Production et exploitation de documents bureautiques en Perl

Introduction à ODF::lpOD

Jean-Marie Gouarné
jmgdoc@cpan.org
http://jean.marie.gouarne.online.fr

EN BREF *Le module* ODF::lpOD *apporte un nouveau style d'interface pour la manipulation des documents au format ODF[1]. S'appuyant sur* Archive::Zip *pour la gestion physique des enveloppes ODF et sur* XML::Twig *pour le traitement du contenu,* ODF::lpOD *est le premier module créé sous l'espace de noms ODF. Cet article, qui n'est pas un manuel de référence, est une introduction didactique à son utilisation et une compilation de recettes faciles ; son objectif est de permettre une prise en main rapide, de présenter par des exemples simples quelques fonctionnalités représentatives, et de faciliter la mise en place d'une « preuve du concept ».*

1 *Open Document Format for Office Applications*, correspondant à la norme *ISO/IEC 26300.*

SOMMAIRE

1 Remarques préliminaires

1.1 Origine

ODF::lpOD est une retombée du projet lpOD[2], mené en 2009 et 2010 avec pour objectif principal la réalisation, à partir d'un tronc commun de spécifications fonctionnelles, d'interfaces de programmation pour ODF dans plusieurs langages.

ODF::lpOD a été, au départ, l'implémentation en Perl de ce projet. Des enrichissements ont par la suite été apportés, au-delà du périmètre des spécifications communes de lpOD.

C'est un logiciel libre distribué sous licences *GNU GPL V3* et *Apache V2*. La distribution de référence est celle du CPAN.

1.2 Environnement et caractéristiques techniques

Neutre par rapport à la plate-forme, ODF::lpOD nécessite *Perl 5.10*, *Archive::Zip 1.30* et *XML::Twig 3.34*. Les modules *File::Slurp*, *Image::Size* et *LWP::Simple* (sans exigence de version) sont également requis[3].

ODF::lpOD travaille *directement sur les fichiers ODF* et n'utilise ni *LibreOffice*, ni *OpenOffice.org*, ni *Microsoft Office*[4] ni aucun autre logiciel bureautique compatible ODF.

L'interface d'ODF::lpOD est entièrement orientée objet, à l'exception des constructeurs d'instances qui peuvent être, au choix, des méthodes de classes ou des fonctions, et de quelques fonctions utilitaires.

Le code d'ODF::lpOD utilise des formes syntaxiques du « Perl moderne », ce qui explique la nécessité d'une version de Perl supérieure ou égale à *5.10.0*. Ce code est en pur Perl et ne comporte aucune dépendance de plate-forme.

1.3 Avant de commencer

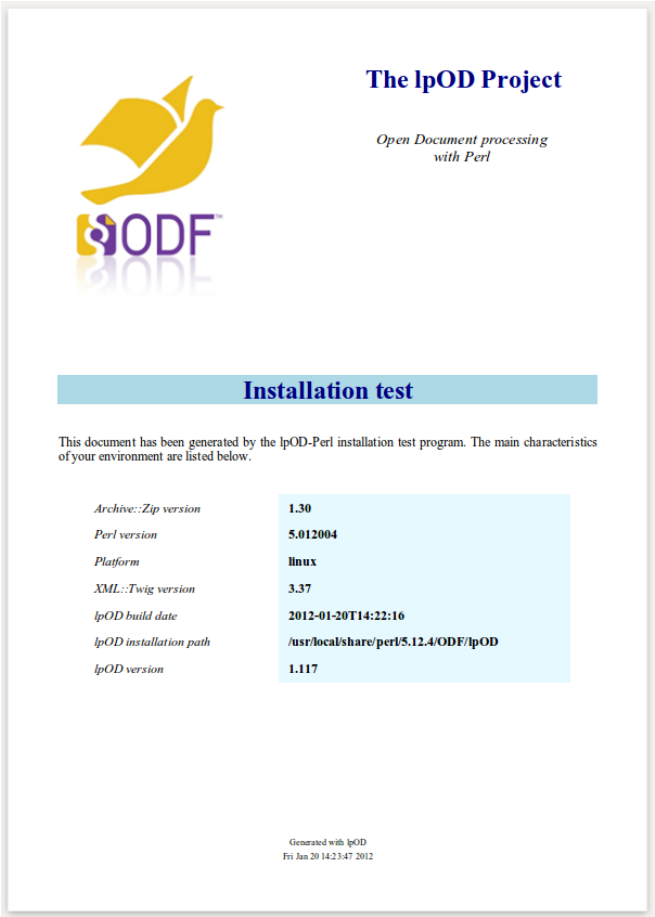
La première précaution à prendre est de vérifier si l'installation de lpOD est parfaitement opérationnelle. Nous ne présentons pas ici la procédure d'installation, d'ailleurs très classique[5]. Cependant, une fois celle-ci exécutée, nous disposons[6] d'un script exécutable lpod_test (fourni dans la distribution) qui, lancé en ligne de commande sans argument, doit renvoyer à la console une ligne indiquant successivement « ODF::lpOD », le numéro de version, la date de « *build* », c'est-à-dire la date de production de la distribution CPAN (en format *ISO 8601*), et enfin le chemin du répertoire d'installation du module. Par exemple avec ODF::lpOD 1.117 sur une plate-forme *Unix/Linux* on obtient une réponse de ce genre (le chemin d'installation pouvant varier selon la plate-forme et la version de Perl) :

```
ODF::lpOD 1.117 (build 2012-01-20T14:22:16) /usr/local/share/perl/5.12.4/ODF/lpOD
```

Plus intéressante est la possibilité de générer de toutes pièces un document de test. Pour cela il suffit de lancer lpod_test avec, en argument, un nom arbitraire (mais légal en tant que nom de fichier pour le système d'exploitation). Dans ce cas, lpod_test crée un fichier ODF de type texte (*.odt*) contenant les mêmes informations ainsi que d'autres, avec un petit effort de présentation. Le fichier est créé en utilisant le nom passé

2 « *Langages et plates-formes pour Open Document* », projet soutenu par l'*Agence Nationale de la Recherche* (2009-2010).
3 Voir le *Makefile.PL* de la dernière distribution CPAN pour la liste exacte des dépendances.
4 Bien qu'il ne soit pas le format par défaut, ODF est supporté à partir de *Microsoft Office 2010*.
5 Voir *INSTALL* pour ce qui concerne la distribution CPAN.
6 À partir de la version *1.117*

en argument (le suffixe *.odt* est automatiquement ajouté si nécessaire). Si tout va bien, ce fichier peut être ouvert avec notre traitement de texte préféré ; il s'agit d'un rapport succinct sur l'installation, en une seule page. L'image suivante décrit l'aspect général de cette page (il faut bien sûr faire abstraction du contenu qui varie selon les environnements).



