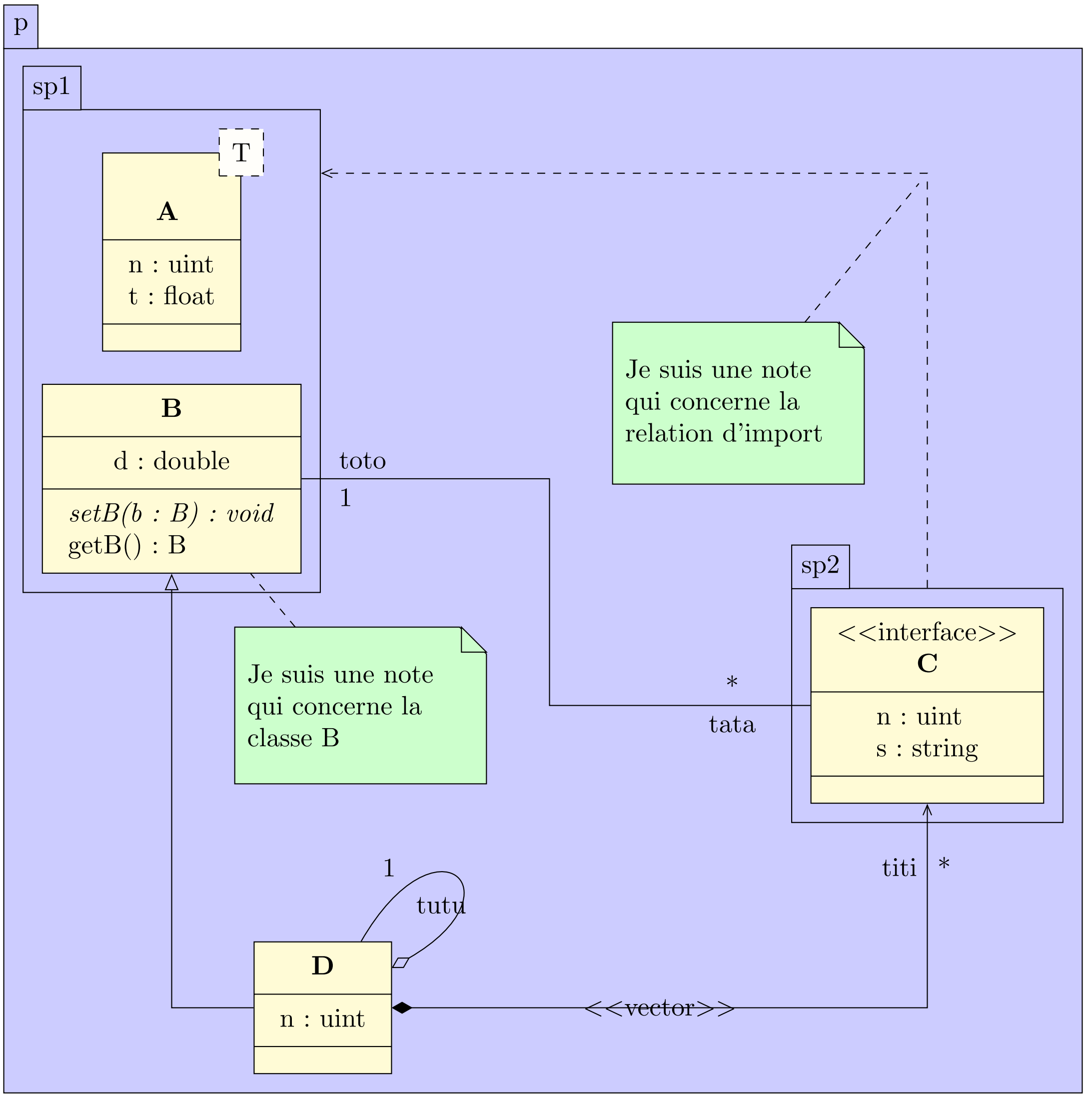
Le package TikZ-uml

Nicolas Kielbasiewicz

24 août 2011

Devant les immenses possibilités offertes par la librairies PGF/TikZ, et devant l'absence apparente de package dédié aux diagrammes UML, j'ai été amené à développer le package TikZ-UML, proposant un ensemble de commandes et d'environnements spécifiques à ce type de diagrammes. Il s'inscrit dans la logique du package pst-uml développé pour des raisons similaires en PSTricks. Dans son état actuel, la librairie permet de définir des diagrammes de classes complets de manière assez simple, ainsi que des diagrammes de cas d'utilisation, des diagrammes de séquence, et des diagrammes d'états-transitions. Un certain nombre d'améliorations sont encore à apporter, mais cela se rapproche de l'état final.

Voici un exemple de diagramme de classes que l'on peut réaliser :



Nous allons maintenant vous présenter les différentes fonctionnalités offertes par TikZ-UML.

Table des matières

Chapitre 1

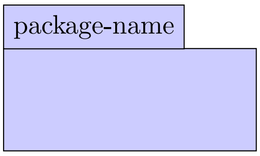
Diagrammes de classes

1.1 Package, classe, attributs et opérations

1.1.1 Définir un package

Un package est défini par l'environnement `umlpackage` :

```
\begin{tikzpicture}
\begin{umlpackage}[x=0,y=0]{package-name}
\end{umlpackage}
\end{tikzpicture}
```



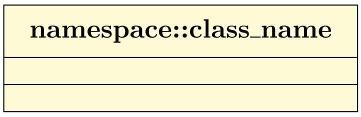
Les options `x` et `y` permettent de définir la position du package dans la figure. Elles valent toutes deux 0 par défaut.

- Quand un package contient des classes ou des sous-packages, sa taille s'ajuste automatiquement à son contenu.
- On peut définir autant de packages que l'on veut dans une figure.
- Il existe un raccourci pour définir un package qui sera vide (on ne définira pas de classes à l'intérieur) : la commande `umlemptypackage` qui prend les mêmes arguments que l'environnement `umlpackage`

1.1.2 Définir une classe

Pour définir une classe, on utilise l'environnement `umlclass`, de la même manière que l'environnement `umlpackage` :

```
\begin{tikzpicture}
\umlclass[x=0,y=0]{namespace::class\_name
    }{}{}
\end{tikzpicture}
```

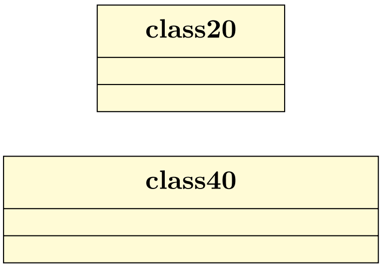


Les options `x` et `y` définissent la position de la classe. 2 possibilités : si la classe est définie dans un package, il s'agit de la position relative de la classe dans le package; dans le cas contraire, il s'agit de la position dans la figure. Elles valent toutes deux 0 par défaut. Comme pour le package, il existe un raccourci pour définir une classe vide (dont on ne définira pas les attributs et les opérations) : la commande `umlemptyclass`, qui prend les mêmes arguments que l'environnement `umlclass`.

Définir la largeur d'une classe

Par défaut, la largeur d'une classe est de 10ex. On peut la modifier avec l'option `width` :

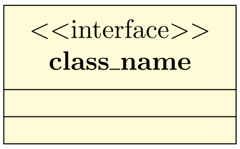
```
\begin{tikzpicture}
\umlemptyclass[width=15ex]{class20}
\umlemptyclass[y=-2, width=30ex]{class40}
\end{tikzpicture}
```



Spécifier le type d'une classe

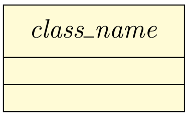
La classe peut être de différents types : class, interface, abstract, typedef, enum. On utilise l'option `type` :

```
\begin{tikzpicture}
\umlemptyclass[type=interface]{class\_
   name}
\end{tikzpicture}
```



Le type apparaît entre guillemets au dessus du nom de la classe, sauf dans le cas d'une classe où rien ne s'affiche (comportement par défaut), et dans le cas d'une classe abstraite, où le nom de la classe s'écrit en italique :

```
\begin{tikzpicture}
\umlemptyclass[type=abstract]{class\_name
   }
\end{tikzpicture}
```

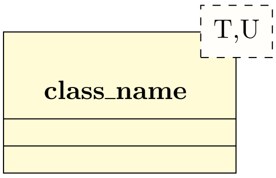


A noter qu'il existe des alias pour chaque valeur du type : `umlabstract`, `umltypedef`, `umlenum`, `umlinterface`.

Spécifier des paramètres templates

La classe peut être un template. On spécifie la liste des paramètres avec l'option `template` :

```
\begin{tikzpicture}
\umlemptyclass[template={T,U}]{class\_
   name}
\end{tikzpicture}
```



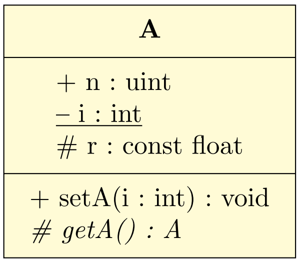
Nom du nœud représentant une classe

Pour donner le nom d'une classe, il arrive que l'on utilise des caractères spéciaux, comme le _ ou les : quand on spécifie le namespace. Le mécanisme interne de TikZ-UML nomme le nœud définissant une classe avec le même nom. Or, l'utilisation du \ et des : est interdite pour nommer un nœud en TikZ. Il est tout à fait possible que d'autres caractères viennent poser problème. Il faut donc procéder à une substitution de caractère, opération appelée dès que l'on définit ou utilise le nom d'une classe. Nous avons vu dans les exemples précédents que cela fonctionne pour le _.

1.1.3 Définir des attributs et des opérations

On définit les attributs à l'intérieur d'un environnement `umlclass` à l'aide de la commande `umlattr`. On lui passe en argument la liste des arguments en les séparants par \\. On procède de même pour les opérations avec la commande `umlop` :

```
\begin{tikzpicture}
\umlclass{A}{
  + n : uint \\ \umlstatic{-- i : int} \\
      \# r : const float
  }{
  + setA(i : int) : void \\ \umlvirt{\#
      getA() : A}
}
\end{tikzpicture}
```



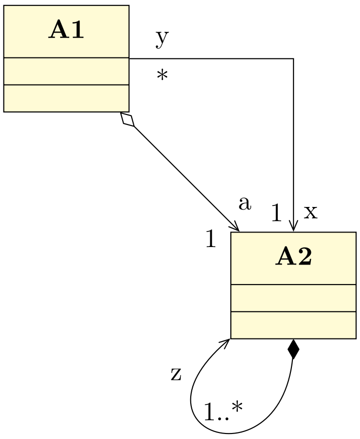
Pour définir un attribut ou une méthode statique, on peut utiliser la commande `umlstatic`. De même, la commande `umlvirt` permet de spécifier des fonctions virtuelles.

1.2 Relations entre classes

1.2.1 Commande générale

Chaque classe et chaque package sont représentés par un nœud ayant le même nom. Pour définir une relation entre 2 classes, on va donc spécifier le nom de la classe de départ, le nom de la classe d'arrivée et un certain nombre d'options propres à la relation.

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=3,y=-3]{A2}
\umluniaggreg[arg2=a,mult2=1,pos2=0.9]{A
    1}{A2}
\umluniassoc[geometry=-|,arg1=x,mult1=1,
    pos1=1.9,arg2=y,mult2=*,pos2=0.2]{A1}{
    A2}
\umlunicompo[arg=z,mult=1..*,pos=0.8,
    angle1=-90, angle2=-140, loopsize=2cm
    ]{A2}{A2}
\end{tikzpicture}
```



D'un point de vue sémantique, il existe 11 relations différentes, toutes présentes dans TikZ-UML :

La dépendance : utiliser la commande `umldep`

L'association : utiliser la commande `umlassoc`

L'association unidirectionnelle : utiliser la commande `umluniassoc`

L'agrégation : utiliser la commande `umlaggreg`

L'agrégation unidirectionnelle : utiliser la commande `umluniaggreg`

La composition : utiliser la commande `umlcompo`

La composition unidirectionnelle : utiliser la commande `umlunicompo`

L'import : utiliser la commande `umlimport`

L'héritage : utiliser la commande `umlinherit`

L'implémentation : utiliser la commande `umlimpl`

La réalisation : utiliser la commande `umlreal`

Ces 11 commandes sont basées sur le même schéma (la commande `umlrelation`) et prennent en théorie exactement le même jeu d'options. En pratique, certaines options concernent certains types de relations.

1.2.2 Définir la géométrie de la relation

Comme vous avez pu le voir dans les exemples précédents, on peut spécifier la forme géométrique de la relation à l'aide de l'option `geometry`. Cette option demande un argument parmi la liste - - (ligne droite), -| (horizontal puis vertical), |- (vertical puis horizontal), -|- (chicane horizontale) ou |-| (chicane verticale), ces arguments étant largement inspirés de la philosophie TikZ.

Il apparaît à l'utilisation que cette option est très souvent utilisée. C'est la raison pour laquelle un alias de la commande `umlrelation` a été défini pour chacune des valeurs possibles de l'option `geometry` :

umlHVrelation : alias de umlrelation avec geometry=-|

umlVHrelation : alias de umlrelation avec geometry=|-

umlHVHrelation : alias de umlrelation avec geometry=-|-

umlVHVrelation : alias de umlrelation avec geometry=|-|

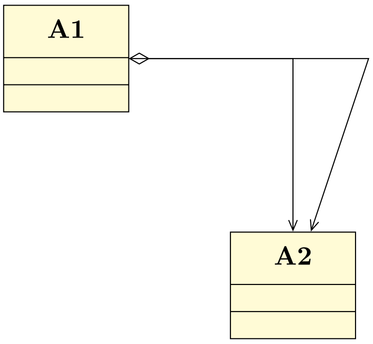
⚠ Pour chacun de ces 4 alias, l'option `geometry` est interdite.

⚠ Il n'y a pas d'alias dans le cas de la valeur - - pour la seule raison qu'il s'agit de la valeur par défaut.

1.2.3 Ajuster la géométrie de la relation

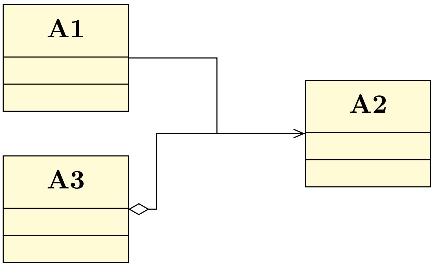
Lorsque la géométrie de la relation contient 2 segments, on peut spécifier les coordonnées du point intermédiaire, ou nœud de contrôle. Plutôt que la commande `umlrelation`, on utilisera `umlCNrelation`, elle aussi déclinée en 11 alias :

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=3,y=-3]{A2}
\umluniaggreg[geometry=-|]{A1}{A2}
\umlCNuniassoc{A1}{4,0}{A2}
\end{tikzpicture}
```



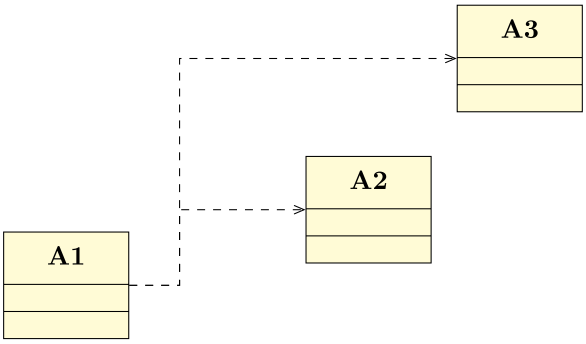
Lorsque la géométrie de la relation contient 3 segments, la position relative du segment central entre les classes est défini comme passant par le milieu entre les classes reliées. On peut ajuster ce paramètre à l'aide de l'option `weight` :

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=-1]{A2}
\umlemptyclass[y=-2]{A3}
\umlassoc[geometry=-|-]{A1}{A2}
\umluniaggreg[geometry=-|-, weight=0.3]{A
    3}{A2}
\end{tikzpicture}
```

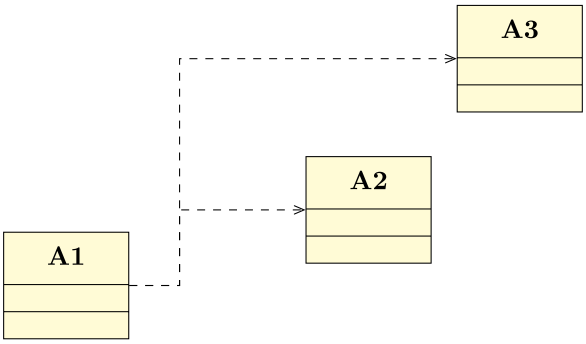


Dans certains cas, cette option est peu pratique, car elle demande de calculer la valeur à passer à l'option. On peut alors procéder autrement en utilisant les options `arm1` et `arm2` qui fixent la taille respectivement du premier et dernier segment. Regardons ici les deux exemples utilisant respectivement l'option `weight` et l'option `arm1` :

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=1]{A2}
\umlemptyclass[x=6, y=3]{A3}

\umlHVHdep[weight=0.375]{A1}{A2}
\umlHVHdep[weight=0.25]{A1}{A3}
\end{tikzpicture}
```

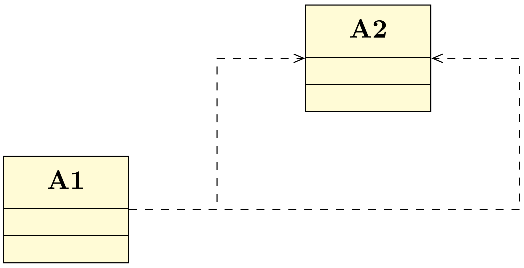


```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=1]{A2}
\umlemptyclass[x=6, y=3]{A3}

\umlHVHdep[arm1=1.5cm]{A1}{A2}
\umlHVHdep[arm1=1.5cm]{A1}{A3}
\end{tikzpicture}
```



Les options `arm1` et `arm2` prennent aussi des valeurs négatives. Que se passe t'il alors ? Si l'on passe une valeur positive, alors le bras sera orienté dans le sens normal (soit à droite, soit vers le haut). Si l'on passe une valeur négative, alors l'arc sera orienté dans l'autre sens, ce qui permet de dessiner d'autres types de relations à 3 segments, comme le montre l'exemple ci-dessous :

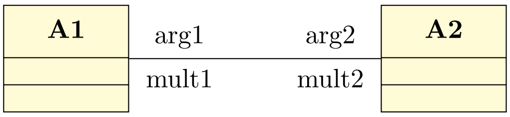
```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=2]{A2}

\umlHVHdep[arm2=-2cm]{A1}{A2}
\umlHVHdep[arm2=2cm]{A1}{A2}
\end{tikzpicture}
```



1.2.4 Définir des informations sur les attributs d'une relation

Une relation visualise la dépendance entre deux classes et se traduit généralement sous la forme d'un attribut. On peut spécifier son nom avec l'option `arg1` ou `arg2`, et sa multiplicité avec `mult1` ou `mult2` :

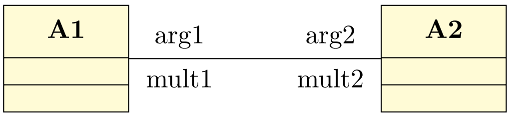
```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=5]{A2}
\umlassoc[arg1=arg1, mult1=mult1, arg2=
    arg2, mult2=mult2]{A1}{A2}
\end{tikzpicture}
```



Pour les relations unidirectionnelles, on ne va être amené à utiliser que les options `arg2` et `mult2`. Comme il paraît dans ce cas peu naturel que ces options finissent par 2, on peut utiliser les options `arg` et `mult` qui jouent le même rôle.

