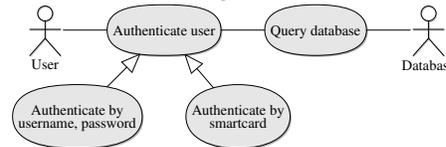
## 1 Introduction

Here are a few diagrams created with MetaUML, just to give you a glimpse of its features:

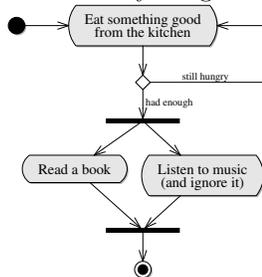
**A** Class Diagram



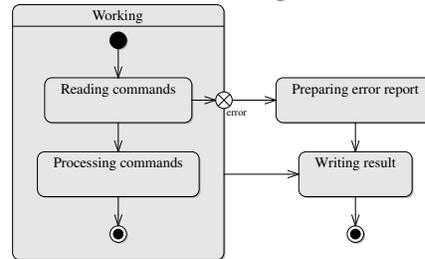
**D** Use Case Diagram



**B** Activity Diagram



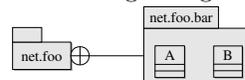
**E** State Machine Diagram



**C** Notes



**F** Package Diagram



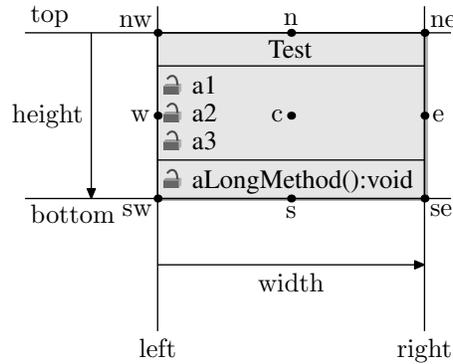
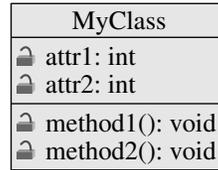


Figure 1: Positioning properties of any MetaUML object (here a class object is depicted).

The code which generates these diagrams is quite straightforward, combining a natural object-oriented parlance with the power of MetaPost equation solving; for more information on MetaPost see [Hobby, 1992].

An UML class, for example, can be drawn as follows:

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



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

This piece of code creates an instance of `Class`, which will be afterward identified as `A`. This object has the following content properties: a name (`MyClass`), a list of attributes (`attr1`, `attr2`) and a list of methods (`method1`, `method2`). The one thing remaining before actually drawing `A` is to set its location.

In `A.nw` we refer to the “north-west” of the class rectangle, that is to its upper-left corner. In general, every MetaUML object has the positioning properties given in figure 1. These properties are used to set where to draw a given object, whether by assigning them absolute values, or by setting them relatively to other objects. Suppose that we have defined two classes `A` and `B`. Then the following code would give a conceivable positioning:

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



After the objects are drawn, one may draw links between them, such as inheritance or association relations between classes in class diagrams, or tran-

sitions between states in state machine diagrams. Whichever the purpose is, MetaUML provides a generic way of drawing an edge in a diagram's graph:

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

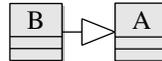
The “how to draw information” is actually an object which defines the style of the line (e.g. solid, dashed) and the appearance of the heads (e.g. nothing, arrow, diamond). One such object, called **inheritance**, defines a solid path ending in a white triangle. The **path-to-draw** parameter is simply a MetaPost path. For example, the following code can be used to represent that class B is derived from A:

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

Note that the direction of the path is important, and MetaUML uses it to determine the type of adornment to attach at 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.

To sum up, we present a short code and the resulting diagram, typical for just about everything else in MetaUML. The positioning of A does not need to be explicitly set because “floating” objects are automatically positioned at (0,0) by their draw method.

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



From a user's perspective, this is all there is to MetaUML. With a reference describing how other UML elements are created, one can set out to typeset arbitrary complex diagrams.