En examinant le document, on remarque qu'il contient un entête, un pied de page, un cadre d'image, un tableau, et des éléments textuels présentés selon différents styles.

En plus de son contenu mis en page, le document contient des informations générales que nous pouvons vérifier via les différents onglets accessibles par la commande *Fichiers/Propriétés* de notre traitement de texte. L'onglet *Général* indique la date et l'heure de création et de modification du document (ces deux dates doivent être identiques et correspondre au moment ou `lpod_test` a été exécuté). Dans l'onglet *Description* les rubriques *Titre*, *Sujet* et *Mots-clés* sont renseignées. L'onglet *Propriétés personnalisées* a été lui aussi renseigné. Le programme a donc également agi sur les « *métadonnées* » du document.

Ce document ayant été entièrement produit par `lpod_test`, on comprend que ce programme est fourni non seulement pour vérifier l'installation, mais aussi comme un exemple de code illustrant la mise en œuvre de plusieurs fonctionnalités usuelles de lpOD. Toutefois, avant de se précipiter sur l'analyse du code de `lpod_test`, il vaut mieux retenir deux choses :

1) ce programme étant fourni à des fins de test et de démonstration, il fait quelque chose qui est de préférence à éviter dans une « vraie » application, à savoir la génération complète d'un document *avec sa présentation*. Un document est une forme d'interface graphique, et comme avec toute interface graphique c'est toujours la présentation qui demande le plus d'efforts de codage quand on veut tout gérer par programme. La bonne pratique consiste à préparer des modèles de documents (*templates*) en utilisant les outils bureautiques appropriés, et non à gaspiller un temps précieux à décrire en Perl des polices, des couleurs, des tailles et des positions. lpOD permet bien entendu de fabriquer des styles en Perl, mais permet encore plus facilement de *rechercher*, de *répliquer* et de *personnaliser* des styles déjà définis ;

2) le code de `lpod_test` sera (j'espère) beaucoup plus facile à comprendre et à réutiliser *après* la lecture de cet article qu'avant.

1.4 Conventions du présent article

Dans les exemples de code, nous utilisons parfois des formes syntaxiques de *Perl 5.10*, dont l'utilisation suppose le pragma `use 5.010_000` (ou supérieur), sans que ce soit rappelé à chaque fois.

Plus généralement, pour l'exécution de tous les exemples, je recommande l'usage systématique d'un entête de ce type :

```
use 5.010_000;
use strict;
use warnings;
use ODF::lpOD;
```

Ou mieux encore :

```
use Modern::Perl;
use ODF::lpOD;
```

2 Première création

Dans le droit fil d'une longue tradition, nous commencerons par l'incontournable « *Bonjour le monde !* »... mais ce message rituel sera affiché dans un document dûment conditionné au format ODF, et non pas comme d'habitude dans une console.

2.1 Création du document

Nous devons donc, dans un premier temps, créer un document, c'est-à-dire une instance de la classe `ODF::lpOD::Document`, que nous pouvons aussi utiliser par son petit nom `odf_document` (car lpOD fournit des noms abrégés pour toutes les classes). Mais quel genre de document ? Le format ODF, qui se décline notamment en *ODT*, *ODS*, *ODP* et *ODG* s'applique aussi bien au traitement de texte, au tableur, au support de présentation ou au dessinateur. Les quatre types de documents que je viens de citer sont supportés par lpOD, et sont respectivement identifiés par les mots-clés `text`, `spreadsheet`, `presentation` et `drawing`. Choisissons, pour commencer, le type `text`. Comme vous le voyez, rien n'est plus simple que la création du document :

```
my $doc = odf_document->create('text');
```

Après cette première instruction, `$doc` est une instance de document pour traitement de texte. En pratique, ce document est vide, hormis quelques définitions de styles basiques. De plus, il n'est pas persistant (si notre programme s'arrête là, rien ne sera enregistré sur le disque). Nous avons juste appelé la méthode `create` de la classe `odf_document` en lui passant le type de document à créer ; cette méthode retourne un squelette de document prêt à l'emploi.

Notez bien que lpOD offre des syntaxes alternatives. La même instruction pourrait s'écrire aussi

```
my $doc = ODF::lpOD::Document->create('text');
```

ou encore

```
my $doc = odf_create_document('text');
```

C'est une question de style sur laquelle chacun prendra son parti en s'appuyant sur la page de manuel de `ODF::lpOD::Document`.

2.2 Création d'un élément de texte

Un fragment de texte ODF ne s'écrit pas directement dans une page comme avec une craie sur un tableau noir. Un texte doit être attaché à un support approprié, soit dans la plupart des cas un *paragraphe*. Nous devons donc créer un paragraphe, c'est-à-dire un objet `ODF::lpOD::Paragraph`, alias `odf_paragraph`, et y placer notre texte. Ce n'est pas non plus très compliqué :

```
my $p = odf_paragraph->create(text => "Bonjour le monde !");
```

Notez que le contenu du paragraphe est passé par un paramètre nommé `text`. Ce paramètre est optionnel, car il est possible de créer un paragraphe vide. Un paragraphe peut aussi être créé avec d'autres paramètres que `text` (mais nous y reviendrons plus loin).