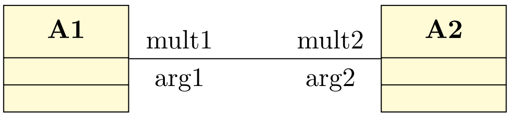
Par ailleurs, lorsqu'on définit à la fois le nom et la multiplicité d'un attribut, on peut le faire sous une forme plus contractée à l'aide des options `attr1`, `attr2` et `attr` :

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=5]{A2}
\umlassoc[attr1=arg1|mult1, attr2=arg2|
    mult2]{A1}{A2}
\end{tikzpicture}
```



L’avantage de cette forme contractée est de pouvoir s’affranchir de la sémantique entre le nom et la multiplicité et de pouvoir alors inverser les deux :

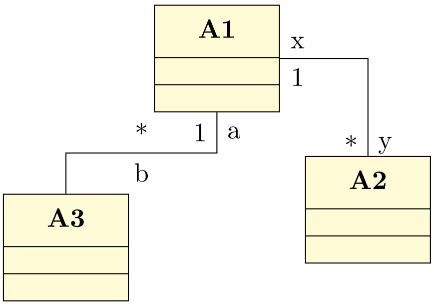
```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=5]{A2}
\umlassoc[attr1=mult1|arg1, attr2=mult2|
    arg2]{A1}{A2}
\end{tikzpicture}
```



1.2.5 Positionner les informations sur les attributs d’une relation

On peut positionner les informations définies dans la section précédente à l’aide des options `pos1`, `pos2` et `pos`. La commande `umlrelation` détermine elle-même si le nom et la multiplicité doivent être respectivement à gauche et à droite, ou au-dessus et au-dessous, de la flèche, selon sa géométrie et leur position. Pour les initiés à TikZ, elle se base sur les options `auto` et `swap`.

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=2,y=-2]{A2}
\umlemptyclass[x=-2,y=-2.5]{A3}
\umlassoc[geometry=-|,arg1=x,mult1=1,pos
    1=0.2,arg2=y,mult2=*,pos2=1.9]{A1}{A2}
\umlassoc[geometry=|-|,arg1=a,mult1=1,pos
    1=0.5,arg2=b,mult2=*,pos2=1.5]{A1}{A3}
\end{tikzpicture}
```

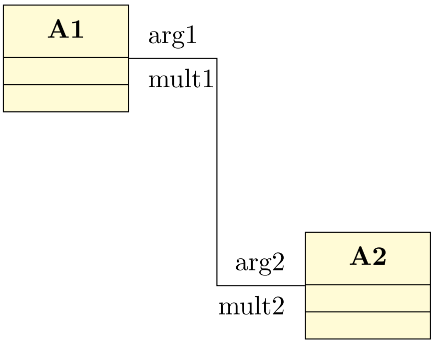


Il est à noter que l’intervalle de valeurs de la position d’un nom d’argument dépend du nombre de segments constituants la flèche de relation. Si la flèche est droite, alors la position doit être comprise entre 0 (classe de départ) et 1 (classe d’arrivée). Si la flèche comporte un seul angle droit, alors la position varie entre 0 et 2 (point d’arrivée), la valeur 1 correspondant à l’angle. Dans les deux autres possibilités, la position varie entre 0 et 3 (point d’arrivée), les valeurs 1 et 2 correspondant respectivement au premier et au deuxième angle.

1.2.6 Ajuster l’alignement des informations sur les attributs d’une relation

Par défaut, nom et multiplicité de l’argument, quand ils sont affichés l’un au dessus de l’autre, sont centrés. Les options `align1`, `align2` et `align` permettent de les justifier à gauche ou à droite.

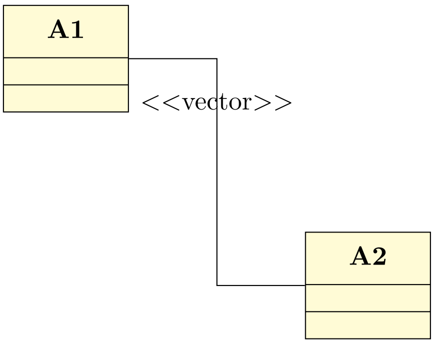
```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=-3]{A2}
\umlassoc[geometry=-|-, arg1=arg1, mult1=
    mult1, pos1=0.1, align1=left , arg2=arg
    2, mult2=mult2, pos2=2.9, align2=right
    ]{A1}{A2}
\end{tikzpicture}
```



1.2.7 Définir et positionner le stéréotype d'une relation

Le stéréotype d'une relation est un mot clé contenu entre << et >>. On peut le définir à l'aide de l'option `stereo` et le positionner à l'aide de l'option `pos stereo`.

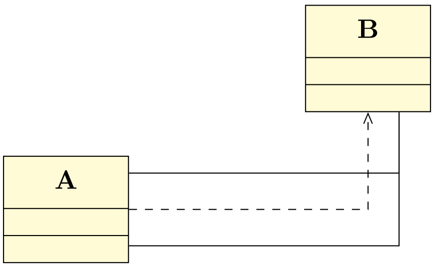
```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=-3]{A2}
\umlassoc[geometry=-|-,stereo=vector, pos
    stereo=1.2]{A1}{A2}
\end{tikzpicture}
```



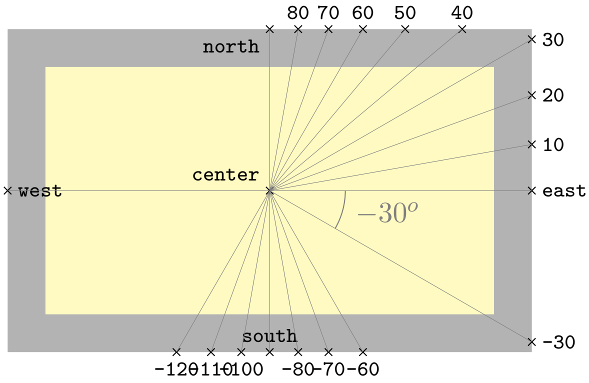
1.2.8 Modifier les points d'ancrage d'une relation

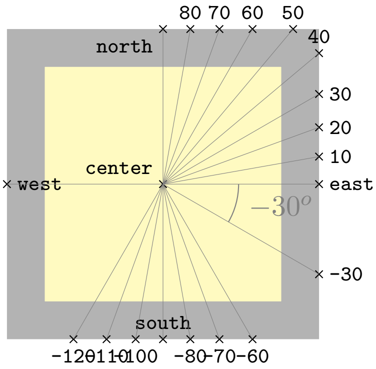
Le paragraphe qui vient concerne les relations dont la géométrie est à base de flèches segmentées. Par défaut, elles partent du centre du nœud de la classe d'origine et vont au centre du nœud de la classe cible. Il est possible d'ajuster ce comportement avec la paire d'options `anchor1`, `anchor2`.

```
\begin{tikzpicture}
\umlemptyclass{A}
\umlemptyclass[x=4,y=2]{B}
\umldep[geometry=-|]{A}{B}
\umlassoc[geometry=-|, anchor1=30, anchor2=300,
   name=assoc1]{A}{B}
\umlassoc[geometry=-|, anchor1=-30, anchor2=-60,
   name=assoc2]{A}{B}
\end{tikzpicture}
```



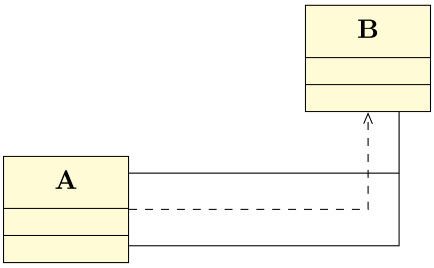
Les arguments que l'on donne sont des angles dont la valeur est en degré et peut être négative. Le mécanisme interne de la librairie TikZ effectue un modulo pour ramener ce nombre dans l'intervalle adéquat. La valeur 0 indique l'est, 90, indique le nord, 180 indique l'ouest, et 270 (ou -90) indique le sud du nœud. La figure ci-dessous illustre cette option et sa signification angulaire, sur 2 exemples de nœud de type rectangle, comme c'est le cas pour les classes. À noter que les points d'ancrage frontières (pour prendre la terminologie TikZ) dépendent bien des dimensions du nœud.





Il arrive finalement très souvent que l'on définisse les deux options `anchor1` et `anchor2` simultanément. On peut donc utiliser une forme contractée : l'option `anchors` qui s'utilise de la manière suivante, en reprenant l'exemple précédent :

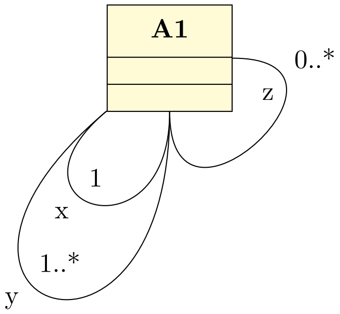
```
\begin{tikzpicture}
\umlemptyclass{A}
\umlemptyclass[x=4,y=2]{B}
\umldep[geometry=-|]{A}{B}
\umlassoc[geometry=-|, anchors=30 and 300, name=
    assoc1]{A}{B}
\umlassoc[geometry=-|, anchors=-30 and -60, name=
    assoc2]{A}{B}
\end{tikzpicture}
```



1.2.9 Définir une relation récursive

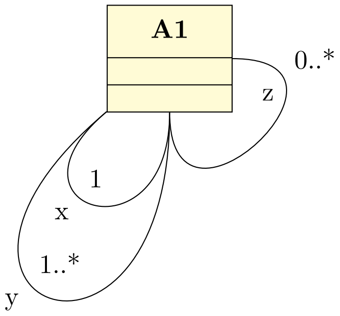
Il est possible de définir une relation récursive, c'est-à-dire une relation d'une classe à elle-même. Dans ce cas, l'option `geometry` doit être ignorée, mais 3 options deviennent très utiles : `angle1` détermine l'angle de départ, `angle2` détermine l'angle d'arrivée, et `loopsize` donne une idée de la taille de la boucle.

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlassoc[arg=x,mult=1,pos=0.6, angle
    1=-90, angle2=-140, loopsize=2cm]{A1}{
    A1}
\umlassoc[arg=y,mult=1..*,pos=0.6, angle
    1=-90, angle2=-140, loopsize=4cm]{A1}{
    A1}
\umlassoc[arg=z,mult=0..*,pos=0.8, angle
    1=-90, angle2=0, loopsize=2cm]{A1}{A1}
\end{tikzpicture}
```



À l'utilisation, il s'avère que l'on utilise très souvent les 3 options en même temps. C'est la raison pour laquelle il existe une forme contractée, l'option `recursive` qui s'utilise de la manière suivante :

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlassoc[arg=x, mult=1, pos=0.6,
    recursive=-90|-140|2cm]{A1}{A1}
\umlassoc[arg=y, mult=1..*, pos=0.6,
    recursive=-90|-140|4cm]{A1}{A1}
\umlassoc[arg=z, mult=0..*, pos=0.8,
    recursive=-90|0|2cm]{A1}{A1}
\end{tikzpicture}
```



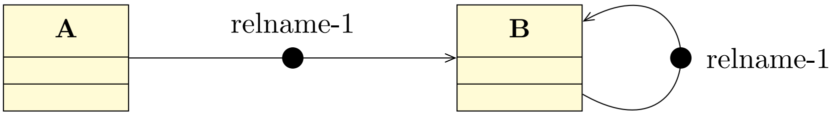
1.2.10 Nom des points de construction d'une relation

Pour voir l'importance de la possibilité de donner un nom à une relation et son utilité, il nous faut d'abord ici répondre à la question technique suivante : comment les flèches sont-elles concrètement définies ?

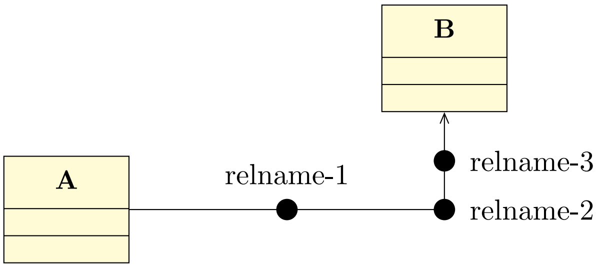
Pour construire une flèche, on a besoin de définir des nœuds de contrôle, auquel il faut donner un nom. Le seul moyen d'identifier de manière unique une relation est de lui donner un identifiant à travers un compteur que l'on va incrémenter. Supposons que notre relation a pour identifiant *i*. Le nom de la relation que l'on appellera dans ce qui suit *relname* est alors initialisé à : relation-*i*

Le premier nœud défini est le milieu de la relation, il s'appelle *relname-middle*. Je ne parlerai pas ici du placement adéquat de l'argument et de sa multiplicité dans un souci de simplification. Il y a donc 3 possibilités :

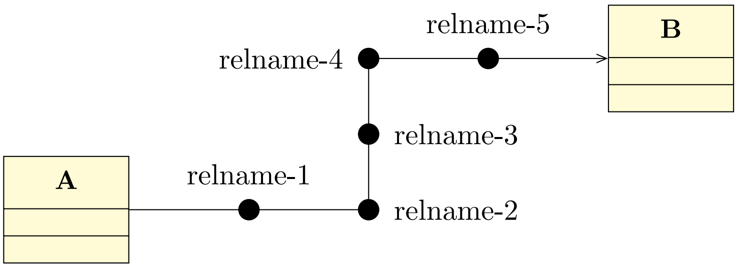
1. Si la flèche est une ligne continue (droite ou récursive), il est renommé en *relname-1*.



2. Si la flèche a un seul angle droit, alors l'unique nœud correspondant à l'angle droit est nommé *relname-2*, ce qui suffit à tracer la flèche. On définit par ailleurs les milieux des 2 arêtes constituant la flèche, nommés respectivement *relname-1* et *relname-3*.



3. Si la flèche à deux angles droits, ceux-ci sont définis de manière unique à l'aide de *relname-middle*, ce qui suffit à tracer la flèche. On nomme les nœuds aux angles droits respectivement *relname-2* et *relname-4*. On définit alors les milieux des 3 arêtes constituant la flèche, nommés respectivement *relname-1*, *relname-3*, et *relname-5*.

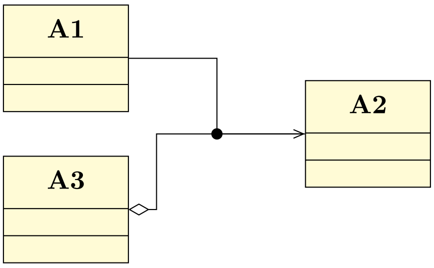


Il faut toutefois reconnaître que la définition par défaut de *relname* est non seulement peu pratique puisqu'on n'a pas vraiment accès à la numérotation, mais aussi fortement sujette à l'ordre dans lequel on définit les relations, donc peu portable en cas de modification de diagrammes. Pour simplifier cela, on va pouvoir définir *relname* à l'aide de l'option `name`. Et la raison pour laquelle cette option existe va être expliquée dans les 2 sections suivantes.

1.2.11 Tracer un point à une intersection de relations

Il arrive que dans un diagramme des relations se croisent et se chevauchent. Prenons 2 flèches qui se croisent. Est-ce que les 2 points de départ peuvent aller aux deux points d'arrivée ? Si oui, alors on va vouloir tracer un point à l'intersection des deux flèches, qui vraisemblablement va être un des nœuds de contrôle définis sur la relation. On utilisera pour cela la commande `umlpoint`.

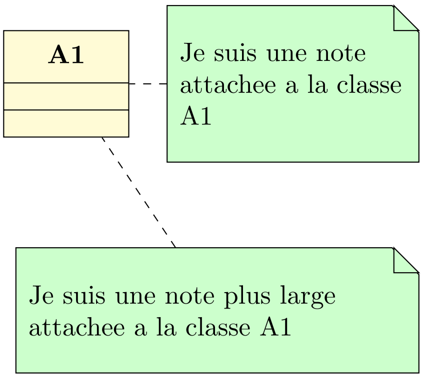
```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlemptyclass[x=4, y=-1]{A2}
\umlemptyclass[y=-2]{A3}
\umlassoc[geometry=-|-, name=assoc]{A1}{A2}
\umluniaggreg[geometry=-|-, weight=0.3]{A3}{A2}
\umlpoint{assoc-4}
\end{tikzpicture}
```



1.3 Note de commentaires / contraintes

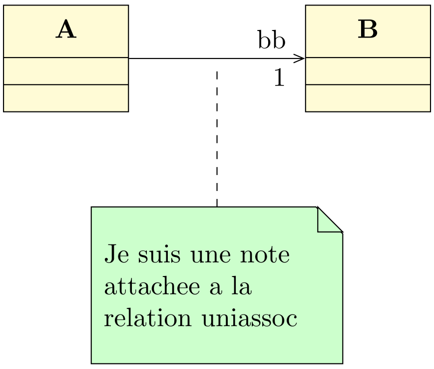
Une note est un commentaire de texte attaché à une classe ou une relation. La commande `umlnote` demande pour cela le nom du nœud auquel il faut se rattacher et le texte du commentaire en argument :

```
\begin{tikzpicture}
\umlemptyclass{A1}
\umlnote[x=3]{A1}{Je suis une note attachee a la classe A1}
\umlnote[x=2,y=-3, width=5cm]{A1}{Je suis une note plus large attachee a la classe A1}
\end{tikzpicture}
```



Là encore, on va pouvoir donner le nom d'un des points de contrôle d'une relation si l'on veut y attacher une note. Il faut donc pour cela pouvoir définir soi-même un nom simple à ces points :

```
\begin{tikzpicture}
\umlemptyclass{A}
\umlemptyclass[x=4]{B}
\umluniassoc[arg=bb, mult=1, pos=0.95, align=right, name=uniassoc]{A}{B}
\umlnote[x=2,y=-3]{uniassoc-1}{Je suis une note attachee a la relation uniassoc}
\end{tikzpicture}
```



Les notes sont utilisées pour 2 choses : les commentaires et la données de contraintes (généralement au format OCL).