## 2 Class Diagrams

A class is created as follows:

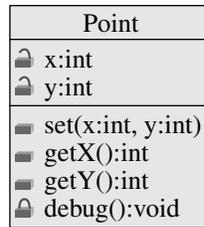
```
Class.name(class-name)
  (list-of-attributes)
  (list-of-methods);
```

The suffix `name` gives a name to the `Class` object (which, of course, represents an UML class). The name of the UML class is a string given by `class-name`; the attributes are given as a comma separated list of strings, `list-of-attributes`; the methods are given as a comma separated list of strings, `list-of-attributes`. The list of attributes and the list of methods may be void.

Each of the strings representing an attribute or a method may begin with a visibility marker: “+” for public, “#” for protected and “-” for private. MetaUML interprets this marker and renders a graphic stereotype in form of a lock which may be opened, semi-closed and closed, respectively.

Here is an example:

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



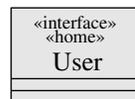
## 2.1 Stereotypes

After a class is created, its stereotypes may be specified by using the macro `classStereotypes`:

```
classStereotypes.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") ();
classStereotypes.A("<<interface>>",
                  "<<home>>");
```



```
drawObject(A);
```

## 2.2 Interfaces and Abstract Classes

At times it is preferred to typeset 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.name(class-name)
  (list-of-methods);
```

Here is an example:

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

<i>Observer</i>
■ update(src:Object)

```
drawObject(A);
```

Note that `Interface` is a special kind of `Class`, the declaration code above being equivalent to:

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

Along the same line, here's how abstract classes can be drawn:

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

<i>Observable</i>
🔒 observers: Observer[0..*]
■ addObserver(o: Observer)
■ notify()

```
drawObject(A);
```

If you prefer, you can use the syntactic sugar:

```
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.iAttributeStack.top := 0;
A.info.iAttributeStack.bottom := 0;
A.info.iMethodStack.top := 0;
A.info.iMethodStack.bottom := 0;
```

MyModel
---------

```
drawObject(A);
```

There are also specific formatting macros defined for this usage scenarios, the formatting information object `iClassNameOnly` and the syntactic sugar macro `ClassName`:

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

MyModel
---------

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

«smart»
AnotherModel

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)

An 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);
```

<u>o: Order</u>
name='book'
{placed}
{payed}

## 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.

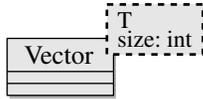
Here is an example:

```

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, one can specify both stereotypes and template parameters for a given class.

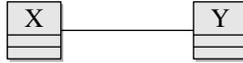
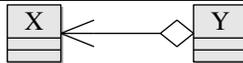
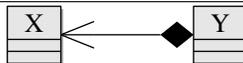
## 2.6 Types of Links

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);

```

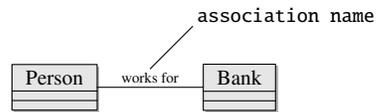
<code>link(association)(X.e -- Y.w)</code>	
<code>link(associationUni)(X.e -- Y.w)</code>	
<code>link(inheritance)(X.e -- Y.w)</code>	
<code>link(realization)(X.e -- Y.w)</code>	
<code>link(aggregation)(X.e -- Y.w)</code>	
<code>link(aggregationUni)(X.e -- Y.w)</code>	
<code>link(composition)(X.e -- 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). Note that the last parameter of the macro `item` is an equation which uses the item name to perform positioning.

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



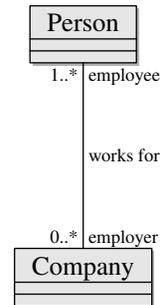
```
item.aName(iAssoc)("works for")
      (aName.s = .5[P.w, C.w]);
draw aName.n -- (aName.n + (20,20));
label.urt("association name" infont "tyxtt",
          aName.n + (20,20));
```

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
```



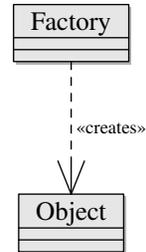
## 2.8 Dependencies and Stereotypes

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

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

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

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



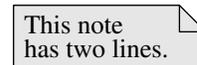
### 3 Notes

A note is created as follows:

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

The suffix `name` is the name of the `Note` object. The 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

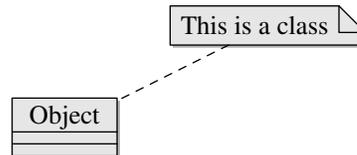
Notes can be attached to diagram elements by using a link of type `dashedLink`.

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

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

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

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



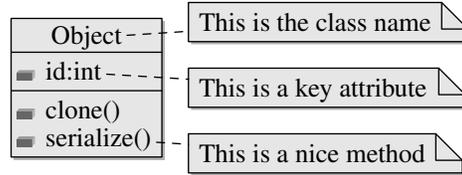
Now let us see a more complex example, which demonstrates the ability of accessing sub-elements in a MetaUML diagram.