Le constructeur de paragraphe, tout comme le constructeur de document, peut être appelé selon des syntaxes alternatives. Ainsi, chacune des deux instructions suivantes est équivalente à l'instruction ci-dessus :

```
my $p = odf_create_paragraph(text => "Bonjour le monde !");
my $p = ODF::lpOD::Paragraph->create(text => "Bonjour le monde !");
```

Nous avons donc à présent un document `$doc` et un paragraphe `$p`. Remarquez que ces deux objets, dans notre exemple, auraient pu être créés dans n'importe quel ordre, vu qu'il n'y a jusqu'à présent aucun lien entre eux. Il reste à ancrer le paragraphe dans le document.

2.3 Insertion du texte dans le document

Le rattachement d'un paragraphe à un document n'est pas syntaxiquement compliqué mais requiert la connaissance d'une caractéristique importante (qui est aussi une limite) de lpOD : l'API permet à l'utilisateur d'insérer pratiquement n'importe quoi n'importe où, ce qui donne beaucoup de liberté mais impose une compréhension minimale de la structure du document. En l'occurrence, il faut d'abord choisir un *contexte*, c'est-à-dire un élément du document auquel nous pourrons rattacher le nouveau paragraphe de telle manière qu'il apparaisse effectivement au début de la première page.

Dans le cas présent, c'est assez simple. Comme il s'agit d'un document de type `text`, le paragraphe peut être directement rattaché à un contexte particulier que nous pouvons appeler le « corps » du document (*document body*), qui est en quelque sorte le « conteneur suprême » de tout ce qui apparaît dans le corps des pages. C'est le cas le plus simple et, bonne nouvelle, il existe une méthode `get_body()` ou `body()`, de la classe `odf_document` qui nous livre ce contexte sur un plateau :

```
my $contexte = $doc->get_body;
```

L'objet `$contexte` ainsi récupéré va nous servir de point d'ancrage pour rattacher le paragraphe de la façon suivante :

```
$contexte->insert_element($p);
```

C'est fait, le paragraphe est désormais accroché au document. La méthode `insert_element()` rattache hiérarchiquement l'objet qui lui est passé en argument à l'objet appelant. Notez-la bien, elle servira souvent ; en fait `insert_element()` permet de rattacher n'importe quoi à n'importe quoi, et pas seulement un paragraphe à un corps de document.

Mais nous n'avons pas tout à fait fini. Nous disposons, *en mémoire*, d'un document dont le corps contient un paragraphe qui contient lui-même un texte, mais ce document n'est pas encore un *fichier* ODF et, en cas de panne de courant, il disparaîtra.

2.4 Enregistrement du document

Là encore, il n'y a pas de complication à attendre. La classe `odf_document` est équipée d'une méthode `save()` prête à terminer le travail. Encore faut-il lui indiquer le nom (et éventuellement le chemin) d'un fichier de destination :

```
$doc->save(target => "bonjour.odt");
```

Après l'instruction ci-dessus, le traitement est complet, le programme peut s'arrêter et vous pouvez vous ruer sur votre traitement de texte pour examiner la physionomie du document ainsi créé.

2.5 Programme complet

Récapitulons la séquence et écrivons le programme entier :

```
use ODF::lpOD;
my $doc = odf_document->create('text');
my $p = odf_paragraph->create(text => "Bonjour le monde !");
my $contexte = $doc->get_body;
$contexte->insert_element($p);
$doc->save(target => "bonjour.odt");
exit;
```

Pour les amateurs d'instructions longues et/ou les allergiques à la prolifération des variables intermédiaires, on peut écrire la même chose sous une forme légèrement plus compacte... et moins pédagogique :

```
use ODF::lpOD;
my $doc = odf_document->create('text');
$doc->body->insert_element(
    odf_paragraph->create(text => "Bonjour le monde !")
    );
$doc->save(target => "bonjour.odt");
exit;
```

2.6 Variante « tableur »

Compliquons un peu l'exercice. Au lieu d'insérer notre petite phrase « à plat » dans un document textuel, nous allons l'inscrire dans la première cellule d'une feuille de calcul toute neuve.

Il s'agit d'abord de créer un document du type approprié :

```
my $doc = odf_document->create('spreadsheet');
```

Ensuite il faut, comme d'habitude, accéder au corps du document :

```
my $contexte = $doc->get_body;
```

Par défaut (sauf si vous avez touché à la configuration de lpOD), tout document de type `spreadsheet` est créé avec une première feuille (arbitrairement nommée « *Sheet1* »). Mais nous ne sommes pas obligés de l'utiliser et, pour les besoins de la démonstration, nous allons régler radicalement la question en partant d'un contexte vide ; il suffit pour cela d'utiliser le nettoyant universel qu'est la méthode `clear()` :

```
$contexte->clear;
```

Notez que `clear()` est une méthode applicable à tout élément dans n'importe quel type de document. Elle a pour effet non seulement d'effacer le contenu direct de l'élément appelant, mais aussi de supprimer tous les éléments qui dépendent de lui dans la hiérarchie. Donc, appliquée au corps du document lui-même (notre objet `$contexte`), elle efface tout le contenu, mais ne touche pas aux styles et aux métadonnées, qui sont ailleurs.

Ayant ainsi fait table rase, nous devons créer une feuille de calcul (destinée à apparaître dans le tableur comme le premier et unique onglet du document). Nos besoins sont modestes : pour écrire notre petit message de bienvenue, il suffit d'une seule cellule, donc nous avons juste besoin d'un tableau de dimensions 1×1 :

```
my $feuille = odf_table->create(
    "Ma Feuille", width => 1, length => 1
    );
```

La syntaxe de création d'une table est assez souple ; à condition de ne pas oublier que c'est la *hauteur*, c'est-à-dire le nombre de lignes, qui doit être indiquée en premier, même si c'est sans importance dans notre exemple, le constructeur de table accepte que les deux paramètres `length` et `width` soient remplacés par un paramètre `size` dont la valeur peut être, au choix, une référence de liste ou une chaîne de deux valeurs séparées par une virgule. Les deux formes suivantes sont donc possibles :

```
my $feuille = odf_table->create("Ma Feuille", size => [1, 1]);
my $feuille = odf_table->create("Ma Feuille", size => "1, 1");
```

Il faut noter aussi que la même classe `ODF::lpOD::Table`, alias `odf_table`, correspond aux feuilles de calcul pour tableurs et aux tableaux destinés à être insérés dans des documents de type texte ou présentation. Dans tous les cas, le premier argument doit être le nom (unique) du tableau.