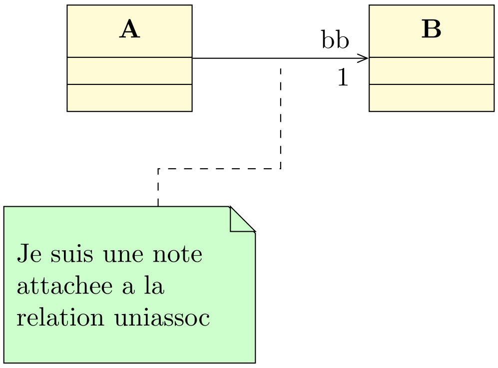
La commande `umlnote` dispose des options suivantes :

x, y Ces 2 options définissent les coordonnées de la note

width Cette option définit la largeur de la note. Pour les habitués de TikZ, elle encapsule l'option **text width**

weight, arm, anchor1, anchor2, anchors Ces options ont exactement le même rôle que pour la commande `umlrelation`, **arm** étant équivalent à **arm1**, c'est-à-dire attachée à la note source et pas à la cible.

```
\begin{tikzpicture}
\umlemptyclass{A}
\umlemptyclass[x=4]{B}
\umluniassoc[arg=bb, mult=1, pos=0.95, align=
   right , name=uniassoc]{A}{B}
\umlnote[y=-3, geometry=|-|, anchor1=70, arm=0.5
   cm]{uniassoc-1}{Je suis une note attachee a la
     relation uniassoc}
\end{tikzpicture}
```



Puisque l'option `geometry` est présente, de la même manière que pour `umlrelation`, ont été définis les alias `umlHVnote`, `umlVHnote`, `umlHVHnote` et `umlVHVnote`.

⚠ Pour chacun de ces 4 alias, l'option `geometry` est interdite.

⚠ Il n'y a pas d'alias dans le cas de la valeur - - pour la seule raison qu'il s'agit de la valeur par défaut.

1.4 Personnalisation

Grâce à la commande `tikzumlset`, il est possible de modifier l'apparence par défaut des packages, des classes et des notes. Les options que l'on peut personnaliser sont :

text : permet de spécifier la couleur du texte (=black par défaut),

draw : permet de spécifier la couleur des traits (=black par défaut),

fill class : permet de spécifier la couleur de fond des classes (=yellow!20 par défaut),

fill template : permet de spécifier la couleur de fond des boites templates (=yellow!2 par défaut),

fill package : permet de spécifier la couleur de fond des packages (=blue!20 par défaut),

fill subpackage : permet de spécifier la couleur de fond des sous-packages (=blue!20 par défaut),

fill note : permet de spécifier la couleur de fond des notes (=green!20 par défaut),

font : permet de définir le style de fonte du texte contenu dans tous les éléments d'un diagramme (=\small par défaut).

Par ailleurs, les commandes de relation disposent toutes d'une option `style` prenant en argument un nom de style au sens de TikZ.

Regardons l'exemple de la définition de la commande `umlinherit` :

```
\tikzstyle{tikzuml inherit style}=[color=\tikzumldrawcolor, -open triangle 45]
\newcommand{\umlinherit}[3][]{\umlrelation[style={tikzuml inherit style},
   #1]{#2}{#3}}
```

Vous pouvez donc très facilement définir une commande sur le même modèle en définissant un style particulier.

1.5 Exemples

1.5.1 Exemple de l'introduction, pas à pas

On va construire petit à petit l'exemple illustrant la première page de ce document afin de mettre en valeur les différentes commandes utilisées.

Définition des packages p, sp1 et sp2

On laisse le package p aux coordonnées (0,0) (comportement par défaut), et on place le sous-package sp1 aux coordonnées (0,0) et le sous-package sp2 aux coordonnées (10,-6).

```
\begin{tikzpicture}
\begin{umlpackage}{p}
  \umlvirt{setB(b : B) : void} \\ getB() : B}
\end{umlpackage}
}{}
  }{}
```

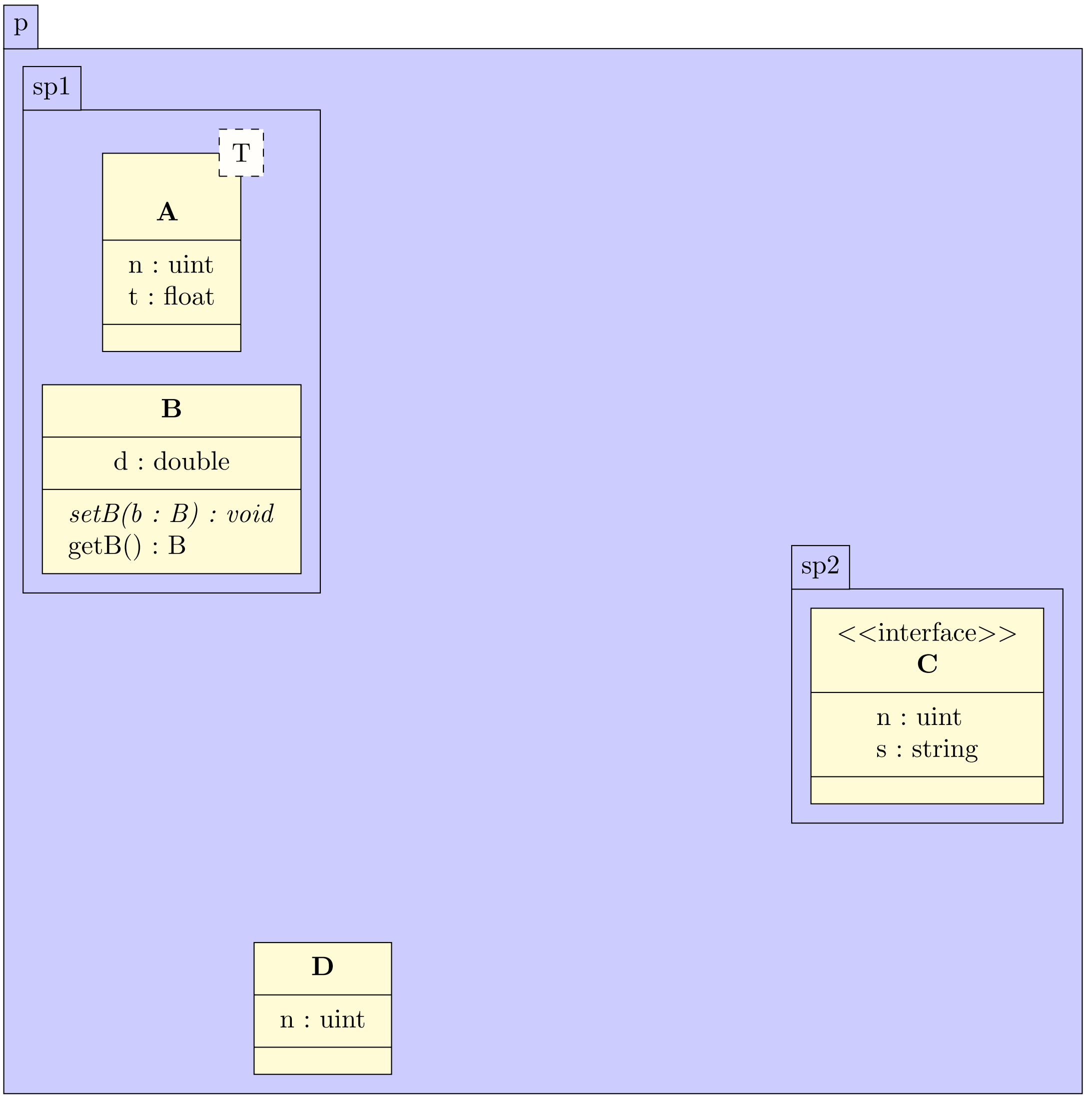
Définition des classes A, B, C, D et de leurs attributs et opérations

La classe A est en (0,0) dans le sous-package sp1 et a un paramètre template : T. La classe B est positionnée 3 unités en dessous de A, toujours dans le sous-package sp1, et possède un attribut statique et une opération virtuelle. La classe C est une interface en (0,0) dans le sous-package sp2. La classe D est placée en (2,-11) dans le package p.

La classe A a deux attributs. La classe B a un attribut et deux opérations dont une virtuelle. La classe C a deux attributs. La classe D a un attribut.

```
\begin{tikzpicture}
\begin{umlpackage}{p}
\begin{umlpackage}{sp1}
\umlclass[template=T]{A}{
  n : uint \\ t : float
}{}
\umlclass[y=-3]{B}{
  d : double
}{
  \umlvirt{setB(b : B) : void} \\ getB() : B}
\end{umlpackage}
\begin{umlpackage}[x=10,y=-6]{sp2}
\umlinterface{C}{
  n : uint \\ s : string
```

```
}{}
\end{umlpackage}
\umlclass[x=2,y=-10]{D}{
  n : uint
  }{}
```

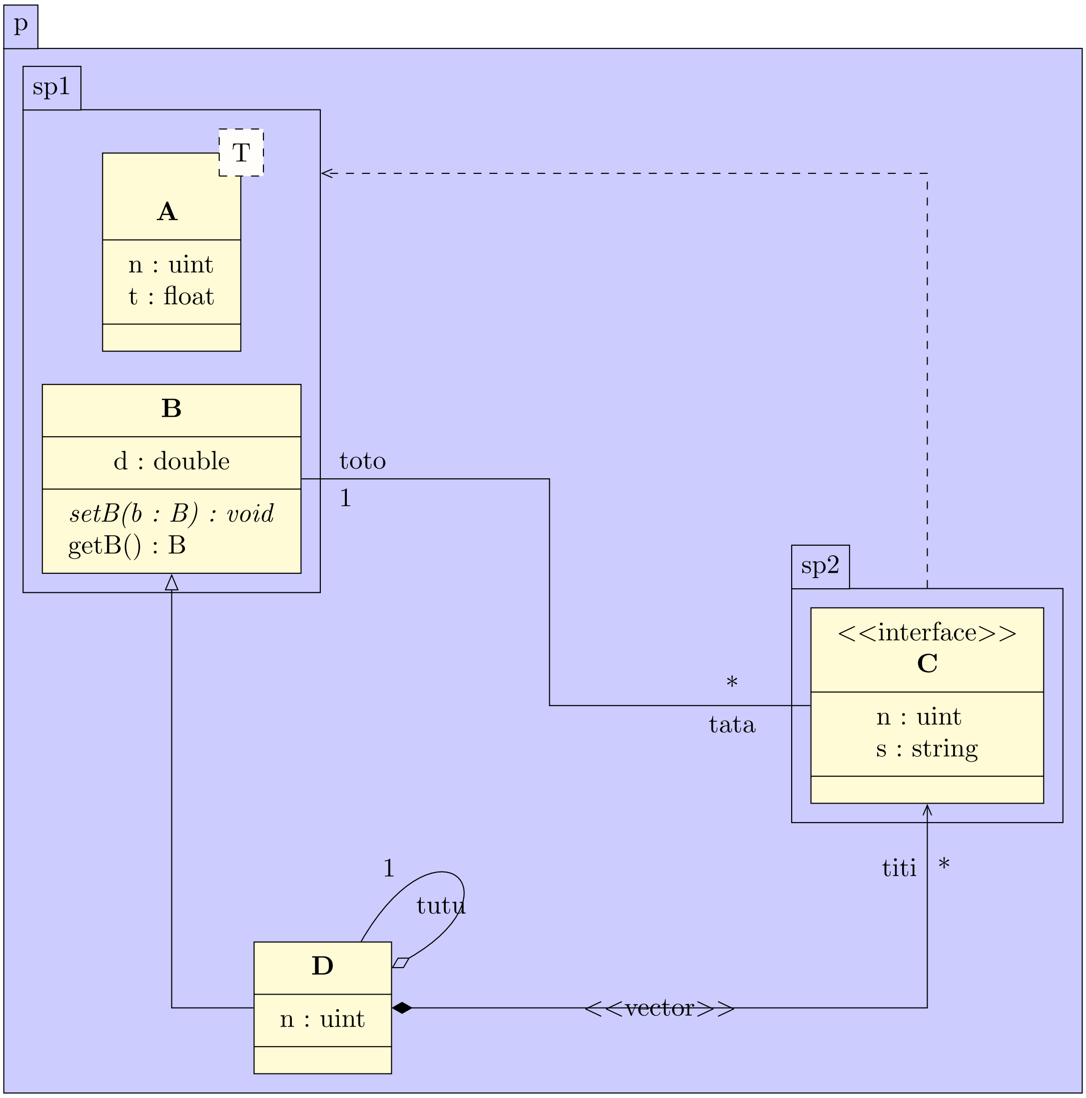


Définition des relations

On définit une association entre les classes C et B, une composition unidirectionnelle entre les classes D et C, une relation d'import nommée "import" entre les sous-packages sp2 et sp1 (avec modification des ancres), une relation d'agrégation récursive sur la classe D et une relation d'héritage entre les classes D et B. Sur ces relations, on va spécifier des noms d'arguments et leurs multiplicités. Regardez la valeur donnée à la position de ces éléments suivant la géométrie de la flèche.

...

```
\end{umlpackage}

\umlassoc[geometry=-|-, arg1=tata, mult1=*, pos1=0.3, arg2=toto, mult2=1, pos
    2=2.9, align2=left]{C}{B}
\umlunicompo[geometry=-|, arg=titi, mult=*, pos=1.7, stereo=vector]{D}{C}
\umlimport[geometry=|-, anchors=90 and 50, name=import]{sp2}{sp1}
```

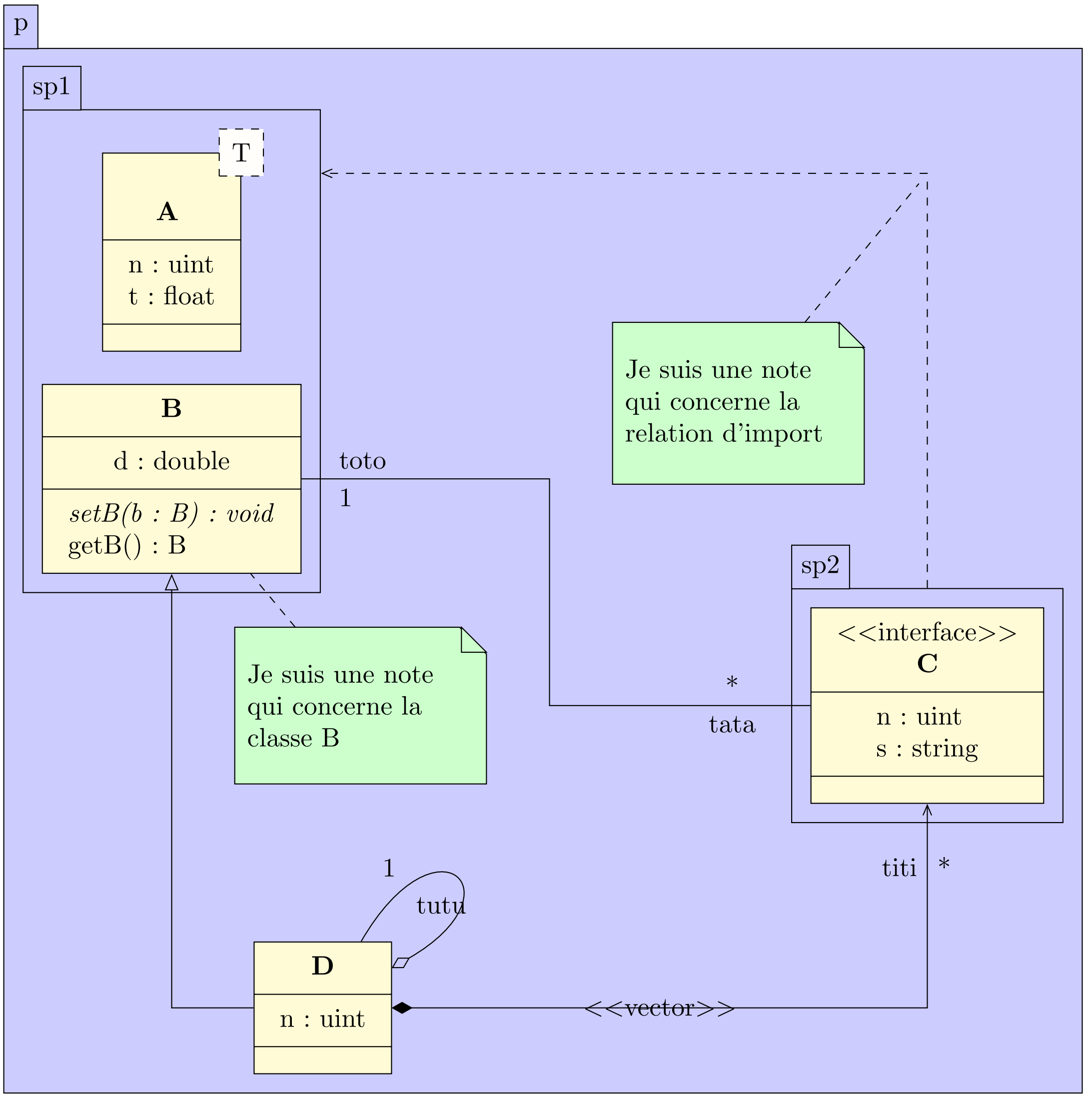


Définition des notes

On rajoute enfin une note attachée à la classe B et une note attachée à la relation d'import affectée du nom import.