```

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.

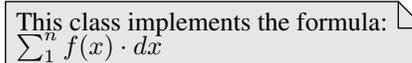
### 3.2 Using mathematical formulae

MetaUML notes can contain mathematical formulae typeset in TeX (and **not** LaTeX). This feature is considered experimental, as it is not always straightforward to use it. Below is a full-code example, note the import of MetaPost package `TEX`.

```

input metauml;
input TEX;

```



```

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

```

```

end

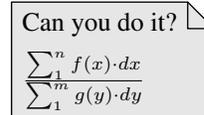
```

If you have a taller formula, you must be prepared to do some advanced stunts. Remark: `"aaa" & "bbb"` is MetaPost's way to concatenate the strings into `"aaabbb"`. The formula's string was split in two for space 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);

```

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);
```

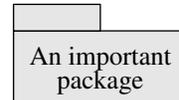
The parameter `package-name` is a string or a list of comma separated strings representing the package's name. The `subitems-list` parameter is used to specify the subitems (typically classes or packages) of this package; its form is as a comma separated list of objects, which can be void.

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



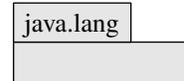
Below is another example:

```
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.

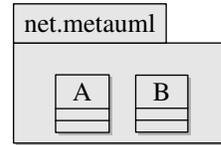
```

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

leftToRight(10)(A, B);

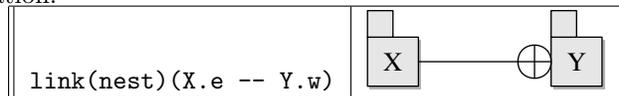
drawObject(P);

```



## 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:

```

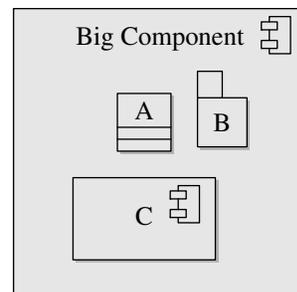
Class.A("A")();
Package.B("B")();
Component.C("C")();

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

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

drawObject(BigC);

```

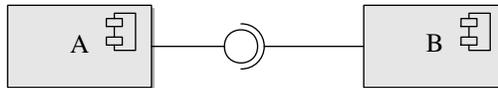


## 5.1 Types of Links

<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

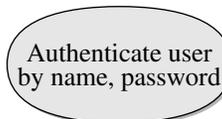
An use case is created by the macro `Usecase`:

```
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);
```



Note that one may prefer 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);
```



Note that 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, containing the text "Take order".

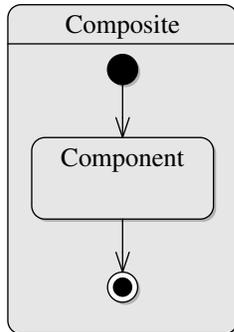
### 8.1 Composite States

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

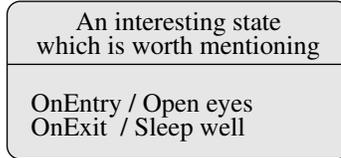
Internal transitions can be specified by using the macro:

```
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);

```



Regardless of how amusing this feature might be, it does become a bit of a nuisance to use it in its bare form. When typesetting UML diagrams in good style, one generally uses rectangular paths. It is for this kind of style that MetaUML offers extensive support, providing a “syntactic sugar” for constructs which can otherwise be done by hand, but with some extra effort.