La feuille est créée mais, tout comme notre paragraphe dans la version « texte », il faut la rattacher au document. Sans surprise, ce rattachement se fait exactement de la même manière que pour un paragraphe :

```
$contexte->insert_element($feuille);
```

Nous avons donc une feuille de calcul intégrée au document et contenant une cellule vide. Nous allons sélectionner cette cellule à l'aide de la méthode `get_cell()` qui, appelée à partir de l'objet `odf_table`, a seulement besoin qu'on lui indique les coordonnées de la cellule recherchée. Ces coordonnées peuvent être exprimées, au choix, dans le format « bataille navale » (lettres et chiffres) représentatif de la vision de l'utilisateur final d'un tableur, ou sous forme numérique en commençant par le numéro de ligne et en sachant que dans ce cas la première position est 0 et non pas 1. Dans le cas présent nous avons donc le choix entre `"A1"` et `(0, 0)` :

```
my $cellule = $feuille->get_cell("A1");
```

Une fois en possession de la cellule, il faut y inscrire notre texte, le moyen le plus simple étant la méthode `set_text()`, disponible pour les cellules de tables comme pour beaucoup d'autres classes d'éléments :

```
$cellule->set_text("Bonjour le monde !");
```

À ce stade il n'y a plus qu'à enregistrer le document, de la manière habituelle. Notre programme modifié peut ressembler, par exemple, à ceci :

```
use ODF::lpOD;
my $doc = odf_document->create('spreadsheet');
my $contexte = $doc->get_body;
$contexte->clear;
my $feuille = odf_table->create("Ma Feuille", size => "1, 1");
$feuille->get_cell("A1")->set_text("Bonjour le monde !");
$contexte->insert_element($feuille);
$doc->save(target => "bonjour.ods");
exit;
```

On pourrait aussi l'abréger comme ci-après, sachant que la valeur de retour de `insert_element()` est l'objet inséré, donc en l'occurrence la table (mais ce style ne favorise pas forcément la compréhension).

```
use ODF::lpOD;
my $doc = odf_document->create('spreadsheet');
my $contexte = $doc->body;
$contexte->clear;
$contexte->insert_element(
    odf_table->create("Ma Feuille", size => "1, 1")
    )
    ->get_cell("A1")
    ->set_text("Bonjour le monde !");
$doc->save(target => "bonjour.ods");
exit;
```

On notera au passage que, avec la méthode `save()`, le paramètre `target` doit être valorisé avec le nom complet du fichier. Ce n'est pas parce que le document est de type `spreadsheet` que lpOD choisira automatiquement de lui ajouter une extension « *.ods* ». Avec lpOD, l'application est totalement libre de choisir l'extension, de sorte que si vous créez des fichiers non destinés à être manipulés ensuite par des utilisateurs bureautiques ordinaires, vous pouvez très bien leur donner par exemple des extensions « *.zip* », « *.jar* » ou autres. (Rien ne vous interdit non plus, d'ailleurs, d'utiliser lpOD pour créer un paquetage « *.par* » embarquant à la fois un document ODF et des modules Perl, mais c'est un sujet qui dépasse le cadre de cette modeste présentation).

3 Opérations globales sur un document

Nous abordons ici certaines fonctionnalités concernant l'ensemble du document (et non son contenu). Ces fonctionnalités sont documentées dans la page de manuel `ODF::lpOD::Document`.

3.1 Création ou accès

Comme nous l'avons vu, un document est un objet de classe `ODF::lpOD::Document`, et une instance peut être créée en utilisant le constructeur `create()`. Cependant, dans beaucoup de situations (voire dans la plupart), nous aurons à reprendre des documents à partir de fichiers déjà existants, créés par une application bureautique, par un autre programme lpOD, ou autrement. Dans ce cas, il faut utiliser le constructeur `get()` dont l'argument obligatoire est le nom complet du fichier[7] , comme dans l'exemple qui suit.

7 À la rigueur, on peut aussi utiliser un objet `IO::File` représentant un fichier déjà ouvert, à condition qu'il s'agisse d'une ressource manipulable en accès direct (*seekable*). lpOD, pour l'instant, ne fonctionne pas en *streaming*…

```
my $doc = odf_document->get("C:\Documents\Factures\F1234.ods");
```

Le traitement d'un document commence donc en général par un `get()` ou par un `create()`.

Au passage, signalons à toutes fins que les constructeurs d'objets de lpOD peuvent être appelés, au choix, selon un style « objet », c'est-à-dire comme méthodes de classe, ou selon un style « fonctionnel ». Dans le second cas, il s'agit toujours d'une fonction dont le nom commence par `odf_`, de manière à éviter de polluer l'espace de nommage des applications (car il s'agit de fonctions exportées). Donc `odf_document->create()` et `odf_document->get()` peuvent être respectivement remplacés par `odf_create_document()` et `odf_get_document()` avec les mêmes arguments.