...

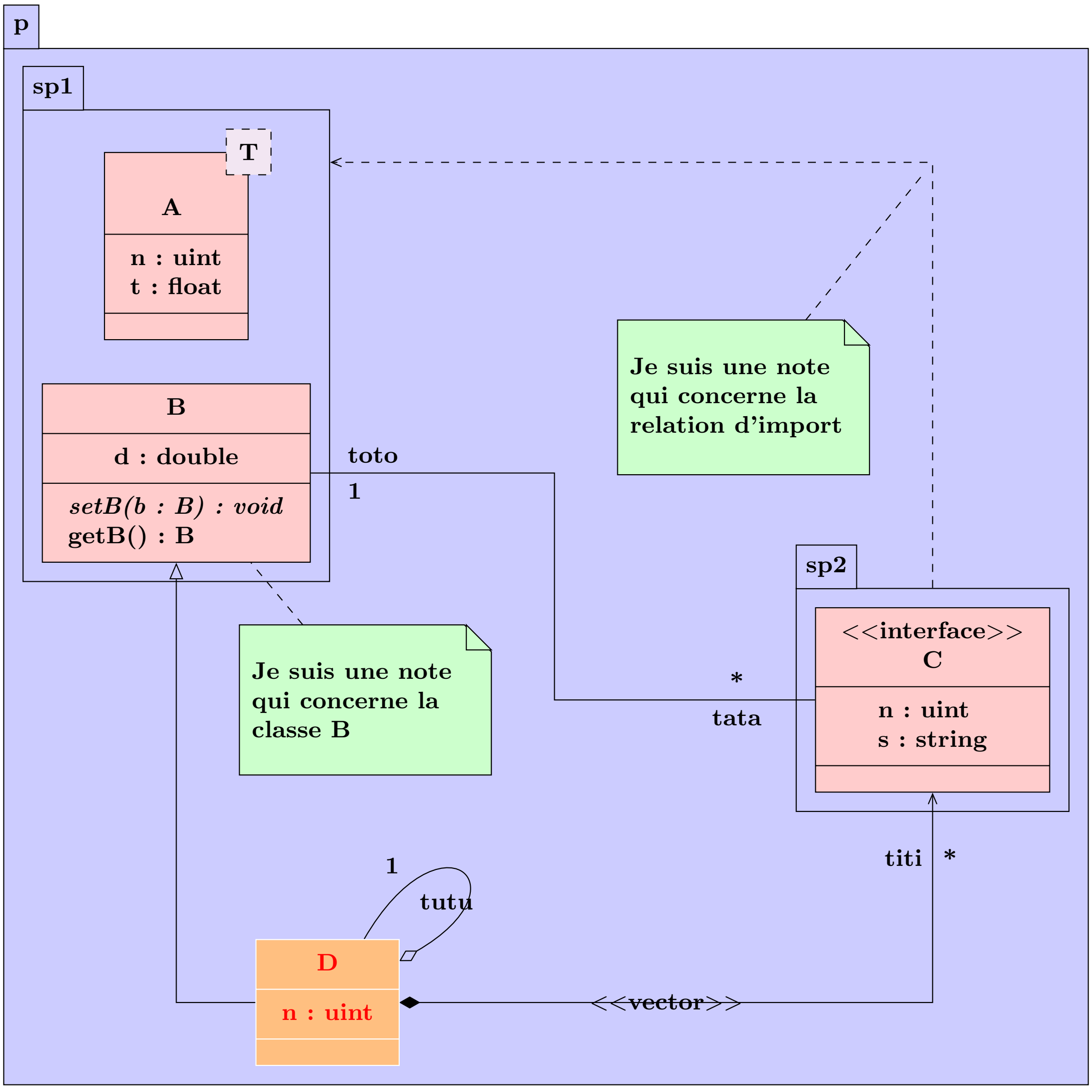
```
\umlaggreg[arg=tutu, mult=1, pos=0.8, angle1=30, angle2=60, loopsize=2cm]{D}{D}
\umlinherit[geometry=-|]{D}{B}
\umlnote[x=2.5,y=-6, width=3cm]{B}{Je suis une note qui concerne la classe B}
\umlnote[x=7.5,y=-2]{import-2}{Je suis une note qui concerne la relation d'import}
\end{tikzpicture}
```



Modification du style

On illustre l'utilisation de la commande `tikzumlset` en changeant les couleurs associées à la classe et le type de font. On peut par ailleurs modifier les couleurs d'une classe donnée avec les options `draw`, `text` et `fill`

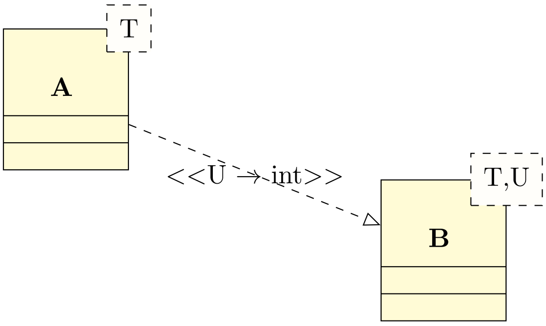
```
\tikzumlset{fill class=red!20, fill template=violet!10, font=\bfseries\
    footnotesize}
\begin{tikzpicture}
...
\umlclass[x=2,y=-11, fill=orange!50, draw=white, text=red]{D}{
  n : uint
  }{}
...
\end{tikzpicture}
```



1.5.2 Définir une spécialisation de classe

Une spécialisation de classe est de l'héritage d'un patron de classe dans lequel l'un des paramètres à son type fixé. Pour définir cette relation, c'est la commande `umlreal` qui va servir ici, ainsi que l'option `stereo` :

```
\begin{tikzpicture}
\umlemptyclass[template=T]{A}
\umlemptyclass[template={T,U}, x=5, y
    =-2]{B}
\umlreal[stereo={U $\rightarrow$ int}]{A
    }{B}
\end{tikzpicture}
```

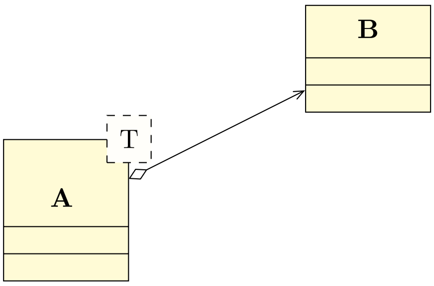


1.6 Règles de priorité des options et bugs identifiés

1. L'option `geometry` prime toujours sur les autres arguments. Cela signifie en particulier que si elle n'a pas sa valeur par défaut (– –), alors les options `angle1`, `angle2` et `loopsize`, paramétrant les relations récursives, seront ignorées.

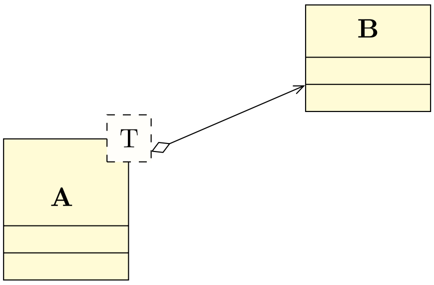
2. Dans le cas d'un patron de classe, il existe des cas où une relation la concernant sera mal définie, comme le montre le dessin ci-dessous, où le losange de la relation d'agrégation est caché par le paramètre template :

```
\begin{tikzpicture}
\umlemptyclass[template=T]{A}
\umlemptyclass[x=4,y=2]{B}
\umluniaggreg{A}{B}
\end{tikzpicture}
```



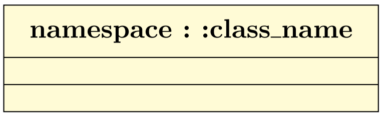
On peut toutefois corriger partiellement ce problème en reliant la flèche directement entre la partie template de la classe A et la classe B en rajoutant le suffixe -template et en ajustant l'ancre de départ (la valeur -30 est assez satisfaisante) :

```
\begin{tikzpicture}
\umlemptyclass[template=T]{A}
\umlemptyclass[x=4,y=2]{B}
\umluniaggreg[anchor1=-30]{A-template}{B}
\end{tikzpicture}
```



3. Si l'on définit une classe dont le nom comporte le caractère : – typiquement lorsqu'on précise le namespace – il peut y avoir un conflit avec l'option french (ou frenchb ou francais) du package babel. En effet, par défaut, ces options ajoutent systématiquement un espace devant le caractère : si le rédacteur du document l'a oublié, ce qui rentre en conflit avec la définition de l'opérateur d'accès ::. Si l'on reprend l'exemple de définition d'une classe, on obtiendrait :

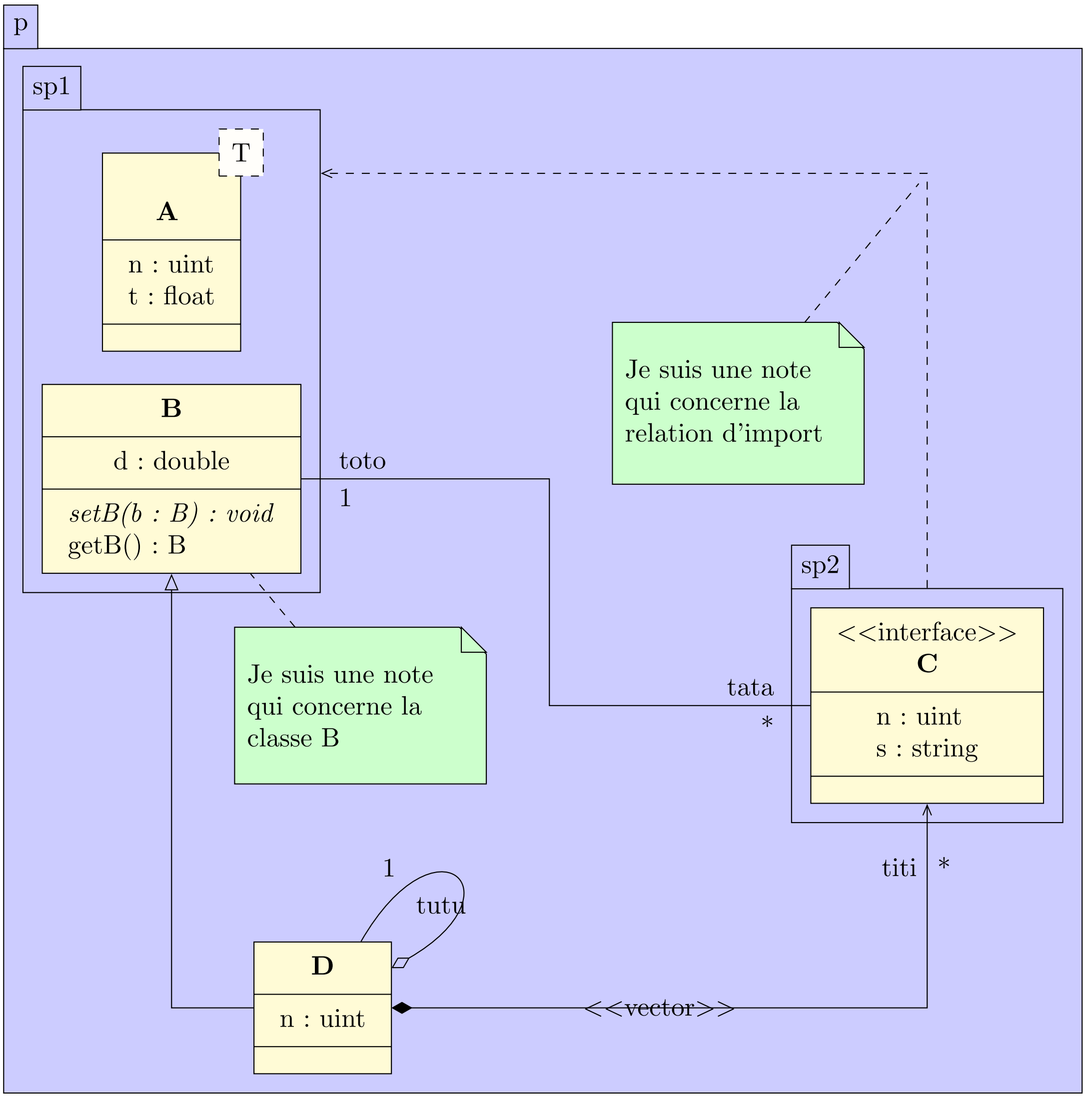
```
\begin{tikzpicture}
\umlclass[x=0,y=0]{namespace::class\_name}{}{}
\end{tikzpicture}
```



La solution consiste en une macro proposée par ces options du package babel, qu'il faut utiliser dans le préambule du document :

```
\frenchbsetup{AutoSpacePunctuation=false}
```

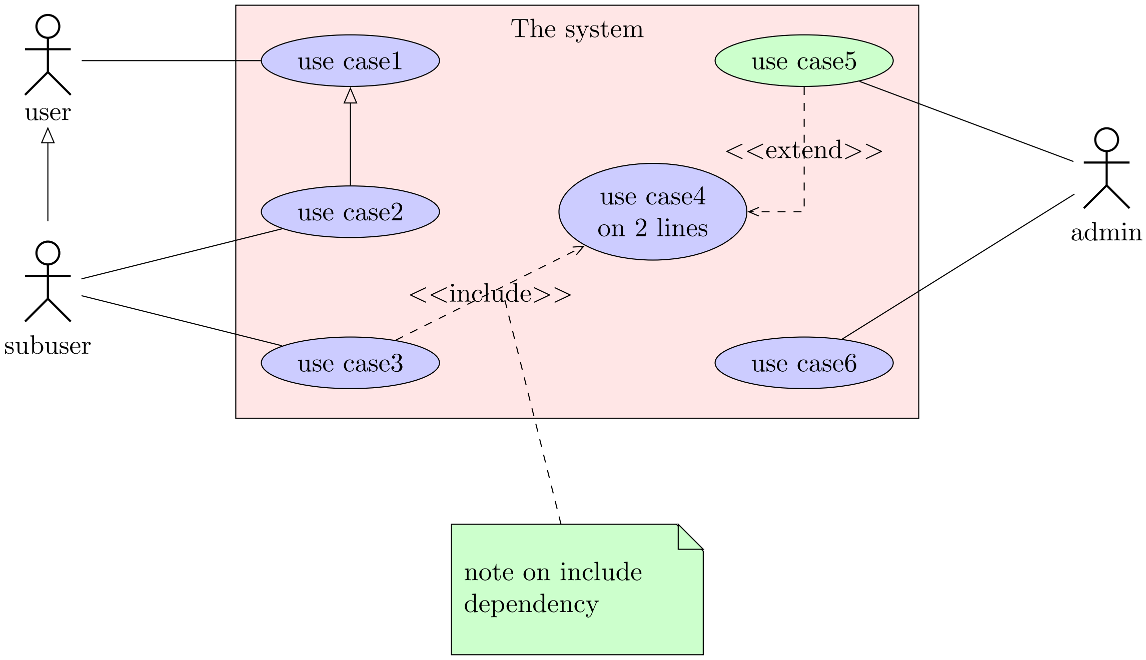
4. Le comportement de placement automatique des informations d'un attribut sur une relation peut surprendre quand on veut le court-circuiter. Reprenons l'exemple de l'introduction. Si l'on regarde la relation d'association et les attributs *toto* et *tata*. Si *toto* est au-dessus, *tata* est lui en dessous. Demandons maintenant de justifier à droite l'attribut *tata* (Au passage, on met sa position à 0.1). On constate alors que les positions de *tata* et de sa multiplicité s'inversent.


p
sp1
T
A
n : uint
t : float
B
d : double
setB(b : B) : void
getB() : B
toto
1
Je suis une note qui concerne la relation d'import
sp2
<<interface>>
C
n : uint
s : string
tata
*
Je suis une note qui concerne la classe B
1
tutu
titi
*
D
n : uint
<<vector>>

Chapitre 2

Diagrammes de cas d’utilisation

Voici un exemple de diagramme de cas d’utilisation que l’on peut réaliser :

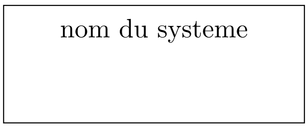


Nous allons voir successivement comment définir les 4 éléments constitutifs d’un tel diagramme : le système, les acteurs, les cas d’utilisation et les relations.

2.1 Définir un système

Un système est défini à l’aide de l’environnement `umlsystem` :

```
\begin{tikzpicture}
\begin{umlsystem}[x=0, y=0]{nom du systeme}

\end{umlsystem}
\end{tikzpicture}
```

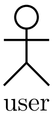


Les options `x` et `y` permettent de positionner le système dans la figure. Elles valent toutes deux 0 par défaut. A l’intérieur de cet environnement, on définira les cas d’utilisation, tandis qu’à l’extérieur, on définira les acteurs.