## 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:

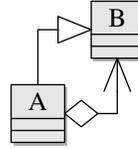
```

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)

```

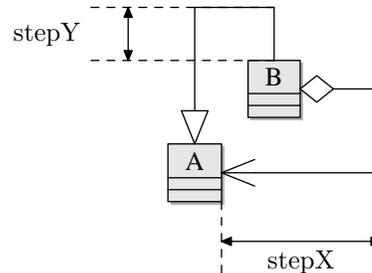
Here is an example:

```

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)

```

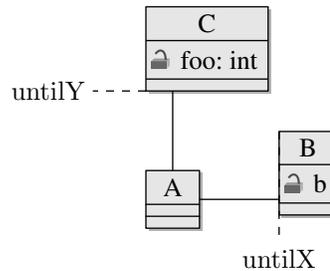
```
rpathHorizontal(pA, untilX)
rpathVertical(pA, untilY)
```

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, it is clear that one can draw links between diagram objects, using a code like:

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

There are times however this may yield unsatisfactory visual results, especially when the appearance of the object’s corners is round. 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)
```

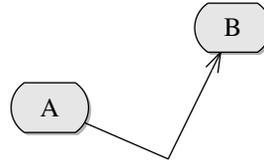
Here, `thePath` is a given MetaPost path and `objectA` and `objectB` are two MetaUML objects. By contract, each MetaUML object of type, say, `X` defines a macro `X.border` which returns the path that surrounds it. Because of that, `pathCut` can make the appropriate modifications to `thePath`.

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

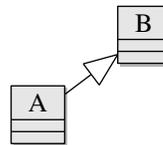
At times is quicker to just draw direct paths between the center of two objects, minding of course the object margins. The macro which does this is `click`:

```
click(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:

```
click(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. A code like the following will do:

```

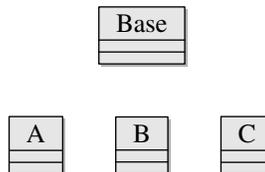
A.ne = B.nw + (20,0);
B.ne = C.nw + (20,0);
Base.s = B.n + (0,-20);

```

Now, look at the code again. What strikes you is that you cannot visualize what it is all about, unless you really try — decoding the intent line by line. What this code lacks is a feature called self-documenting: the code is good only if you can read it as a story and understand its meaning.

Perhaps the following version of the code will make the point. All you need to know is that the numeric argument represents a distance.

```
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>	A diagram showing three boxes labeled 'a', '...', and 'Cyan' arranged horizontally. Red dashed lines indicate the spacing between 'a' and '...', and between '...' and 'Cyan'.
<code>leftToRight.midy(spacing)(X, Y, Z);</code>	A diagram showing three boxes labeled 'a', '...', and 'Cyan' arranged horizontally. Red dashed lines indicate the spacing between 'a' and '...', and between '...' and 'Cyan'. The spacing is centered vertically between the boxes.
<code>leftToRight.bottom(spacing)(X, Y, Z);</code>	A diagram showing three boxes labeled 'a', '...', and 'Cyan' arranged horizontally. Red dashed lines indicate the spacing between 'a' and '...', and between '...' and 'Cyan'. The spacing is at the bottom of the boxes.
<code>topToBottom.left(spacing)(X, Y, Z);</code>	A diagram showing three boxes labeled 'a', '...', and 'Cyan' stacked vertically. Red dashed lines indicate the spacing between 'a' and '...', and between '...' and 'Cyan'. The spacing is on the left side of the boxes.
<code>topToBottom.midx(spacing)(X, Y, Z);</code>	A diagram showing three boxes labeled 'a', '...', and 'Cyan' stacked vertically. Red dashed lines indicate the spacing between 'a' and '...', and between '...' and 'Cyan'. The spacing is centered horizontally between the boxes.
<code>topToBottom.right(spacing)(X, Y, Z);</code>	A diagram showing three boxes labeled 'a', '...', and 'Cyan' stacked vertically. Red dashed lines indicate the spacing between 'a' and '...', and between '...' and 'Cyan'. The spacing is on the right side of the boxes.

To make typesetting even quicker in frequent usage scenarios, 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>

To specify the relative position of two points more easily, one can use 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` ([Roegel, 2002]) 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). Also, it had a degree of flexibility which became apparent to be insufficient.