3.2 Enregistrement

Dans notre premier exemple de programme nous avons vu comment rendre persistante la création d'un document via la méthode `save()`. Lorsqu'il s'agit d'un document nouvellement créé, `save()` requiert un paramètre `target` indiquant la cible. Mais s'il s'agit d'un document préexistant chargé avec `get()`, ce paramètre est facultatif ; si `target` n'est pas spécifié, alors `save()` réécrit par-dessus le fichier source (comme le fait n'importe quel traitement de texte ou tableur). L'exemple ci-dessous montre comment reprendre un document existant, insérer un paragraphe en première position, puis enregistrer le changement :

```
my $doc = odf_document->get("travail.odt");
$doc->get_body->insert_element(
    odf_paragraph->create(text => "Premier paragraphe")
    );
$doc->save;
```

Cela dit, même si le document a été instancié à partir d'un fichier existant, il est toujours permis d'utiliser le paramètre `target` pour spécifier une cible différente, de manière à laisser le fichier source inchangé. C'est même une pratique recommandée, sachant que, dans une application réelle, on a généralement intérêt à partir d'un fond de document comportant une structure et une présentation appropriées, et à ne traiter par programme que les parties variables.

Remarque : la méthode de classe `odf_paragraph->create()` invoquée dans cet exemple pourrait être remplacée par la fonction `odf_create_paragraph()`, selon le principe déjà signalé et qui vaut pour tous les constructeurs d'objets de lpOD.

Pour les utilisateurs à l'aise en XML et qui souhaitent pouvoir investiguer à loisir dans les fichiers ODF créés ou modifiés par lpOD (à des fins de débogage ou pour y faire de quelconques opérations manuelles), il est possible de passer à `save()` un paramètre `pretty` qui, si sa valeur est `TRUE`, permet d'obtenir un XML plus « joli », c'est-à-dire indenté, donc humainement lisible (le seul prix à payer est alors un fichier très légèrement plus volumineux) :

```
$doc->save(pretty => TRUE);
```

À titre d'illustration combinée des options `target` et `pretty`, l'exemple ci-dessous montre comment, en une seule instruction, on peut créer un nouveau document qui, par son contenu et sa présentation, sera intégralement identique à un document existant, mais dont le XML sera plus aéré :

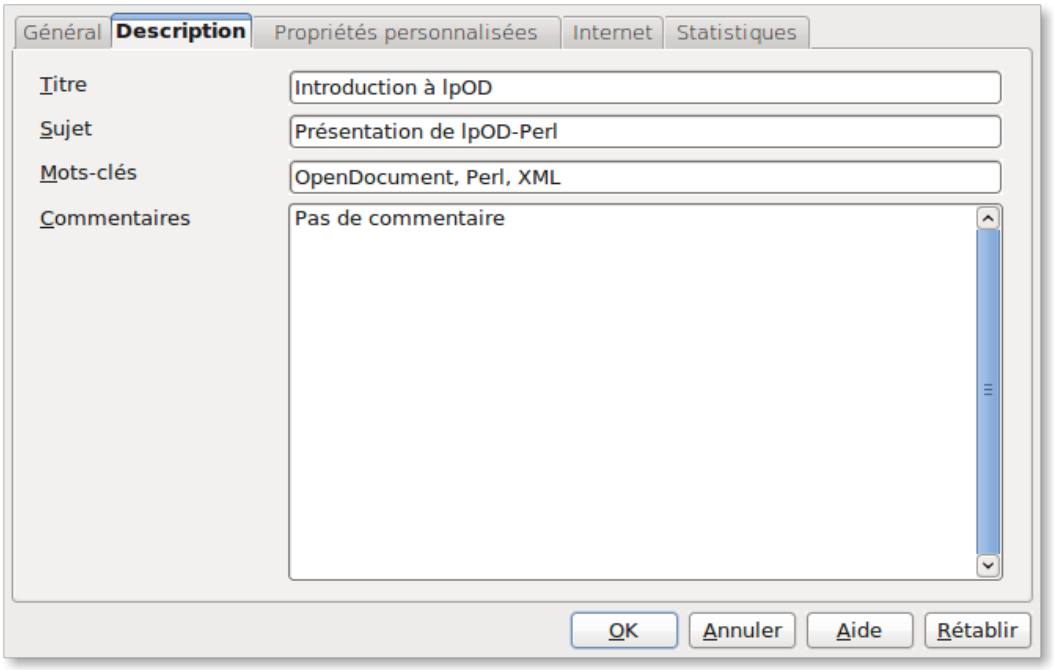
```
odf_document->get("source.odt")
    ->save(target => "cible.odt", pretty => TRUE);
```

3.3 Métadonnées globales

3.3.1 Description générale

On appelle ici *métadonnées* les informations globales sur le document, dont la plupart (mais pas toutes) sont visibles pour l'utilisateur d'une suite bureautique classique à travers une boîte de dialogue « Propriétés du document » ou équivalent.

Pour commencer, prenons un exemple sous traitement de texte (*Writer*), choisissons l'entrée *Propriétés* du menu *Fichier*, et sélectionnons l'onglet *Description*. Nous obtenons un cadre présentant un titre, un sujet, une liste de mots-clés séparés par des virgules, et une zone de commentaires. Ces données sont parfois renseignés par l'utilisateur… mais pas toujours. Nous allons voir comment, avec lpOD, on peut facilement les injecter par programme, de manière à obtenir ensuite une fenêtre conforme à l'image suivante :



Une petite explication préalable : un document ODF (vu à travers lpOD) se compose de plusieurs espaces de travail appelés « *parts* » et accessibles chacun au moyen d'une méthode `get_part()` dont l'argument est le nom de l'espace recherché. Tout ce qui apparaît dans le corps des pages ou dans les feuilles de calcul appartient à une « *part* » dont le nom de code est `CONTENT` (le contenu). Dans l'exemple précédent, nous n'avons pas eu besoin de faire un appel explicite de `get_part(CONTENT)` car nous avons utilisé une méthode `get_body()` qui est un raccourci donnant directement accès au contexte le plus souvent utilisé, à l'intérieur de l'espace `CONTENT`. Mais pour accéder aux métadonnées, notre contexte est l'espace `META`, qui doit être explicitement sélectionné, de la façon suivante, en supposant bien sûr que `$doc` est un document déjà instancié via un `get()` ou un `create()` :

```
my $contexte = $doc->get_part(META);
```

La même instruction peut s'écrire sous une forme abrégée :

```
my $contexte = $doc->meta;
```

Le contexte ainsi défini est un objet « spécialisé dans les métadonnées », possédant un certain nombre d'accesseurs `set_xxx()` et `get_xxx()` permettant de modifier ou de consulter toutes les métadonnées et en particulier celles qui apparaissent dans notre copie d'écran.