2.2 Définir un acteur

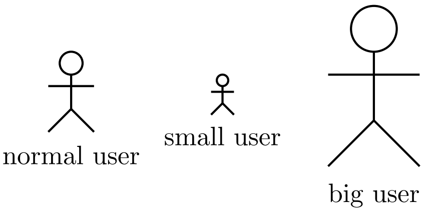
Un acteur est défini par la commande `umlactor` :

```
\begin{tikzpicture}
\umlactor[x=0, y=0]{user}
\end{tikzpicture}
```



Les options `x` et `y` permettent de positionner l'acteur dans la figure. Elles valent toutes deux 0 par défaut. On peut également redimensionner le symbole de l'acteur avec l'option `scale`. Cela adapte également la position du texte en dessous :

```
\begin{tikzpicture}
\umlactor{normal user}
\umlactor[x=2, scale=0.5]{small user}
\umlactor[x=4, scale=2]{big user}
\end{tikzpicture}
```



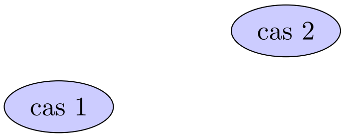
Le symbole de l'acteur est défini en unités relatives de la taille de fonte (unité en ex). Ce n'est par contre pas le cas de la distance séparant le symbole du texte en dessous. On peut alors corriger cela en fixant cette distance avec l'option `below` (0.5cm par défaut).

```
\tikzumlset{font=\tiny}
\begin{tikzpicture}
\umlactor{normal user}
\umlactor[x=2, scale=0.5, below=0.1cm]{
   small user}
\umlactor[x=4, scale=2]{big user}
\end{tikzpicture}
```

2.3 Définir un cas d'utilisation

Un cas d'utilisation est défini avec la commande `umlusecase` :

```
\begin{tikzpicture}
\umlusecase[x=0, y=0]{cas 1}
\umlusecase[x=3, y=1]{cas 2}
\end{tikzpicture}
```



Les options `x` et `y` permettent de positionner le cas d'utilisation dans la figure ou dans le système le contenant. Elles valent toutes deux 0 par défaut. Le texte passé en argument obligatoire est l'intitulé du cas d'utilisation. Le nœud le représentant possède un nom par défaut, basé sur un compteur global, de la forme usecase-17. Pour des raisons pratiques, on peut renommer un cas d'utilisation à l'aide de l'option `name`.

Par ailleurs, on peut fixer la taille du nœud avec l'option `width`.

Maintenant que nous avons vu tous les éléments constitutifs d'un diagramme de cas d'utilisation, nous allons pouvoir aborder les relations entre ces éléments.

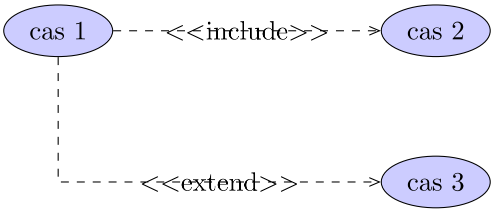
2.4 Définir une relation

Les relations dans un diagramme de cas d'utilisation sont de 4 natures :

- Des relations d'héritage, entre acteurs, ou entre cas d'utilisation. On utilisera pour cela la commande `umlinherit` et ses dérivées, abordées en sous-section 1.2.1.
- Des relations d'association, entre un acteur et un cas d'utilisation. On utilisera pour cela la commande `umlassoc` et ses dérivées, abordées en sous-section 1.2.1.
- Des relations d'inclusion ou d'extension. Graphiquement, ce sont des flèches de dépendance, au sens des diagrammes de classes, avec le stéréotype `extend` ou `include`. Des alias de la commande `umlrelation`, nommés `umlinclude`, `umlHVinclude`, . . . , `umlextend`, `umlHVextend`, . . . , sont disponibles pour définir de telles flèches.

Les options `anchor1`, `anchor2`, `anchors`, `arm1`, `arm2`, `weight`, `geometry` (uniquement pour `umlinclude` et `umlextend`), et `pos stereo` sont toujours utilisables.

```
\begin{tikzpicture}
\umlusecase[name=case1]{cas 1}
\umlusecase[x=5, name=case2]{cas 2}
\umlusecase[x=5, y=-2, name=case3]{cas 3}

\umlinclude{case1}{case2}
\umlVHextend[pos stereo=1.5]{case1}{case3}
\end{tikzpicture}
```



2.5 Personnalisation

Grace à la commande `tikzumlset`, on peut modifier globalement les couleurs par défaut des cas d'utilisation, systèmes, acteurs et relations :

text : permet de spécifier la couleur du texte (=black par défaut),

draw : permet de spécifier la couleur des traits (=black par défaut),

fill usecase : permet de spécifier la couleur de fond des cas d'utilisation (=blue!20 par défaut),

fill system : permet de spécifier la couleur de fond des systèmes (=white par défaut),

font : permet de spécifier le style de fonte du texte (=\small par défaut).

On peut également utiliser les options `text`, `draw` et `fill` sur un élément particulier pour lui modifier ses couleurs, comme illustré dans l'exemple d'introduction.

2.6 Exemples

2.6.1 Exemple de l'introduction, pas à pas

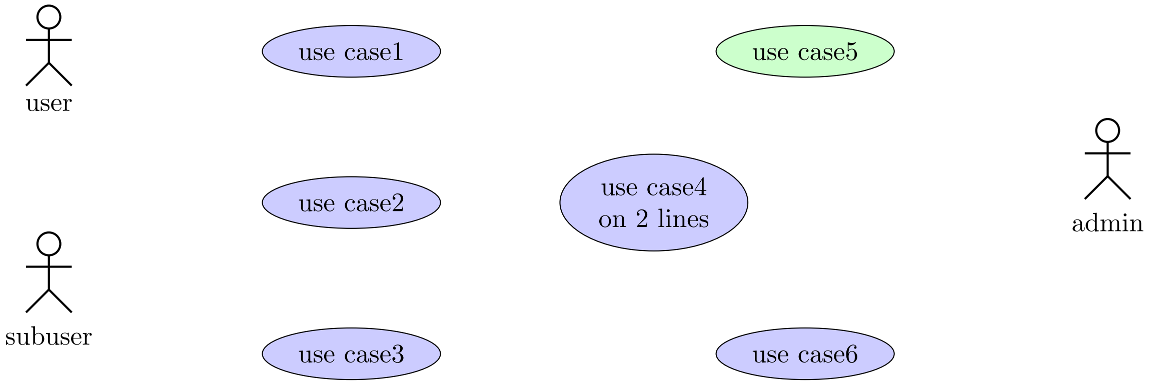
Définition des acteurs

```
\umlactor{user}
\umlactor[y=-3]{subuser}
\umlactor[x=14, y=-1.5]{admin}
```

Définition des cas d'utilisation

En plus de définir les cas d'utilisation, on illustre l'utilisation locale de l'option `fill`.

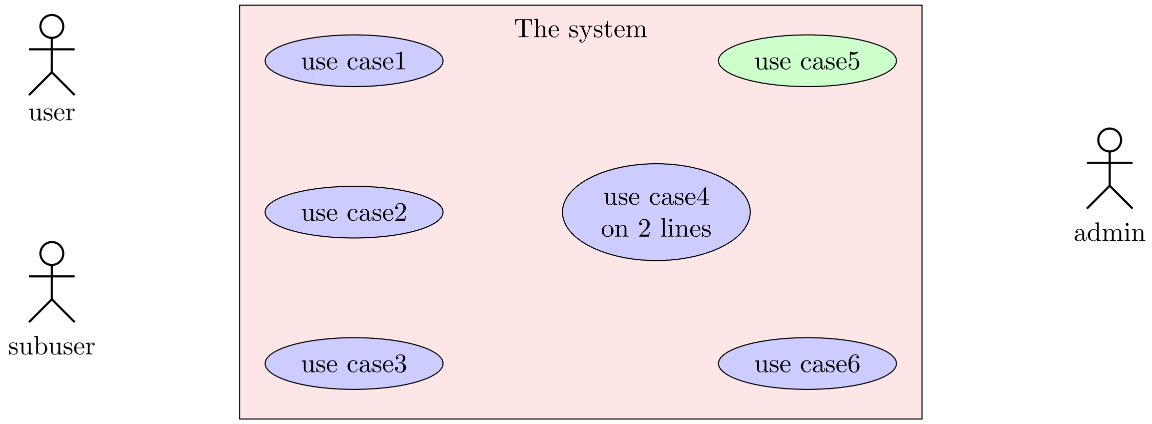
```
\umlusecase{use case1}
\umlusecase[y=-2]{use case2}
\umlusecase[y=-4]{use case3}
\umlusecase[x=4, y=-2, width=1.5cm]{use case4 on 2 lines}
\umlusecase[x=6, fill=green!20]{use case5}
\umlusecase[x=6, y=-4]{use case6}
\umlactor{user}
\umlactor[y=-3]{subuser}
\umlactor[x=14, y=-1.5]{admin}
```



Définition du système

Le système étant défini comme une boîte dont servant de nouveau repère de coordonnées, on va devoir modifier les coordonnées des cas d'utilisation relativement à celles du système.

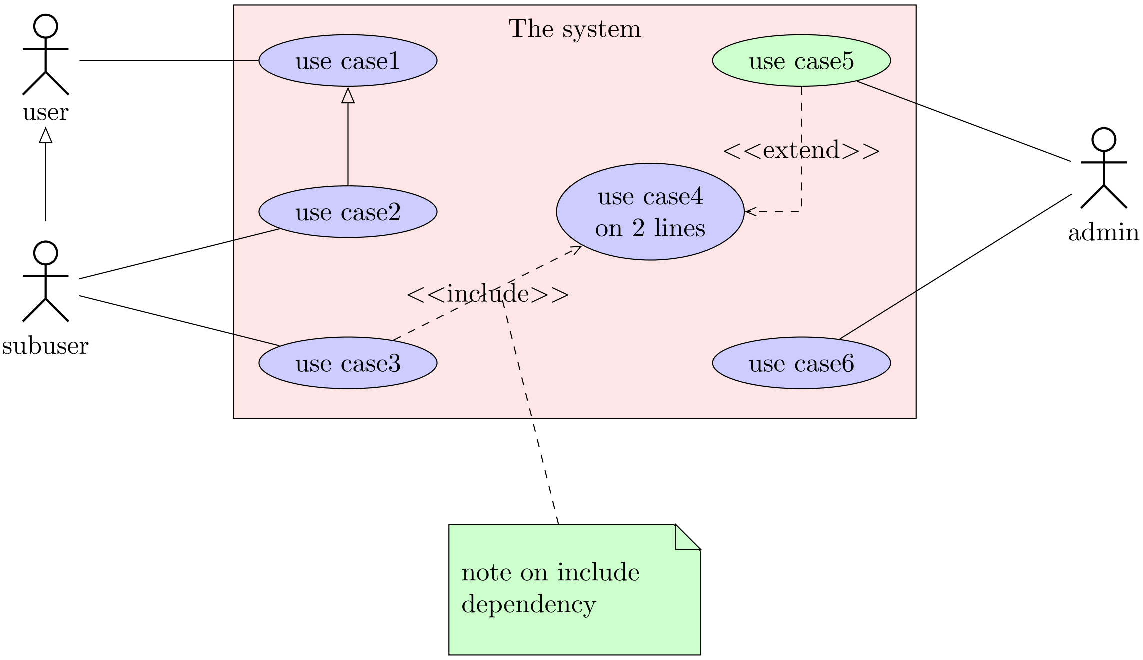
```
\begin{umlsystem}[x=4, fill=red!10]{The system}
\umlusecase{use case1}
\umlusecase[y=-2]{use case2}
\umlusecase[y=-4]{use case3}
\umlusecase[x=4, y=-2, width=1.5cm]{use case4 on 2 lines}
\umlusecase[x=6, fill=green!20]{use case5}
\umlusecase[x=6, y=-4]{use case6}
\end{umlsystem}

\umlactor{user}
\umlactor[y=-3]{subuser}
\umlactor[x=14, y=-1.5]{admin}
```



Définition des relations et de la note

On remarquera l'utilisation de l'option `name` pour s'assurer simplement de la bonne définition de la note, et son intérêt également pour les cas d'utilisation, afin de ne pas avoir à se rappeler de l'ordre dans lequel ils sont définis, comme c'est le cas dans cet exemple simple :

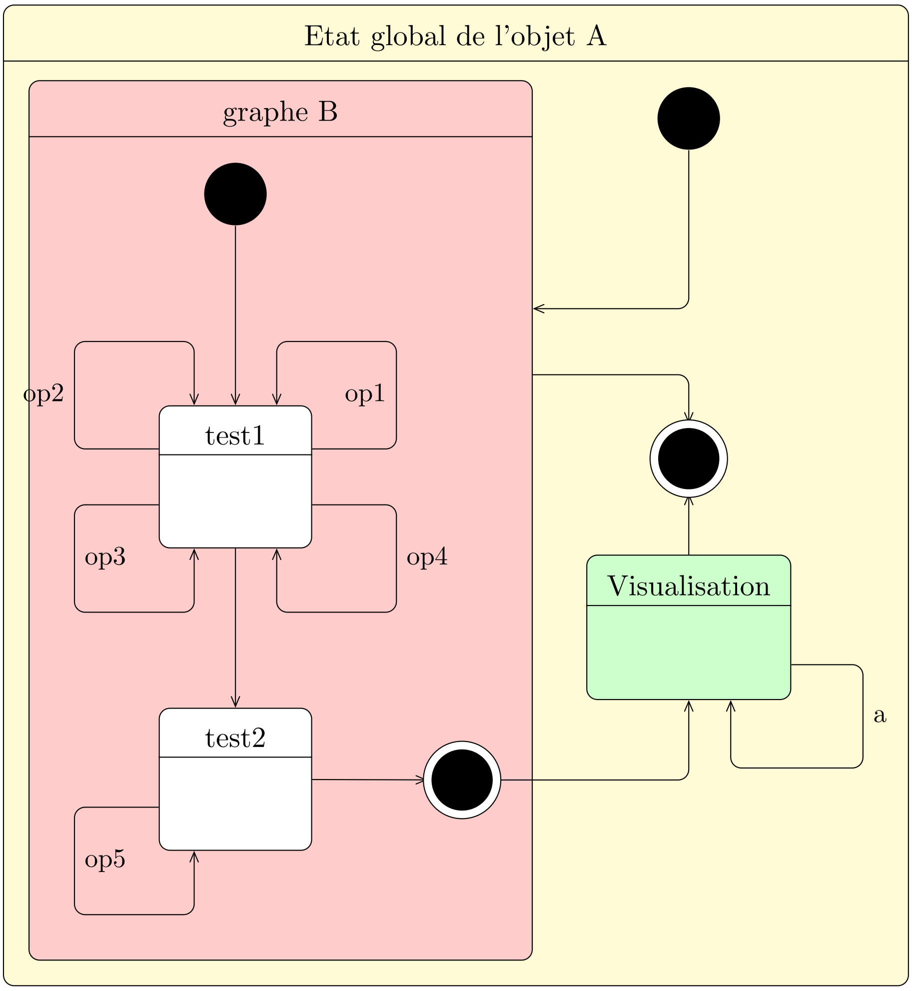
```
\begin{umlsystem}[x=4, fill=red!10]{The system}
\umlusecase{use case1}
\umlusecase[y=-2]{use case2}
\umlusecase[y=-4]{use case3}
\umlusecase[x=4, y=-2, width=1.5cm]{use case4 on 2 lines}
\umlusecase[x=6, fill=green!20]{use case5}
\umlusecase[x=6, y=-4]{use case6}
\end{umlsystem}

\umlactor{user}
\umlactor[y=-3]{subuser}
\umlactor[x=14, y=-1.5]{admin}

\umlinherit{subuser}{user}
\umlassoc{user}{usecase-1}
\umlassoc{subuser}{usecase-2}
\umlassoc{subuser}{usecase-3}
\umlassoc{admin}{usecase-5}
\umlassoc{admin}{usecase-6}
\umlinherit{usecase-2}{usecase-1}
\umlVHextend{usecase-5}{usecase-4}
\umlinclude[name=incl]{usecase-3}{usecase-4}

\umlnote[x=7, y=-7]{incl-1}{note on include dependency}
```


The system
user
use case1
use case5
<<extend>>
use case4
on 2 lines
admin
use case2
subuser
<<include>>
use case3
use case6
note on include
dependency

Chapitre 3

Diagrammes d'états-transitions

Voici un exemple de diagramme d'états-transitions que l'on peut réaliser :

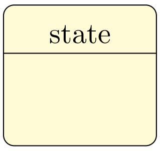


Nous allons voir successivement comment définir les éléments constitutifs d'un tel diagramme : les 10 types d'états et les transitions.

3.1 Définir un état

Un état « standard » se définit à l'aide de l'environnement `umlstate` :

```
\begin{tikzpicture}
\begin{umlstate}[x=0, y=0, name=state]{
  state}

\end{umlstate}
\end{tikzpicture}
```

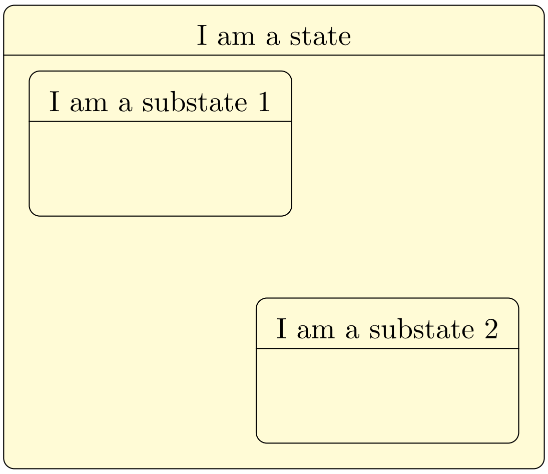


Les options `x` et `y` permettent de positionner l'état dans la figure, ou dans un autre état. Elles valent toutes deux 0 par défaut. L'argument est le label de l'état. Le nœud représentatnt l'état a par défaut un nom basé sur un compteur. Pour des raisons pratiques, lors de la définition de transitions par exemple, on peut le renommer à l'aide de l'option `name`.

On peut également définir la largeur minimale d'un état avec l'option `width` (10ex par défaut).

On peut définir un état dans un autre état. Dans ce cas, les coordonnées des sous-états sont relatives à l'état parent :

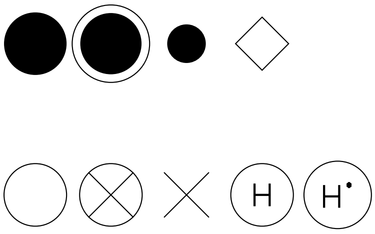
```
\begin{tikzpicture}
\begin{umlstate}[name=state]{I am a state}
\begin{umlstate}[name=substate1]{I am a
  substate 1}

\end{umlstate}
\begin{umlstate}[x=3, y=-3, name=substate
  2]{I am a substate 2}

\end{umlstate}
\end{umlstate}
\end{tikzpicture}
```



Si l'on veut définir un état sans le décomposer, on peut utiliser la commande `umlbasicstate` qui est un raccourci vers l'environnement `umlstate`.