### 11.1 Motivation

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

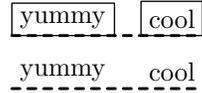
```

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 bottoms alligned, “yummy” *looks* slightly higher than “cool”. This would be unacceptable in an UML class diagram, when roles are placed at the ends of a horizontal association. Regardless of 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;

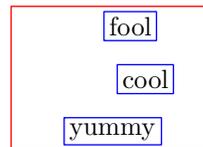
Picture.a("yummy");
Picture.b("cool");
Picture.c("fool");

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);

```



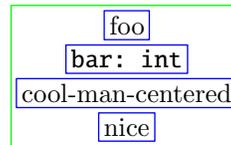
Note that after some objects are grouped, they can all be drawn by invoking the `drawObject` macro solely on the group that aggregates them. Another important remark is that it is necessary only to set the relative positioning of

objects within a group (line A and B); afterward, one can simply “move” the group to a given 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, one must know the following information:

- 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 come in handy when one wants to quickly place Picture objects near others, 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
```

Note that an object mustn’t 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, an association has no heads, and one must simply draw along the path with a solid pen. An unidirectional aggregation has a solid path and two heads: one is an arrow and the other is a diamond. So the template 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

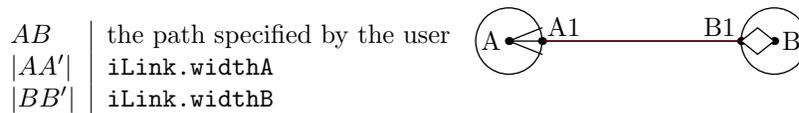


Figure 2: Details on how a link is drawn by MetaUML.

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;
```

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.

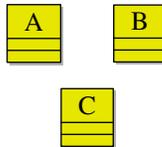
### 13.1 Global settings

Let us assume that we do not particularly like the default foreground color of all classes, and wish to change it so 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

When one wants to make modifications to 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);

```

The diagram shows three class objects. Object A is a yellow rectangle with the letter 'A' inside. Object B is a yellow rectangle with the letter 'B' inside. Object C is a blue rectangle with the letter 'C' inside, and the letter 'C' is significantly larger than the letters in A and B. All three objects have a thin black border.

As an aside, note that for each `Class` object its `info` member is created as a copy of `iClass`: the actual drawing is performed using this copied information. Because of that, one can modify the `info` member after the object has been created and still get the desired results.

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:

```

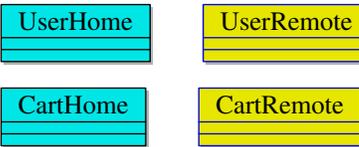
ClassInfoCopy.iHome(iClass);
iHome.foreColor := (0, .9, .9);

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, one may also want to modify the spacings between the attributes' and methods' baselines. Figure below is the result of the (unlikely) code:

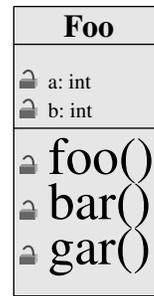
```
Class.A("Foo")
  ("a: int", "b: int")
  ("foo()", "bar()", "gar()");

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.

## References

- [Roegel, 2002] Roegel, D. (2002). The METAOBJ tutorial and reference manual. Available from [www.loria.fr/~roegel/TeX/momanual.pdf](http://www.loria.fr/~roegel/TeX/momanual.pdf).
- [Knuth, 1986] Knuth, D. E. (1986). *The T<sub>E</sub>Xbook*. Addison-Wesley Publishing Company.
- [Lamport, 1994] Lamport, L. (1994). *L<sup>A</sup>T<sub>E</sub>X a Document Preparation System*. Addison-Wesley Publishing Company, 2nd edition.

- [Hobby, 1992] Hobby, J. (1992) A User's Manual for MetaPost. Available from <http://www.tug.org/tutorials/mp/>.
- [Gjelstad, 2001] Gjelstad, E. (2001). `uml.sty` 0.09.09. Available from <http://heim.ifi.uio.no/~ellefg/uml.sty/>.
- [Diamantini, 1998] Diamantini, M. (1998). Interface utilisateur du package `pst-uml`. Available from <http://perce.de/LaTeX/pst-uml/>.
- [Palmer, 1999] Palmer, D. (1999). The `umldoc` UML Documentation Package. Available from <http://www.charvolant.org/~elements/>.
- [OMG, 2003] Object Management Group (2003). XML Metadata Interchange (XMI) Specification. Available from <http://www.omg.org/>.

## 14 Test suite

### 14.1 Low-level

Test 1 —  
nothing-shown-(intentionally)

Test 2 —  
nothing-shown-(intentionally)

### 14.2 Fonts

Test 1 —