Les zones titre, sujet et commentaires correspondent respectivement aux propriétés *title*, *subject* et *description*, et il est facile d'en déduire les noms des accesseurs permettant de les modifier. Inscrivons sans plus tarder les valeurs adéquates :

```
$contexte->set_title("Introduction à lpOD");
$contexte->set_subject("Présentation de lpOD-Perl");
$contexte->set_description("Pas de commentaire");
```

Pour la rubrique « *Mots-clés* », les choses sont très légèrement plus élaborées, puisqu'il s'agit en réalité d'une liste. C'est l'interface visuelle du traitement de texte qui présente cette liste sous une forme concaténée en une seule ligne, avec des virgules insérées comme séparateurs. lpOD, assez accommodant, fournit deux méthodes `set_keywords()` et `set_keyword()` dont la combinaison autorise le programmeur à fournir les mots-clés, au choix, sous la forme perçue par l'utilisateur final (en une seule chaîne qui sera automatiquement découpée sur les virgules) ou un par un. La première forme est la plus compacte :

```
$contexte->set_keywords("OpenDocument, Perl, XML");
```

La seconde est plus « chirurgicale » :

```
$contexte->set_keyword("OpenDocument");
$contexte->set_keyword("Perl");
$contexte->set_keyword("XML");
```

Et on peut combiner les deux :

```
$contexte->set_keywords("OpenDocument, Perl");
$contexte->set_keyword("XML");
```

Grâce à `check_keyword()`, il est facile de déterminer si un mot-clé est enregistré ou non dans un document. Pour illustrer une utilisation possible, la boucle ci-dessous explore tous les documents ODF (*odt*, *ods*, *odp*, etc.) du répertoire courant et compte ceux dont les métadonnées contiennent le mot-clé « *lpOD* » :

```
my $compte = 0;
foreach my $f (glob "*.od?") {
    $compte++ if odf_document->get($f)->meta->check_keyword("lpOD");
    }
say "Trouvé $compte documents sur lpOD";
```

Les accesseurs `get_xxx()` permettent de lire les valeurs courantes des métadonnées. La ligne ci-dessous (qui pourrait constituer un programme à elle seule) affiche le titre d'un document dont le nom est passé en argument de ligne de commande :

```
say odf_document->get($ARGV[0])->meta->get_title;
```

3.3.2 Dates, versions, auteurs

Parmi les métadonnées les plus sollicitées figurent la date de création du document, sa date de dernière modification, et le nombre de révisions. Ces données sont initialisées et mises à jour par les logiciels bureautiques. Avec lpOD, elles peuvent être consultées et mises à jour n'importe quand (ou jamais), selon la volonté discrétionnaire du programmeur (qui peut même choisir une date de création postérieure à la date de dernière modification).

L'instruction suivante affiche la date de dernière mise à jour d'un document :

```
say $doc->meta->get_modification_date;
```

Attention, les variables de dates (qu'il s'agisse ou non de métadonnées) sont toujours retournées par les méthodes de lpOD en format ISO. Au besoin, il appartient aux applications de les traduire dans tel ou tel format local. Cependant, lpOD fournit deux fonctions utilitaires `iso_date()` et `numeric_date()` facilitant les conversions entre dates ISO et dates numériques Perl. Si par exemple nous voulions savoir combien de secondes se sont écoulées entre la création d'un document et l'instant présent, nous pourrions écrire :

```
say time - numeric_date($doc->meta->get_creation_date);
```

Les modificateurs `set_creation_date()` et `set_modification_date()`, appelés sans argument, enregistrent par défaut la date courante. En conséquence, pour enregistrer la vraie date de dernière mise à jour dans un programme, il suffit d'écrire, juste avant `save()`, l'instruction suivante :

```
$doc->meta->set_modification_date;
```

Le numéro de révision du document, accessible via `get_editing_cycles()` et `set_editing_cycles()`, peut aussi être mis à jour sans argument. En effet, par défaut `set_editing_cycles()` incrémente de 1 le numéro de révision.

Des accesseurs `get_xxx()` et `set_xxx()` sont également disponibles pour les propriétés `initial_creator` et `creator` qui, en jargon ODF, désignent respectivement l'auteur de la première version et celui de la dernière mise à jour. Les accesseurs `set_xxx()` correspondant à ces deux propriétés permettent en réalité de mettre n'importe quoi ; cependant, s'ils sont appelés sans argument, ils choisissent le nom de l'utilisateur courant.

On peut aussi, via les accesseurs `get_generator()` et `set_generator()`, consulter et modifier une signature applicative (généralement invisible pour l'utilisateur final mais inscrite dans le fichier). Cette signature peut par exemple, dans un document créé avec *LibreOffice 3.5* sous Linux, ressembler à ceci :

```
LibreOffice/3.5$Linux_X86_64 LibreOffice_project/350m1$Build-2
```

Appelé sans argument, `set_generator()` place une signature indiquant simplement le nom du programme en cours d'exécution.

En pratique, cette propriété « générateur » (qui n'est pas vue par l'utilisateur final) permet à chaque application créant un document de passer un message aux applications qui vont éventuellement le retraiter par la suite.

3.3.3 Métadonnées spécifiques

Le standard ODF donne à l'utilisateur la possibilité de compléter la description d'un document par des métadonnées spécifiques typées. Il s'agit en quelque sorte de variables persistantes, chacune ayant un nom, un type et une valeur.