Regardons maintenant les états spécifiques :

```
\begin{tikzpicture}
\umlstateinitial[name=initial]
\umlstatefinal[x=1, name=final]
\umlstatejoin[x=2, name=join]
\umlstatedecision[x=3, name=decision]
\umlstateenter[y=-2, name=enter]
\umlstateexit[x=1, y=-2, name=exit]
\umlstateend[x=2, y=-2, name=end]
\umlstatehistory[x=3, y=-2, name=hist]
\umlstatedeephistory[x=4, y=-2, name=
  deephist]
\end{tikzpicture}
```



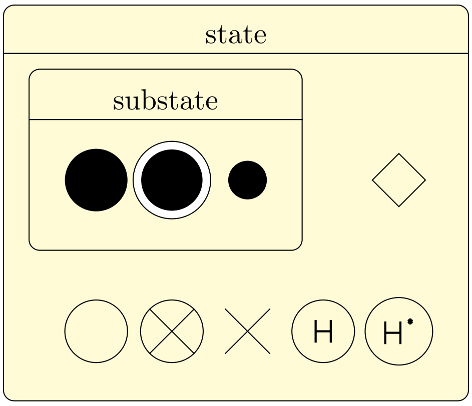
De gauche à droite et de haut en bas :

- Un état initial se définit à l'aide de la commande `umlstateinitial`.
- Un état final se définit à l'aide de la commande `umlstatefinal`.
- Un état de jonction se définit à l'aide de la commande `umlstatejoin`.

- Un état de décision se définit à l'aide de la commande `umlstatedecision`.
- Un état d'entrée se définit à l'aide de la commande `umlstateenter`.
- Un état de sortie se définit à l'aide de la commande `umlstateexit`.
- Un état de fin se définit à l'aide de la commande `umlstateend`.
- Un état d'historique se définit à l'aide de la commande `umlstatehistory`.
- Un état d'historique profond se définit à l'aide de la commande `umlstatedeephistory`.

Ces commandes prennent toutes pour options `name`, pour renommer le nœud, et `width` pour fixer sa taille, et on peut les utiliser à l'intérieur d'un environnement `umlstate` :

```
\begin{tikzpicture}
\begin{umlstate}[name=state]{state}
\begin{umlstate}[name=substate]{substate}
\umlstateinitial[name=initial]
\umlstatefinal[x=1, name=final]
\umlstatejoin[x=2, name=join]
\end{umlstate}
\umlstatedecision[x=4, name=decision]
\umlstateenter[y=-2, name=enter]
\umlstateexit[x=1, y=-2, name=exit]
\umlstateend[x=2, y=-2, name=end]
\umlstatehistory[x=3, y=-2, name=hist]
\umlstatedeephistory[x=4, y=-2, name=
    deephist]
\end{umlstate}
\end{tikzpicture}
```



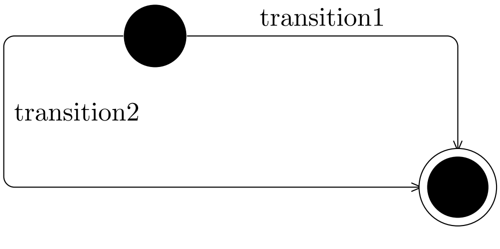
3.2 Définir une transition

Les transitions sont les noms des relations dans un diagramme d'états-transitions. On les définit à l'aide de la commande `umltrans` qui dérive de `umlrelation`. On va distinguer le cas des transitions unidirectionnelles et celui des transitions récursives.

3.2.1 Définir une transition unidirectionnelle

De par l'existence de l'option `geometry`, on va définir de même les alias `umlHVtrans`, `umlVHtrans`, `umlVHVtrans` et `umlHVHtrans`. Graphiquement, ce sont l'utilisation de ces alias qui sont les plus intéressants, dans la mesure où tous les coins d'une flèche de transition sont arrondis.

```
\begin{tikzpicture}
\umlstateinitial[name=initial]
\umlstatefinal[x=4, y=-2, name=final]
\umlHVtrans[arg=transition1, pos=0.5]{
    initial}{final}
\umlHVHtrans[arm1=-2cm, arg=transition2,
    pos=1.5]{initial}{final}
\end{tikzpicture}
```

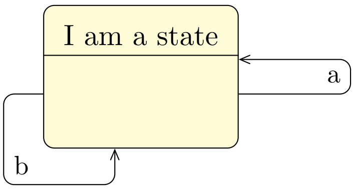


Toutes les options de `umlrelation` peuvent être utilisées avec `umltrans` et ses dérivées.

3.2.2 Définir une transition récursive

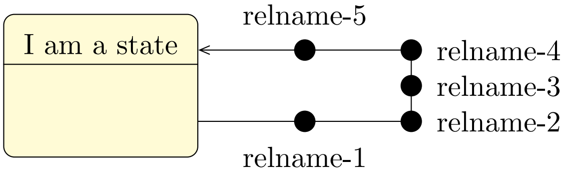
Les transitions récursives sont graphiquement les plus délicates à mettre en œuvre, car elles ont l'aspect de rectangles aux coins arrondis, contrairement aux flèches récursives des diagrammes de classes. Elles se comportent donc, sur le principe, comme les flèches dont l'option `geometry` est -|- ou |-|, c'est-à-dire des flèches composées de plusieurs segments.

```
\begin{tikzpicture}
\umlbasicstate[name=state]{I am a state}
\umltrans[recursive=-10|10|2cm, arg=a, pos
   =1.5, recursive direction=right to
   right]{state}{state}
\umltrans[recursive=-170|-110|2cm, arg=b,
   pos=2, recursive direction=left to
   bottom]{state}{state}
\end{tikzpicture}
```

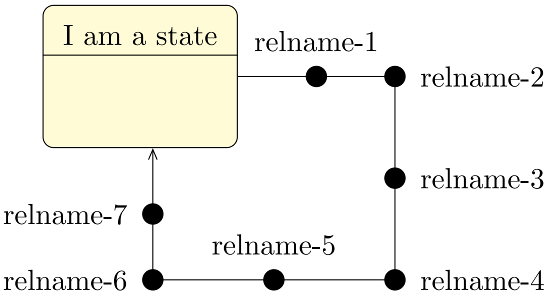


L'option `recursive direction` est fondamentale. En effet, la donnée des angles de départ et d'arrivée ne suffit pas à savoir la direction de départ et d'arrivée de la flèche récursive. On est donc obligé de le préciser. 2 cas se présentent :

- Soit la flèche est constituée de 3 segments. Dans ce cas, les nœuds utilisables sont répartis comme suit :



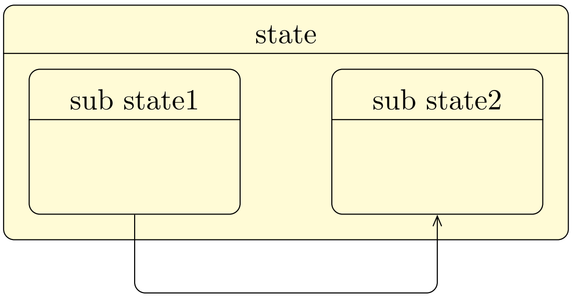
- Soit la flèche est constituée de 4 segments. Dans ce cas, les nœuds utilisables sont répartis comme suit :



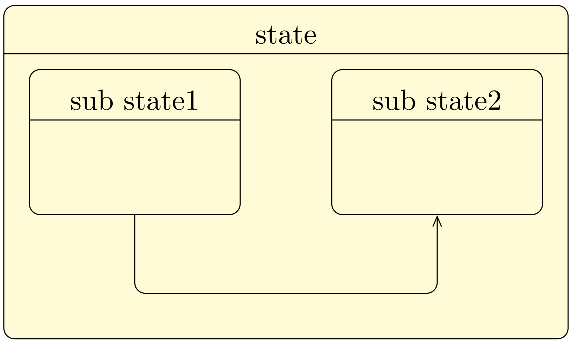
3.2.3 Définir une transition entre sous-états

Lorsqu'on définit des transitions entre sous-états, la transition doit être intégralement représentée à l'intérieur de l'état parent. C'est la raison pour laquelle on définira la transition à l'intérieur de l'environnement `umlstate`. Comparez les 2 séquences de codes suivantes :

```
\begin{tikzpicture}
\begin{umlstate}[name=state]{state}
\umlbasicstate[name=substate1]{sub state1}
\umlbasicstate[x=4, name=substate2]{sub
   state2}
\end{umlstate}

\umlVHVtrans[arm1=-2cm]{substate1}{
   substate2}
\end{tikzpicture}
```



```
\begin{tikzpicture}
\begin{umlstate}[name=state]{state}
\umlbasicstate[name=substate1]{sub state1}
\umlbasicstate[x=4, name=substate2]{sub
    state2}

\umlVHVtrans[arm1=-2cm]{substate1}{
    substate2}
\end{umlstate}
\end{tikzpicture}
```



3.3 Personnalisation

Grace à la commande `tikzumlset`, on peut modifier globalement les couleurs par défaut des états et transitions :

text : permet de spécifier la couleur du texte (=black par défaut),

draw : permet de spécifier la couleur des traits et des états initiaux, finaux et de jonction (=black par défaut),

fill state : permet de spécifier la couleur de fond des cas d'utilisation (=yellow!20 par défaut),

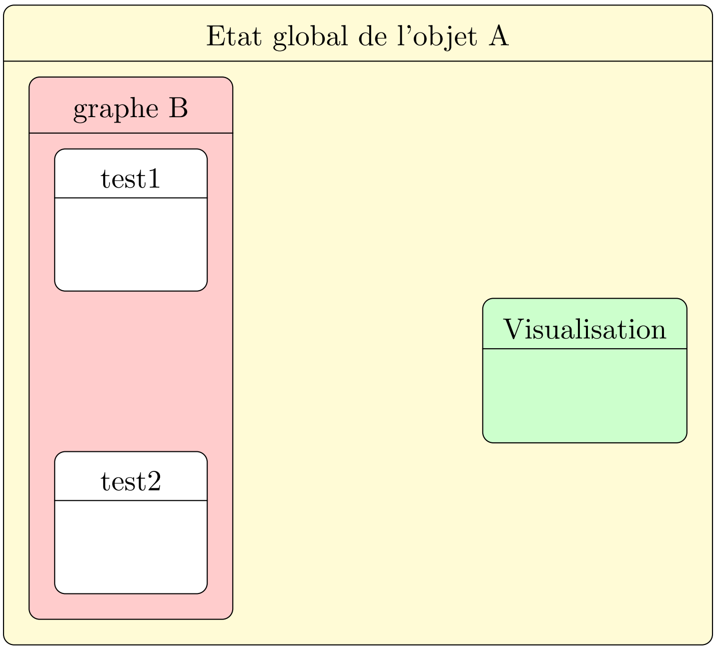
font : permet de spécifier le style de fonte du texte (=\small par défaut).

On peut également utiliser les options `text`, `draw` et `fill` sur un élément particulier pour lui modifier ses couleurs, comme illustré dans l'exemple d'introduction.

3.4 Exemples

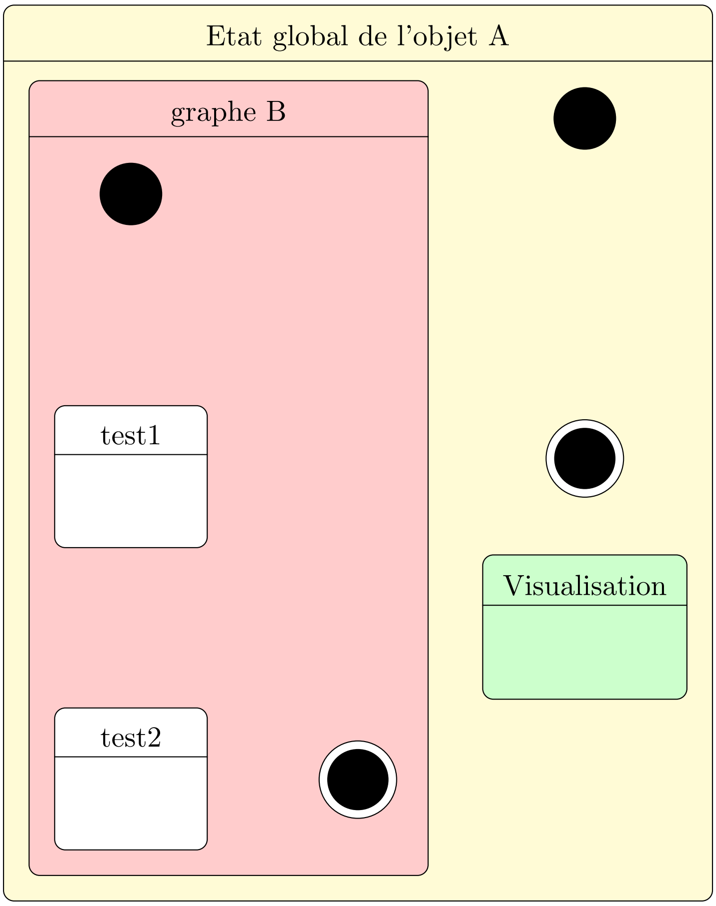
3.4.1 Exemple de l'introduction, pas à pas

Définition des états standards



```
\begin{umlstate}[name=Amain]{Etat global de l'objet A}
\begin{umlstate}[name=Bgraph, fill=red!20]{graphe B}
\umlbasicstate[y=-4, name=test1, fill=white]{test1}
\umlbasicstate[y=-8, name=test2, fill=white]{test2}
\end{umlstate}
\umlbasicstate[x=6, y=-6, name=visu, fill=green!20]{Visualisation}
\end{umlstate}
```

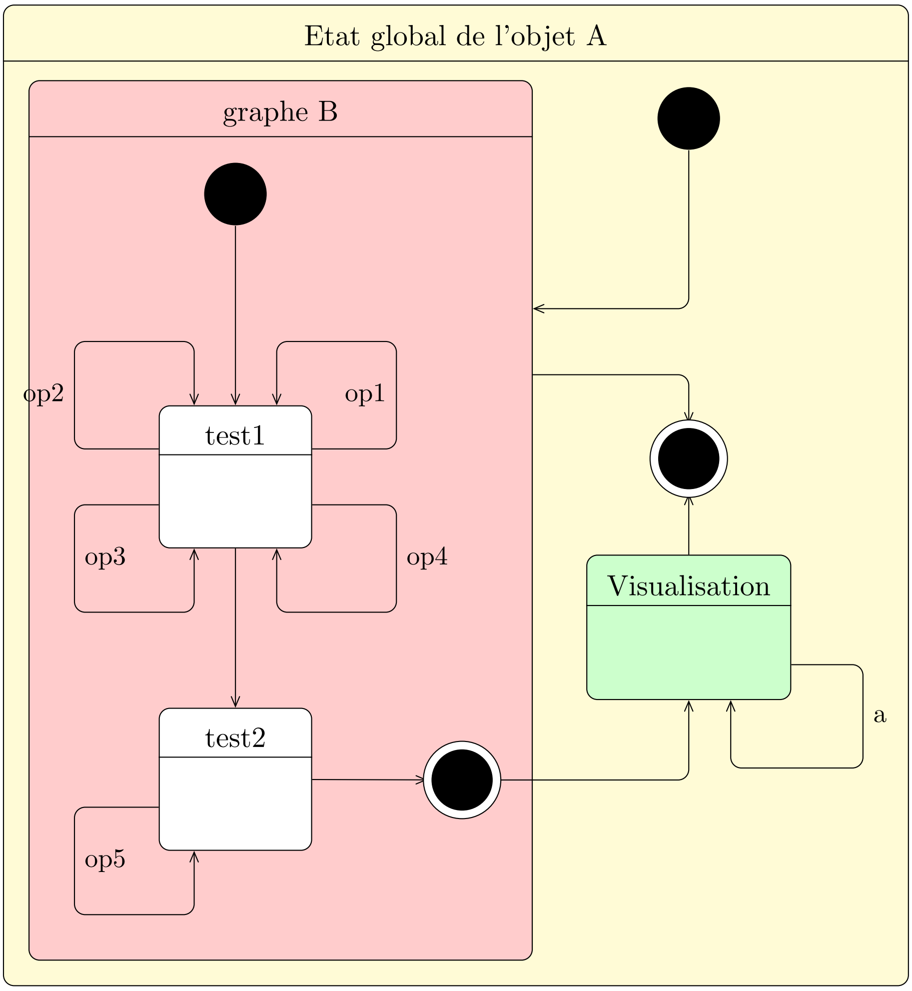
Définition des états spécifiques



```
\begin{umlstate}[name=Amain]{Etat global de l'objet A}
\begin{umlstate}[name=Bgraph, fill=red!20]{graphe B}
\umlstateinitial[name=Binit]
\umlbasicstate[y=-4, name=test1, fill=white]{test1}
\umlbasicstate[y=-8, name=test2, fill=white]{test2}
\umlstatefinal[x=3, y=-7.75, name=Bfinal]
\end{umlstate}
\umlstateinitial[x=6, y=1, name=Ainit]
\umlstatefinal[x=6, y=-3.5, name=Afinal]
\umlbasicstate[x=6, y=-6, name=visu, fill=green!20]{Visualisation}
\end{umlstate}
```