```
Font name: ( ) pcurr  
<<stereotype>>  
<<a>>, <<b>>, <<c>>  
[guard]  
[stillhungry] closing paranthesis SHOWN after hungry !  
[still hungry] closing paranthesis NOT shown after hungry !  
[] [] [] [] hm  
{constraint}  
{a constraint} closing paranthesis NOT shown !
```

Test 2 —

```
Font name: ( ) tyxbtt  
«stereotype»  
«a», «b», «c»  
[guard]  
[stillhungry] closing paranthesis SHOWN after hungry !  
[still hungry] closing paranthesis NOT shown after hungry !  
[] [] [] [] hm  
{constraint}  
{a constraint} closing paranthesis NOT shown !
```

Test 3 —

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

### 14.3 Cliparts



### 14.4 Util library

#### 14.4.1 Picture tests

Test 1 —  
nice, ugly

yy  
xxx  
foo bar, foo  
**what a nice feature**

Test 2 —  
toof

root  
bar  
foo  
Test 3 —  
goof  
-----

a  
f: int goofy: int goot  
cool  
good  
tust  
fook  
dddd  
ast  
good

Test 4 —  
goof Aoorian fpp f: int aa()  
goof Aoorian fpp f: int aa()

Test 5 —  
«foo»

Test 6 —  
x: int  
an-anonymous-item  
foo-bar-cool

Test 7 —  
a yummy .-. job football

Test 8 —  
a yummy .-. job football

Test 9 —

a
yummy
.-.
job
football

Test 10 —

a
yummy
.-.
job
football

#### 14.4.2 Picture tests - TeX rendering

Test 1 —

Hello, world  $x = 7$

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

#### 14.4.3 Group tests

Test 1 —

p0  
p1

Test 2 —

s
s

□

Test picture in group

#### 14.4.4 PictureStack tests

Test 1 —

Test 2 —  
foo

bar: int

cool-man

nice

Test 3 —

foornika	foornika	foornika
gar-nichts	gar-nichts	gar-nichts
nicelina	nicelina	nicelina

#### 14.4.5 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

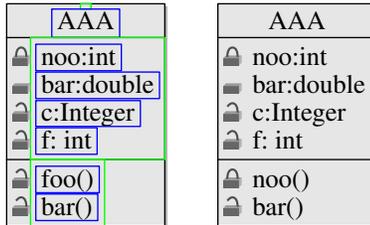
...

XYZ

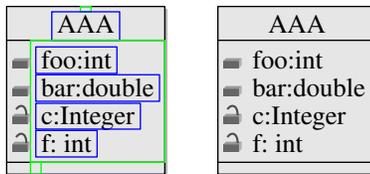
## 14.5 Class diagram

### 14.5.1 Class tests

Test 1 —



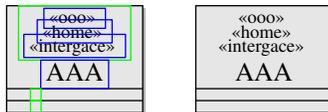
Test 2 —



Test 3 —



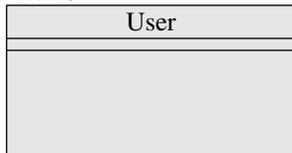
Test 4 —