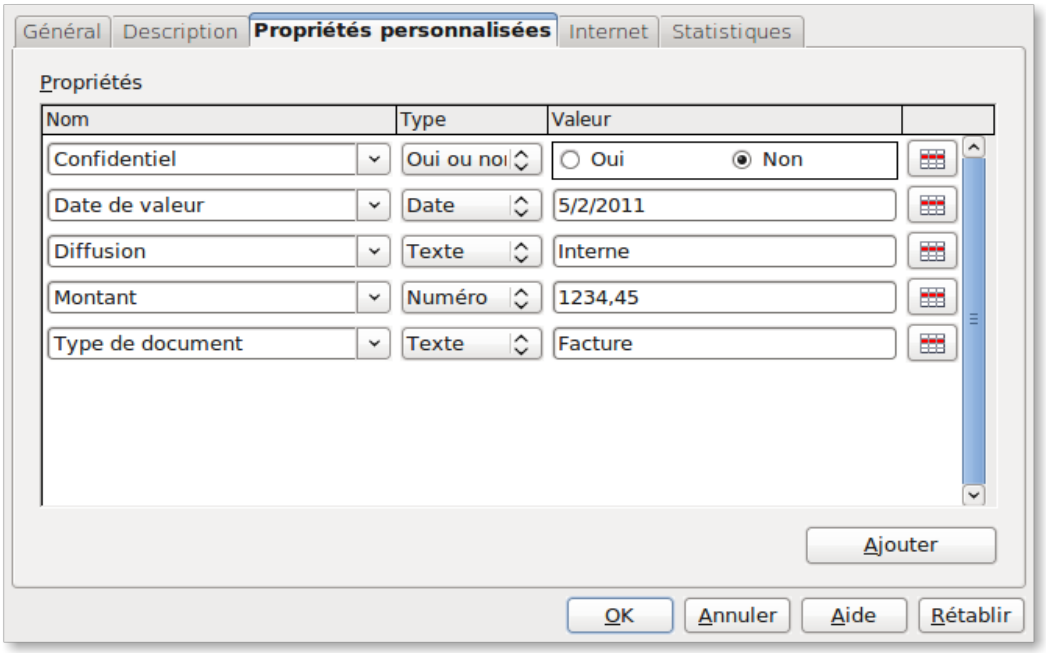
lpOD permet de créer, de consulter ou de modifier facilement ces informations qui, pour l'utilisateur final d'un logiciel bureautique, sont accessibles à travers l'onglet *Propriétés personnalisées* sous *Fichier/Propriétés*.

L'exemple suivant permet de renseigner par programme des propriétés personnalisées de différents types.

```
my $meta = $doc->meta;
$meta->set_user_field("Confidentiel", FALSE, "boolean");
$meta->set_user_field("Date de valeur", "2011-05-02", "date");
$meta->set_user_field("Diffusion", "Interne", "string");
$meta->set_user_field("Montant", 1234.45, "float");
$meta->set_user_field("Type de document", "Facture", "string");
```

Cette séquence utilise la méthode `set_user_field()` du contexte des métadonnées. Cette méthode attend trois arguments, à savoir le *nom* (au libre choix de l'utilisateur mais unique) de la propriété, la *valeur* associée et le *type* de valeur. Le troisième argument peut être omis ; dans ce cas lpOD sélectionne le type `string` par défaut.

Le résultat de cette opération (après enregistrement du document), dans l'interface d'un outil bureautique, ressemble à la copie d'écran suivante.



Dans une propriété de type `string`, on peut inscrire tout ce qu'on veut. Pour ce qui concerne le type présenté dans l'interface graphique comme « *oui ou non* », c'est-à-dire le `boolean`, lpOD considère comme synonyme de la valeur « *non* » toute valeur fausse au sens de Perl ainsi que les chaînes de caractères `"false"`, `"no"`, `"off"` et bien sûr la constante `FALSE`, les autres valeurs vraies au sens de Perl étant traduites comme la valeur « *oui* » (la valeur effectivement stockée est toujours une chaîne `"true"` ou `"false"`). Pour ce qui concerne les propriétés de type `date`, la valeur peut être passée au choix en format ISO (comme dans notre exemple) ou sous la forme d'une date numérique Perl (retournée notamment par `time`). Cela dit, il est parfaitement possible de se passer du typage et de stocker toutes sortes de propriétés sous le type `string`.

La recherche d'une propriété dans un document se fait par `get_user_field()` en donnant comme argument le nom de la propriété (toujours dans le contexte `META`). Si la propriété n'existe pas, cette méthode retourne simplement `undef`. Sinon elle retourne la valeur. En appelant `get_user_field()` en contexte de liste on obtient la valeur et le type, comme dans l'exemple suivant, qui affiche la valeur et le type d'une propriété « *Montant* » si elle est définie.

```
my ($fichier, $propriete) = @ARGV;
my ($valeur, $type) =
    odf_document->get($fichier)
                ->meta
                ->get_user_field($propriete);
say "La variable $propriete est de type $type et vaut $valeur"
    if defined $valeur;
```

4 Traitement des objets textuels

Les contenus textuels d'un document appartiennent (à de rares exceptions près) à des objets de la classe `ODF::lpOD::Paragraph`, alias `odf_paragraph`. Certains appartiennent à la classe `ODF::lpOD::Heading`, alias `odf_heading`, qui est une sous-classe de `odf_paragraph` : il s'agit des *titres*, un titre possédant, en plus des autres paragraphes, une propriété `level` qui indique son niveau hiérarchique dans le document. Les paragraphes sont omniprésents, même quand il n'y paraît pas. Ainsi, dans une feuille de tableur, dès lors que du texte apparaît dans une cellule, cela signifie que la cellule contient un ou plusieurs paragraphes. De même dans un support de présentation, une liste à puces est un mode d'organisation appliqué à une collection de paragraphes. Pour (presque) tout savoir sur les paragraphes, la page de manuel correspondante est `ODF::lpOD::TextElement`, mais voyons ici quelques fonctionnalités basiques et indispensables.

4.1 Recherche de paragraphes

Selon les circonstances, on peut choisir entre plusieurs moyens pour retrouver un paragraphe ou une liste de paragraphes. Les plus simples passent par la méthode `get_paragraph()` et ses diverses options.