Définition des transitions

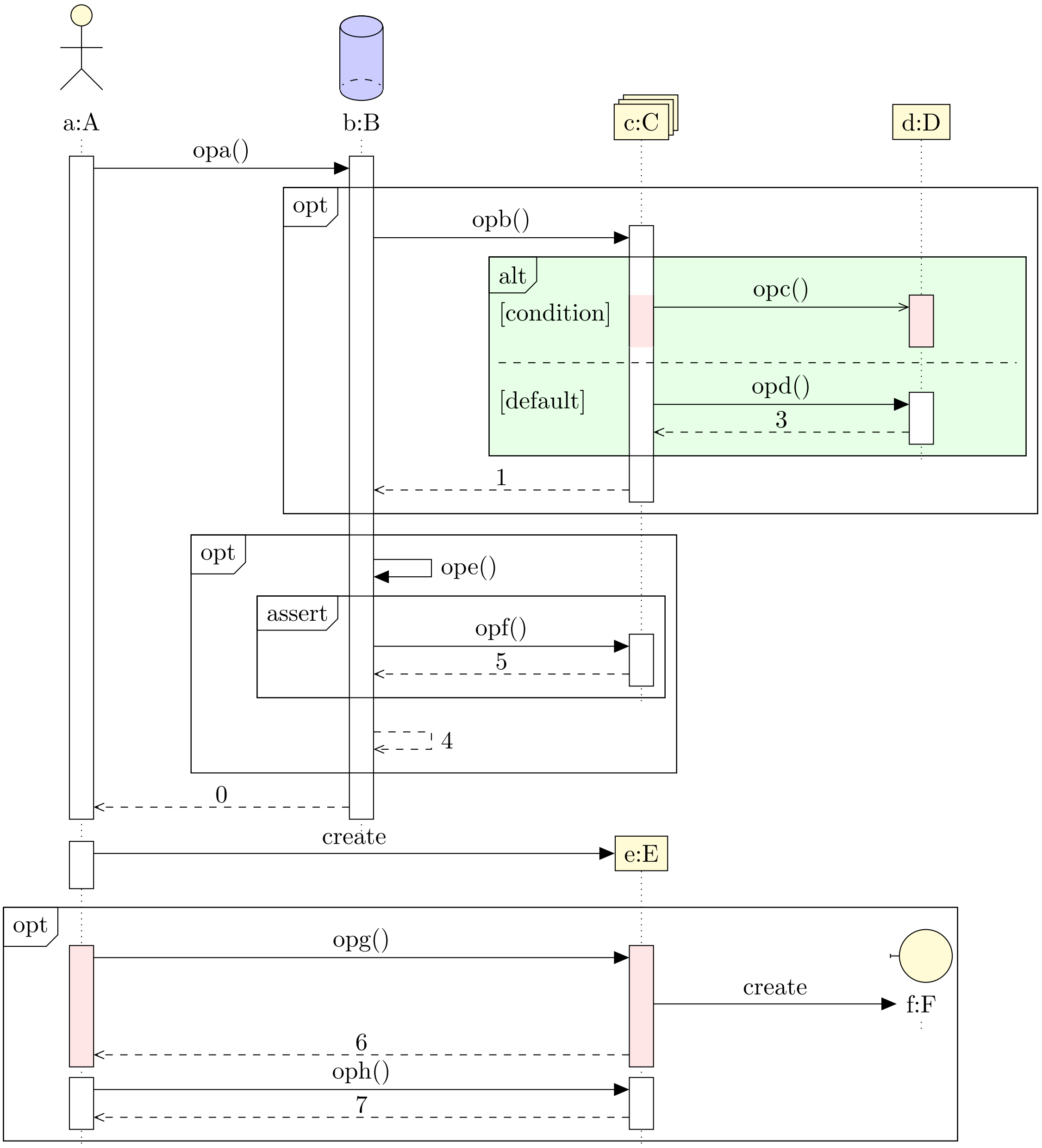
```
\begin{umlstate}[name=Amain]{Etat global de l'objet A}
\begin{umlstate}[name=Bgraph, fill=red!20]{graphe B}
\umlstateinitial[name=Binit]
\umlbasicstate[y=-4, name=test1, fill=white]{test1}
\umltrans{Binit}{test1}
\umltrans[recursive=20|60|2.5cm, recursive direction=right to top, arg={op1}, pos
   =1.5]{test1}{test1}
\umltrans[recursive=160|120|2.5cm, recursive direction=left to top, arg={op2}, pos
   =1.5]{test1}{test1}
\umltrans[recursive=-160|-120|2.5cm, recursive direction=left to bottom, arg={op
   3}, pos=1.5]{test1}{test1}
\umltrans[recursive=-20|-60|2.5cm, recursive direction=right to bottom, arg={op4},
    pos=1.5]{test1}{test1}
\umlbasicstate[y=-8, name=test2, fill=white]{test2}
\umltrans[recursive=-160|-120|2.5cm, recursive direction=left to bottom, arg={op
   5}, pos=1.5]{test2}{test2}
\umltrans{test1}{test2}
\umlstatefinal[x=3, y=-7.75, name=Bfinal]
\umltrans{test2}{Bfinal}
\end{umlstate}
\umlstateinitial[x=6, y=1, name=Ainit]
\umlVHtrans[anchor2=40]{Ainit}{Bgraph}
\umlstatefinal[x=6, y=-3.5, name=Afinal]
\umlHVtrans[anchor1=30]{Bgraph}{Afinal}
\umlbasicstate[x=6, y=-6, name=visu, fill=green!20]{Visualisation}
\umlHVtrans{Bfinal}{visu}
\umltrans{visu}{Afinal}
\umltrans[recursive=-20|-60|2.5cm, recursive direction=right to bottom, arg=a, pos
   =1.5]{visu}{visu}
\end{umlstate}
```


Etat global de l'objet A
graphe B
op2
op1
test1
op3
op4
Visualisation
a
test2
op5

Chapitre 4

Diagrammes de séquence

Voici un exemple de diagramme de séquence que l'on peut réaliser :



Regardons maintenant les différents éléments constitutifs d'un tel diagramme.

4.1 Définir un diagramme de séquence

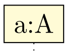
C'est là la principale différence avec les diagrammes précédents. Pour définir un diagramme de séquence, il faut travailler dans l'environnement `umlseqdiag`, dont le but est d'initialiser un ensemble de variables globales, et surtout de tracer les lignes de vies de chaque objet intervenant dans le diagramme. Ce qu'il faut comprendre avant toute chose, c'est que les commandes et environnements permettant de définir un diagramme de séquence placent les éléments de manière automatique et les espacent également automatiquement. Nous y reviendrons en temps voulu.

4.2 Définir un objet

4.2.1 Les types d'objets

On peut définir un objet avec la commande `umlobject` :

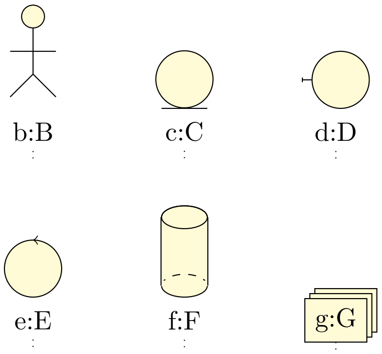
```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\end{umlseqdiag}
\end{tikzpicture}
```



Par défaut, il représente une instance de classe. Le nom de la classe est spécifié avec l'option `class` (vide par défaut).

L'option `stereo` permet de spécifier le type d'objet. Elle accepte les valeurs suivantes : object (valeur par défaut), actor, entity, boundary, control, database, multi. Les 6 derniers sont représentés ci-dessous, de gauche) droite et de haut en bas.

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlactor[class=B]{b}
\umlentity[x=2, class=C]{c}
\umlboundary[x=4, class=D]{d}
\umlcontrol[x=0, y=-2.5, class=E]{e}
\umldatabase[x=2, y=-2.5, class=F]{f}
\umlmulti[x=4, y=-2.5, class=G]{g}
\end{umlseqdiag}
\end{tikzpicture}
```



4.2.2 Positionnement automatique d'un objet

Les options `x` et `y` permettent de positionner un objet. En pratique, elles sont peu utilisées. En effet, le comportement automatique est le suivant :

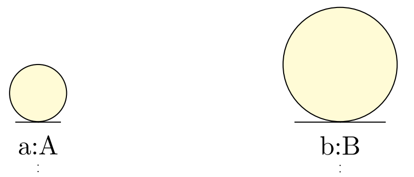
- L'option `y` vaut 0 par défaut, ce qui signifie qu'un objet se trouve par défaut en haut d'un diagramme de séquence.
- L'option `x` vaut un multiple de 4 par défaut. Ce multiple correspond à l'indice attribué à l'objet : si c'est le deuxième objet défini dans le diagramme, cet indice vaut 2, ...

A moins que la taille de l'objet soit trop grande, le décalage de 4 convient. Dans le cas contraire, on spécifie la coordonnée.

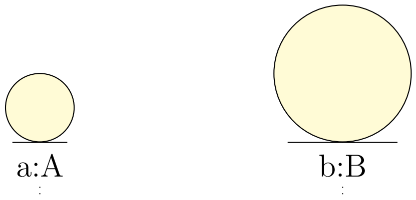
4.2.3 Dimensionner un objet

L'option `scale` de `umlobject` permet de dimensionner un objet. De même, les symboles sont adaptés à la taille de la fonte :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A, stereo=entity]{a}
\umlobject[x=4, scale=2, class=B, stereo=
    entity]{b}
\end{umlseqdiag}
\end{tikzpicture}
```



```
\tikzumlset{font=\large}
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A, stereo=entity]{a}
\umlobject[x=4, scale=2, class=B, stereo=
    entity]{b}
\end{umlseqdiag}
\end{tikzpicture}
```



4.3 Définir un appel de fonction

Les appels de fonction représentent le cœur d'un diagramme de séquence. Il faut donc en conséquence un moteur suffisamment intelligent pour permettre à la fois un comportement par défaut satisfaisant et une grande souplesse dans le paramétrage.

Pour parler un peu technique – j'ouvre une parenthèse – , il y avait 2 approches possibles :

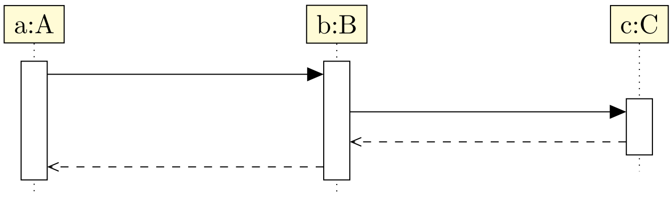
1. soit se baser sur la structure de matrice de nœuds de TikZ. Cela présente l'avantage de travailler sur une grille de nœuds précalculée et donc de positionner facilement (et rapidement à la compilation) à l'aide d'un unique compteur les éléments d'un diagramme de séquence.
2. soit se baser sur un positionnement automatique des nœuds par un jeu de coordonnées, dans le cas présent la coordonnée temporelle, qui permet une totale souplesse et facilite la reflexion de l'utilisateur.

C'est la seconde approche qui a été choisie, car davantage conforme à l'état d'esprit dans lequel les autres diagrammes ont été implémentés dans cette librairie. En effet, si l'absence de grille demande un moteur de calcul plus élaboré, et donc un temps de compilation un peu plus long, cette approche permet de définir facilement un certain nombre d'éléments de manière simple, comme un appel de constructeur, dessiné selon la norme et nom « bidouillé » pour que ça soit compréhensible, comme c'est le cas de la plupart des logiciels de conception UML que j'ai pu utiliser auparavant. Je referme donc la parenthèse.

4.3.1 Appels simples / récursifs

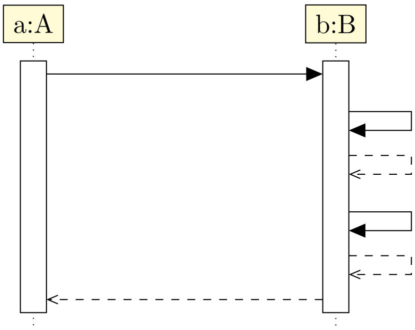
Un appel de fonction simple se fait à l'aide de l'environnement `umlcall`. Bien entendu, on peut utiliser des environnements `umlcall` imbriqués :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\umlobject[class=C]{c}
\begin{umlcall}{a}{b}
\begin{umlcall}{b}{c}
\end{umlcall}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



On précise en arguments obligatoires le nom de l'objet source et celui de l'objet destination. Si ces noms sont les mêmes, alors l'appel est récursif. Pour ne pas définir les deux arguments, dans ce cas identiques, on peut utiliser l'environnement `umlcallself` :

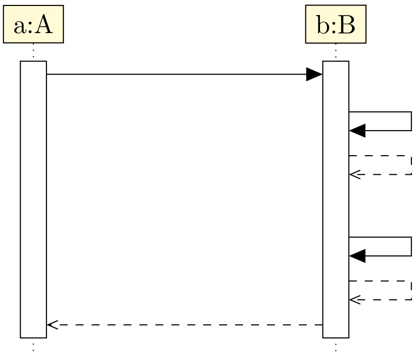
```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\begin{umlcall}{a}{b}
\begin{umlcall}{b}{b}
\end{umlcall}
\begin{umlcallself}{b}
\end{umlcallself}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



4.3.2 Positionnement d'un appel

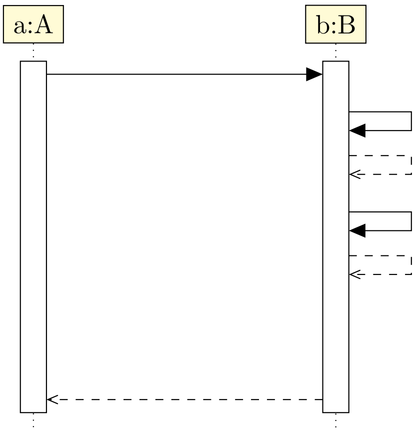
L'option `dt` permet de positionner un appel sur une ligne de vie, relativement au dernier élément présent sur cette ligne. Elle est vide par défaut. Elle se mesure en unité ex. En pratique, le comportement par défaut est de calculer un décalage suffisant pour que rien ne se chevauche entre 2 appels consécutifs :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\begin{umlcall}{a}{b}
\begin{umlcall}{b}{b}
\end{umlcall}
\begin{umlcallself}[dt=5]{b}
\end{umlcallself}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



On peut également ajuster l'espacement pour des appels imbriqués avec l'option `padding`. Cela concerne l'espacement en bas uniquement :

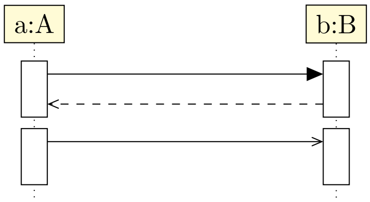
```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\begin{umlcall}[padding=10]{a}{b}
\begin{umlcall}{b}{b}
\end{umlcall}
\begin{umlcallself}{b}
\end{umlcallself}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



4.3.3 Appels synchrones/ asynchrones

L'option `type` permet de spécifier s'il s'agit d'un appel synchrone (valeur par défaut) ou asynchrone :

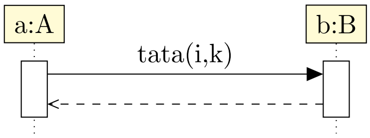
```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\begin{umlcall}[type=synchron]{a}{b}
\end{umlcall}
\begin{umlcall}[type=asynchron]{a}{b}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



4.3.4 Opération, arguments et valeur de retour

On spécifie le nom de la fonction appelée et ses arguments d'entrée avec l'option `op` :

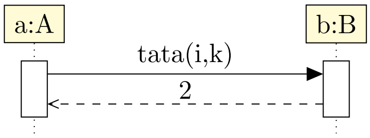
```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\begin{umlcall}[op={tata(i,k)}]{a}{b}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



⚠ Notez l'importance ici des accolades pour que la virgule séparant les 2 arguments i et k. Sans les accolades, elle serait interprétée comme un séparateur d'options et provoquerait une erreur.

On spécifie la valeur de retour avec l'option `return` avec les mêmes précautions :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlobject[class=B]{b}
\begin{umlcall}[op={tata(i,k)}, return
    =2]{a}{b}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```

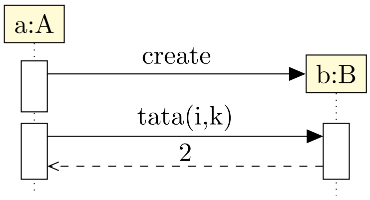


4.3.5 Définir un appel de constructeur

Les appels de constructeurs sont des appels de fonctions particuliers dans la mesure où ils créent un nouvel objet. Il ne s'agit pas de messages échangés entre 2 lignes de vies, mais entre une ligne de vie et un objet.

Pour définir un appel de constructeur, on utilise la commande `umlcreatecall` :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlcreatecall[class=B]{a}{b}
\begin{umlcall}[op={tata(i,k)}, return
    =2]{a}{b}
\end{umlcall}
\end{umlseqdiag}
\end{tikzpicture}
```



On peut remarquer que tout se passe normalement ensuite à la définition d'autres appels utilisant l'objet créé.

En tant que constructeur d'objet, la commande `umlcreatecall` dispose des options `class`, `stereo` et `x`.

En tant qu'appel de fonction, elle dispose de l'option `dt`.

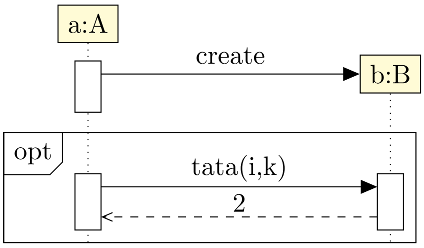
4.3.6 Nommer un appel

L'option `name` permet de donner un nom à un appel de fonction. Elle n'a que peu d'intérêt pour l'instant à moins de développements futurs, dans la mesure où elle avait été pensée pour la définition des fragments combinés, mais il n'y en a plus besoin.

4.4 Définir un fragment combiné

Les fragments combinés constituent la deuxième grande famille d'éléments constitutifs d'un diagramme de séquence. On peut les définir à l'aide de l'environnement `umlfragment` :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlcreatecall[class=B]{a}{b}
\begin{umlfragment}
\begin{umlcall}[op={tata(i,k)}, dt=7,
    return=2]{a}{b}
\end{umlcall}
\end{umlfragment}
\end{umlseqdiag}
\end{tikzpicture}
```

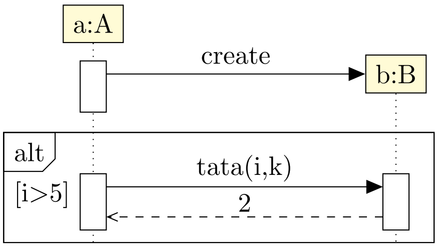


4.4.1 Informations d'un fragment

L'option `type` permet de spécifier le mot-clé apparaissant dans l'étiquette en haut à gauche : opt, alt, loop, par, assert, . . . La valeur par défaut est opt.