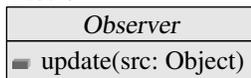
Test 5 —



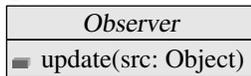
Test 6 —



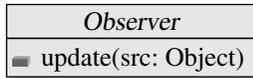
Test 7 —



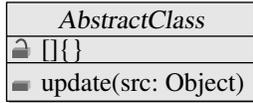
Test 8 —



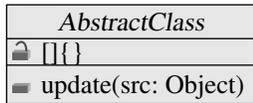
Test 9 —



Test 10 —



Test 11 —



Test 12 —



Test 13 —

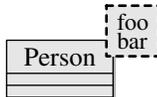


Test 14 —

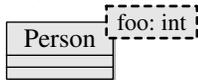


### 14.5.2 Class template tests

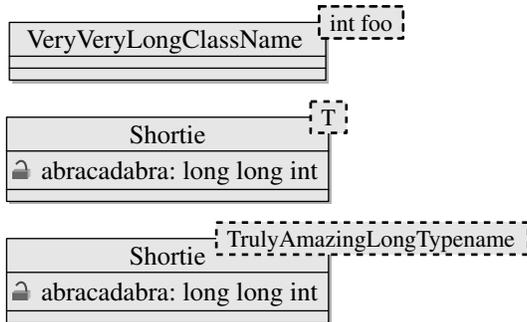
Test 1 —



Test 2 —

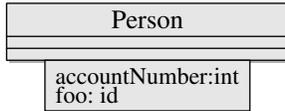


Test 3 —

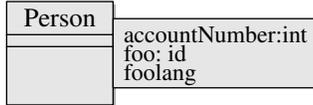


### 14.5.3 Qualified Association tests

Test 1 —



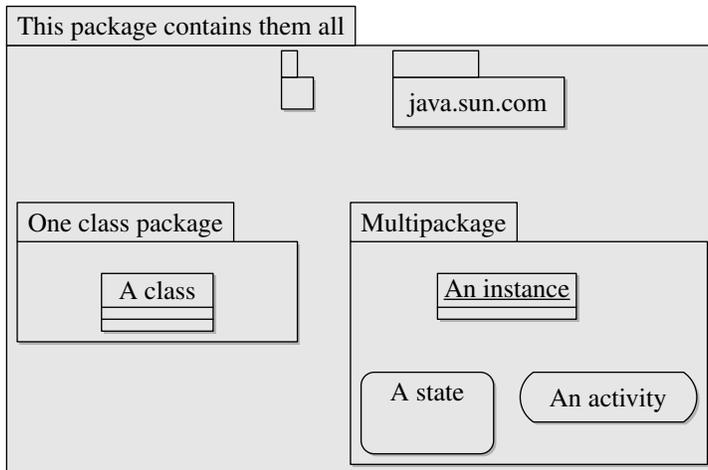
Test 2 —



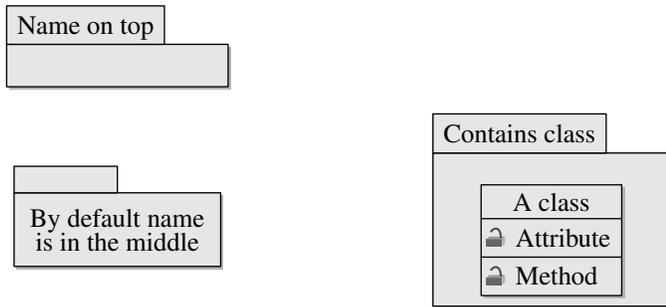
## 14.6 Package diagram

### 14.6.1 Package tests

Test 1 —



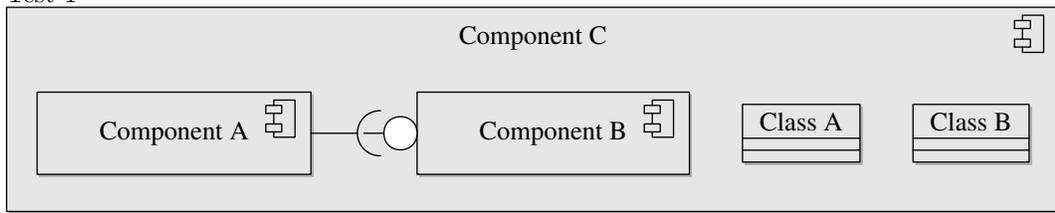
Test 2 —



## 14.7 Component diagram

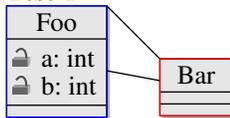
### 14.7.1 Component tests

Test 1 —

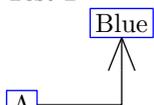


## 14.8 Paths

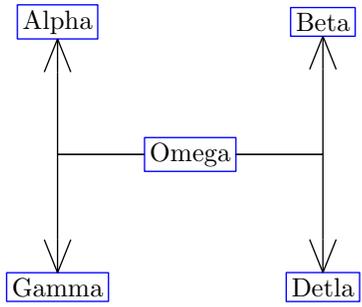
Test 1 —



Test 2 —

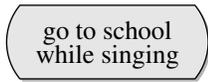
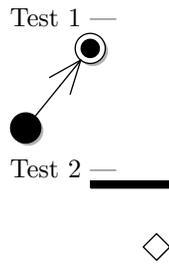


Test 3 —

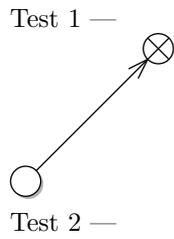


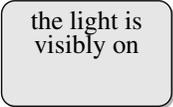
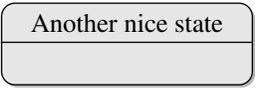
## 14.9 Behavioral diagrams

### 14.9.1 Activity tests

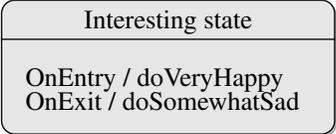


### 14.9.2 State Machine tests

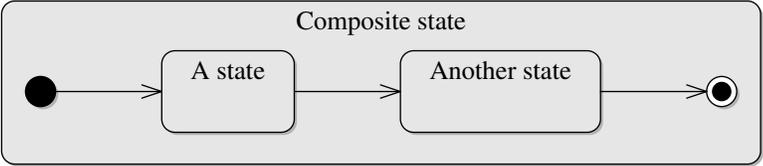




Test 3 —



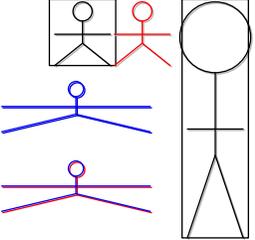
Test 4 —



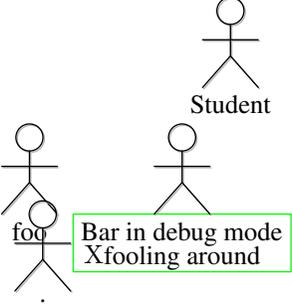
Test 5 —

**14.9.3 Usecase tests**

Test 1 —



Test 2 —

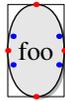


Test 3 —

foo xasdf asdf as  
asdfa  
cru asdf asdf ygh  
Sdfg s

foo xasdf asdf as  
asdfa

Log in for an eagerly  
awaiting user  
which spans the 3rd line



Test 4 —



User A2  
doesn't look all too nice  
by today's standards

Test 5 —



User A  
on two lines

Test 6 —



User A  
reloaded

Test 7 —

A highly customizable  
usecase. Foo bar!

Test 8 —

A highly customizable usecase. Foo bar!

Test 9 —

A highly customizable usecase.

Another very customizable usecase.

## 14.10 Miscellaneous

### 14.10.1 Notes

Test 1 —

Antananarivo  
Machupichu

Test 2 —

Please take the other note very seriously

Please disregard this note

### 14.10.2 Objects (Class Instances)

Test 1 —

:Foo  
int: val1  
bool: val2

:Bar  
very long text for testing purposes

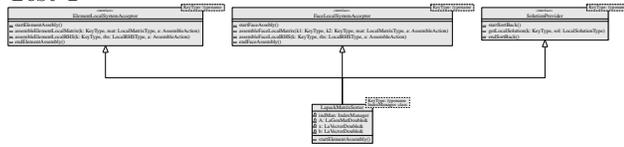
s: Student  
line1  
line2  
line3  
line4  
line5

Example  
small

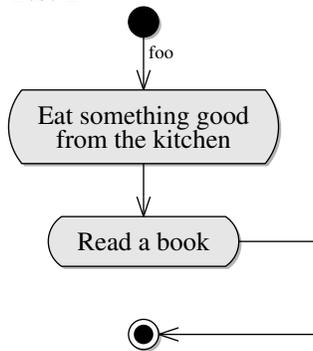
g: Yummy  
{placed}  
{color=red}

## 14.11 User requests

Test 1 —



Test 2 —



## 14.12 Skins

Test 1 —

HelloSkin
🔒 nice:int
🔒 done():void

Test 1 —

HelloSkinGlobal
🔒 foo:int
🔒 bar():void

## 15 GNU Free Documentation License

Version 1.2, November 2002  
Copyright ©2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary

Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## **2. VERBATIM COPYING**

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## **3. COPYING IN QUANTITY**

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network

location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all

of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## **6. COLLECTIONS OF DOCUMENTS**

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## **7. AGGREGATION WITH INDEPENDENT WORKS**

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## **8. TRANSLATION**

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders,

but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

### **ADDENDUM: How to use this License for your documents**

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright ©YEAR YOUR NAME. 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. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.