L'option `label` permet de spécifier les éléments complémentaires tels que la condition d'un fragment opt :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlcreatecall[class=B]{a}{b}
\begin{umlfragment}[type=alt, label=i>5,
    inner xsep=2]
\begin{umlcall}[op={tata(i,k)}, dt=7,
    return=2]{a}{b}
\end{umlcall}
\end{umlfragment}
\end{umlseqdiag}
\end{tikzpicture}
```

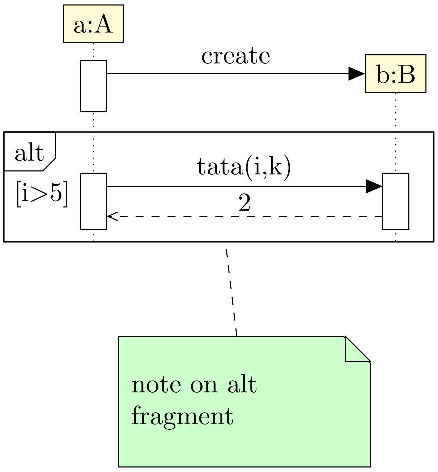


L'option `inner xsep` permet de décaler l'étiquette et le label vers la gauche. Sa valeur par défaut est 1 et son unité de longueur est ex.

4.4.2 Renommer un fragment

On peut nommer un fragment à l'aide de l'option `name`. Elle peut être très utile par exemple quand on veut attacher une note à un fragment :

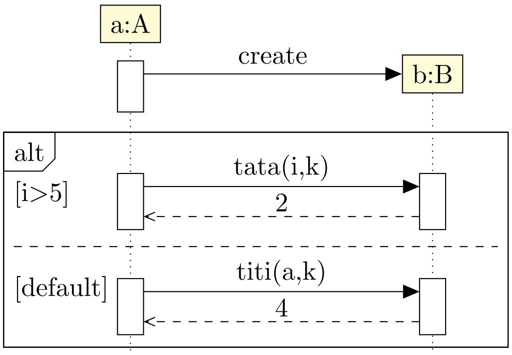
```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlcreatecall[class=B]{a}{b}
\begin{umlfragment}[type=alt, label=i>5,
   name=alt, inner xsep=2]
\begin{umlcall}[op={tata(i,k)}, dt=7,
   return=2]{a}{b}
\end{umlcall}
\end{umlfragment}
\umlnote[x=2, y=-5]{alt}{note on alt
   fragment}
\end{umlseqdiag}
\end{tikzpicture}
```



4.4.3 Définir les régions d'un fragment

Prenons le cas d'un fragment de type alt. Il s'agit de la représentation d'un bloc d'instructions de type switch - case. Pour pouvoir représenter ceci, on doit définir des régions dans le fragment. On utilise pour cela la commande `umlfpart` qui prend en option le label associé :

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlobject[class=A]{a}
\umlcreatecall[class=B]{a}{b}
\begin{umlfragment}[type=alt, label=i>5,
   inner xsep=5]
\begin{umlcall}[op={tata(i,k)}, dt=7,
   return=2]{a}{b}
\end{umlcall}
\umlfpart[default]
\begin{umlcall}[op={titi(a,k)}, return
   =4]{a}{b}
\end{umlcall}
\end{umlfragment}
\end{umlseqdiag}
\end{tikzpicture}
```



4.5 Personnalisation

Grace à la commande `tikzumlset`, on peut modifier globalement les couleurs par défaut des appels, des fragments et des objets :

text : permet de spécifier la couleur du texte (=black par défaut),

draw : permet de spécifier la couleur des traits et des flèches (=black par défaut),

fill object : permet de spécifier la couleur de fond des objets (=yellow!20 par défaut),

fill call : permet de spécifier la couleur de fond des appels (=white par défaut),

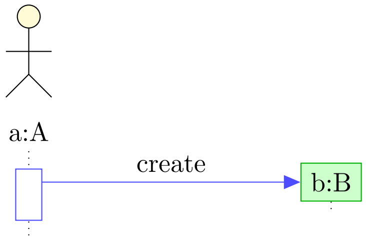
fill fragment : permet de spécifier la couleur de fond des cas d'utilisation (=white par défaut),

font : permet de spécifier le style de fonte du texte (=\small par défaut).

On peut également utiliser les options `text`, `draw` et `fill` sur un élément particulier pour lui modifier ses couleurs, comme illustré dans l'exemple d'introduction.

Il existe un cas particulier : `umlcreatecall`. Les options `text`, `draw` et `fill` permettent de modifier l'aspect du message de création, tandis que les options `text obj`, `draw obj` et `fill obj` permettent de modifier l'aspect de l'objet créé.

```
\begin{tikzpicture}
\begin{umlseqdiag}
\umlactor[class=A]{a}
\umlcreatecall[class=B, draw obj=green!70!black,
    fill obj=green!20, draw=blue!70]{a}{b}
\end{umlseqdiag}
\end{tikzpicture}
```



4.6 Exemples

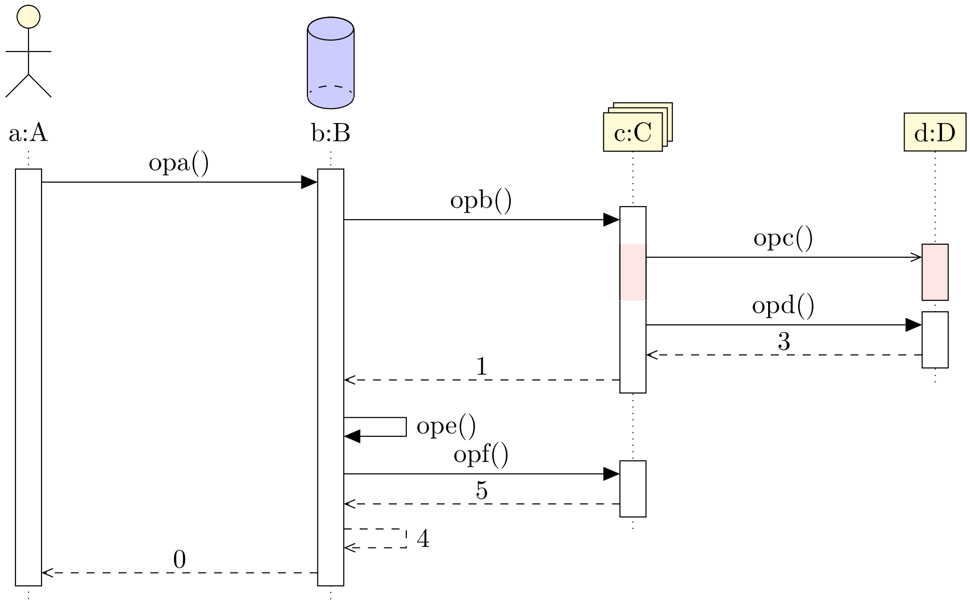
4.6.1 Exemple de l'introduction, pas à pas

Définition des objets

```
\begin{umlseqdiag}
\umlactor[class=A]{a}
\umldatabase[class=B, fill=blue!20]{b}
\umlmulti[class=C]{c}
\umlobject[class=D]{d}
\end{umlseqdiag}
```

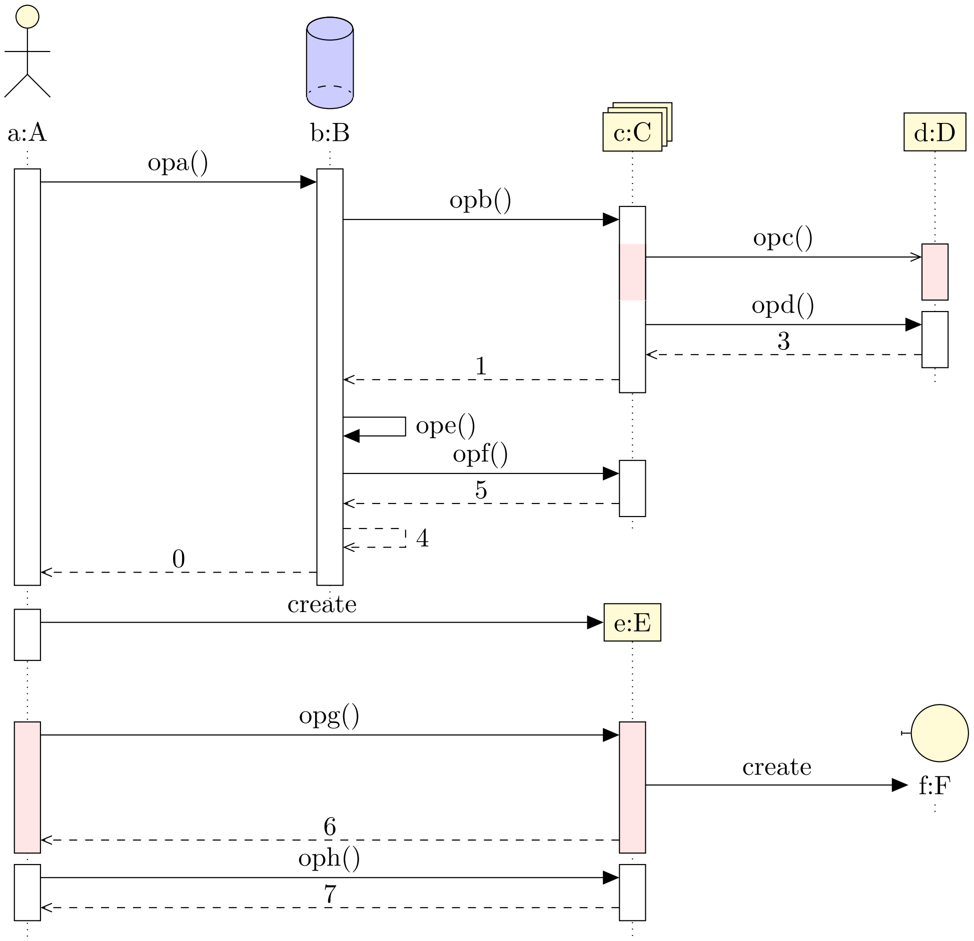
Définition de l'appel opa et ses composants

```
\begin{umlseqdiag}
\umlactor[class=A]{a}
\umldatabase[class=B, fill=blue!20]{b}
\umlmulti[class=C]{c}
\umlobject[class=D]{d}
\begin{umlcall}[op=opa(), type=synchron, return=0]{a}{b}
\begin{umlcall}[op=opb(), type=synchron, return=1]{b}{c}
\begin{umlcall}[op=opc(), type=asynchron, fill=red!10]{c}{d}
\end{umlcall}
\begin{umlcall}[op=opd(), type=synchron, return=3]{c}{d}
\end{umlcall}
\end{umlcall}
\begin{umlcallself}[op=ope(), type=synchron, return=4]{b}
\begin{umlcall}[op=opf(), type=synchron, return=5]{b}{c}
\end{umlcall}
\end{umlcallself}
\end{umlcall}
\end{umlseqdiag}
```



Définition des appels consécutifs à la construction de E

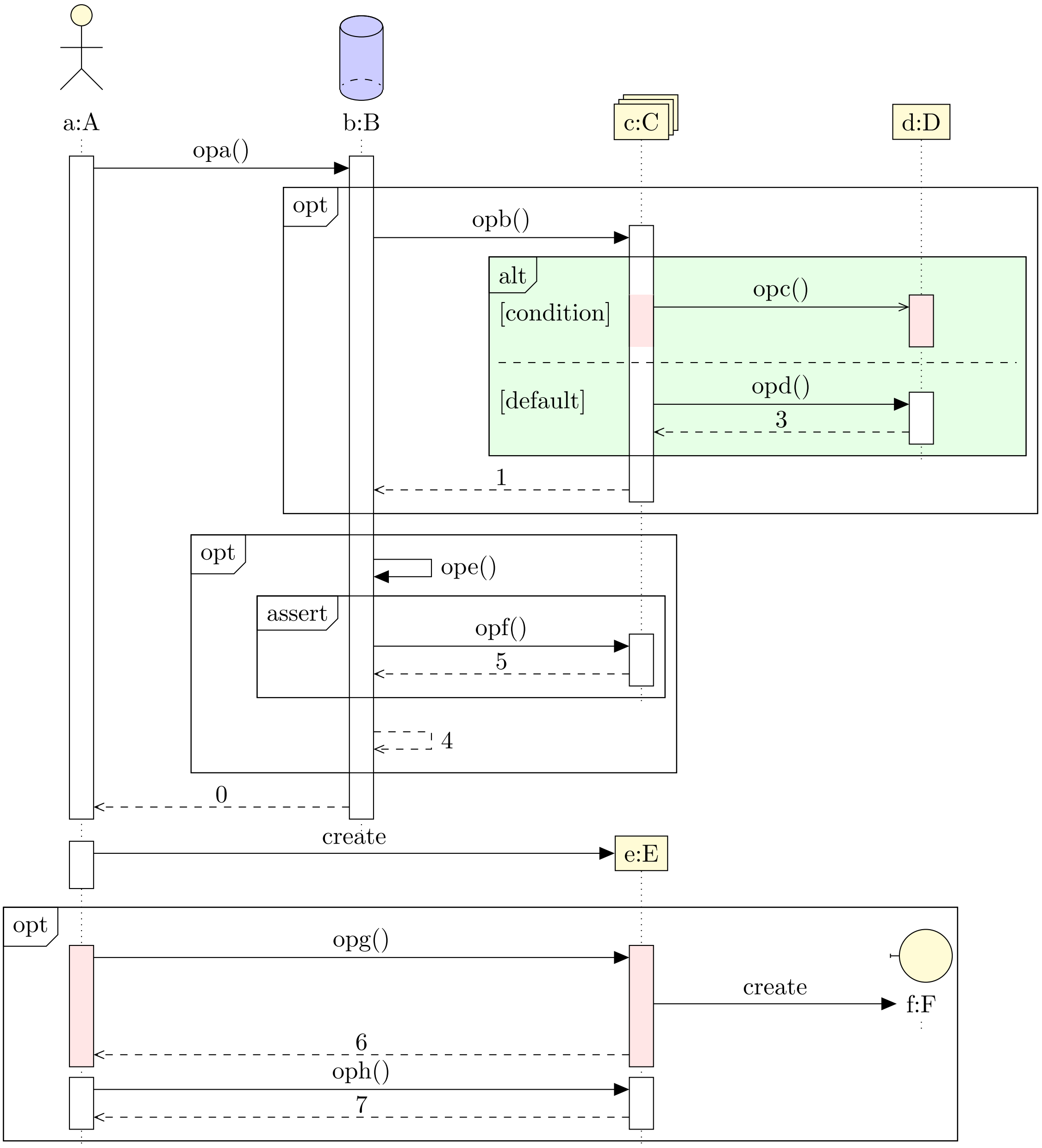
```
\begin{umlseqdiag}
\umlactor[class=A]{a}
\umldatabase[class=B, fill=blue!20]{b}
\umlmulti[class=C]{c}
\umlobject[class=D]{d}
\begin{umlcall}[op=opa(), type=synchron, return=0]{a}{b}
\begin{umlcall}[op=opb(), type=synchron, return=1]{b}{c}
\begin{umlcall}[op=opc(), type=asynchron, fill=red!10]{c}{d}
\end{umlcall}
\begin{umlcall}[op=opd(), type=synchron, return=3]{c}{d}
\end{umlcall}
\end{umlcall}
\begin{umlcallself}[op=ope(), type=synchron, return=4]{b}
\begin{umlcall}[op=opf(), type=synchron, return=5]{b}{c}
\end{umlcall}
\end{umlcallself}
\end{umlcall}
\umlcreatecall[class=E, x=8]{a}{e}
\begin{umlcall}[op=opg(), name=test, type=synchron, return=6, dt=7, fill=red!10]{a
    }{e}
\umlcreatecall[class=F, stereo=boundary, x=12]{e}{f}
\end{umlcall}
\begin{umlcall}[op=oph(), type=synchron, return=7]{a}{e}
\end{umlcall}
\end{umlseqdiag}
```



Définition des fragments

```
\begin{umlseqdiag}
\umlactor[class=A]{a}
\umldatabase[class=B, fill=blue!20]{b}
\umlmulti[class=C]{c}
\umlobject[class=D]{d}
\begin{umlcall}[op=opa(), type=synchron, return=0]{a}{b}
\begin{umlfragment}
\begin{umlcall}[op=opb(), type=synchron, return=1]{b}{c}
\begin{umlfragment}[type=alt, label=condition, inner xsep=8, fill=green!10]
\begin{umlcall}[op=opc(), type=asynchron, fill=red!10]{c}{d}
\end{umlcall}
\umlfpart[default]
\begin{umlcall}[op=opd(), type=synchron, return=3]{c}{d}
\end{umlcall}
\end{umlfragment}
\end{umlcall}
\end{umlfragment}
\begin{umlfragment}
\begin{umlcallself}[op=ope(), type=synchron, return=4]{b}
\begin{umlfragment}[type=assert]
\begin{umlcall}[op=opf(), type=synchron, return=5]{b}{c}
\end{umlcall}
\end{umlfragment}
\end{umlcallself}
\end{umlfragment}
```

```
\end{umlcall}
\umlcreatecall[class=E, x=8]{a}{e}
\begin{umlfragment}
\begin{umlcall}[op=opg(), name=test, type=synchron, return=6, dt=7, fill=red!10]{a
   }{e}
\umlcreatecall[class=F, stereo=boundary, x=12]{e}{f}
\end{umlcall}
\begin{umlcall}[op=oph(), type=synchron, return=7]{a}{e}
\end{umlcall}
\end{umlfragment}
\end{umlseqdiag}
```



4.7 Bugs identifiés et perspectives

1. Lorsqu'on définit un fragment sur un ensemble d'appels immédiatement après un appel de constructeur, le décalage automatique ne fonctionne pas. Il faut nécessairement donner à l'option `dt` du premier appel du fragment une valeur supérieure ou égale à 7.

2. Le placement automatique des objets avec un multiple de 4 ne convient pas. Il faudrait définir un décalage de 4 avec le dernier objet défini.
3. Il n'est pas possible pour l'instant de donner des arguments à un appel de constructeur.
4. On en peut pas forcer le tracé d'une zone d'activité d'un objet qui ne « travaille » pas.