On peut appeler cette méthode à partir de n'importe quel contexte, mais tous ne sont pas pertinents. Le contexte le plus fréquent est le corps du document, et nous nous en tiendrons le plus souvent à celui-là dans nos exemples, mais il est tout à fait normal d'appeler `get_paragraph()` dans un contexte plus restrictif (par exemple une section, ou un entête de page).

L'exemple ci-dessous, sans paramètre de recherche, ne fait que retourner le premier paragraphe du corps du document (s'il y en a un) :

```
my $doc = odf_document->get("article.odt");
my $contenu = $doc->get_body;
my $p = $contenu->get_paragraph;
```

On peut également sélectionner un paragraphe selon son numéro de séquence dans l'ordre du contexte, spécifié via un paramètre optionnel `position`. Attention, les numéros commencent (comme dans une liste) par zéro. Donc l'instruction suivante récupère le *cinquième* paragraphe du contexte :

```
$p = $contenu->get_paragraph(position => 4);
```

Les positions peuvent être comptées à l'envers, à partir de la fin du contexte ; il suffit pour cela des les exprimer en valeurs négatives, sachant que `-1` est la position du dernier paragraphe.

Un paragraphe peut être sélectionné d'après son contenu. Le paramètre approprié est alors `content`, sa valeur étant le filtre de recherche. Ainsi, cette instruction retourne le premier paragraphe du contexte contenant la chaîne « *ODF* » :

```
$p = $contenu->get_paragraph(content => "ODF");
```

Sachant que plusieurs paragraphes peuvent contenir la chaîne recherchée, et que ce n'est pas forcément toujours le premier qui nous intéresse, on peut combiner les options `content` et `position` de manière, par exemple, à obtenir le cinquième paragraphe contenant la chaîne recherchée (s'il y en a au moins cinq) :

```
$p = $contenu->get_paragraph(content => "ODF", position => 4);
```

Un troisième critère de recherche est utilisable : le *style*. Nous verrons plus loin comment gérer les styles de paragraphes, mais on peut d'ores et déjà considérer que, dès lors qu'un paragraphe possède un style (ce qui est le cas la plupart du temps), le nom de ce style peut être un paramètre de recherche, au moyen de l'option `style`, qui peut se combiner avec les autres. Ainsi, l'instruction suivante retrouve (s'il existe) le cinquième paragraphe parmi ceux qui contiennent « *ODF* » et dont le style est « *Code* » :

```
$p = $contenu->get_paragraph(
     style => "Code", content => "ODF", position => 4
     );
```

Notez bien qu'il est également possible d'extraire des listes de paragraphes. Pour cela, on utilise une méthode `get_paragraphs()` au lieu de `get_paragraph()`. Les options de recherche sont les mêmes, sauf le paramètre `position` qui implique un paragraphe unique. Logiquement, `get_paragraphs()` utilisé sans paramètre retourne la liste complète des paragraphes du contexte.

Ces deux méthodes sélectionnent uniquement les paragraphes simples, à l'exclusion des *titres*. La recherche des titres est réservée à `get_heading()` et `get_headings()` qui fonctionnent respectivement comme `get_paragraph()` et `get_paragraphs()`, à une importante différence près : l'option `style` est remplacée par une option `level`, dont la valeur est un entier positif indiquant le niveau hiérarchique de la recherche. L'exemple suivant ramène tous les titres de niveau 1 (le plus élevé dans la hiérarchie) contenant la chaîne « ODF » :

```
@titres = $contenu->get_headings(level => 1, content => "ODF");
```

Bien sûr, sans option de recherche, `get_heading()` retourne le premier titre trouvé dans le contexte, et `get_headings()` retourne tous les titres dans l'ordre du contexte, quels que soient le niveau et le contenu.

4.2 Lecture, modification et suppression d'un paragraphe

La caractéristique la plus importante d'un paragraphe est généralement son contenu textuel. Dès lors qu'un objet a été sélectionné par `get_paragraph()` ou `get_heading()`, ce contenu peut être extrait par `get_text()`, une méthode générique applicable à toutes sortes d'éléments. L'instruction suivante affiche donc le contenu du dernier paragraphe d'un document quelconque :

```
say odf_document->get("quelconque.odt")
                ->get_body
                ->get_paragraph(position => -1)
                ->get_text;
```

Si on veut être absolument certain que `get_text()` extraira la totalité du contenu d'un paragraphe, il est préférable d'appeler cette méthode avec le paramètre optionnel `recursive` et la valeur `TRUE`. En effet, un paragraphe peut contenir une imbrication d'autres objets contenant une partie de son texte (nous y reviendrons), et cette option permet de forcer la prise en compte non seulement du texte contenu dans l'objet appelant mais aussi de celui de tous les contenus des objets inclus (concaténés dans le bon ordre). La forme complète ressemble donc à ceci :

```
$texte = $paragraphe->get_text(recursive => TRUE);
```

Pour remplacer le contenu existant, ou pour le créer, le paragraphe dispose aussi d'une méthode `set_text()` :

```
$paragraphe->set_text("Nouveau texte");
```

Attention, `set_text()` écrase tout contenu antérieur et supprime tous les objets éventuellement imbriqués à l'intérieur du paragraphe.

Comme on l'a vu dès le début de cet article, on peut assigner du texte à un paragraphe lors de sa création. En effet, les deux séquences ci-dessous sont équivalentes :

```
$p = odf_paragraph->create;
$p->set_text("Texte du paragraphe");
```

```
$p = odf_paragraph->create(text => "Texte du paragraphe");
```

S'il s'agit d'un titre, on peut aussi consulter et modifier son niveau hiérarchique avec `get_level()` et `set_level()`. D'ailleurs, le constructeur de titre permet de spécifier ce niveau dès la création avec un paramètre `level`. Ainsi, la création d'un titre de niveau 3 pourrait s'écrire ainsi :

```
$t = odf_heading->create(text => "Le titre", level => 3);
```

Pour terminer cette rubrique, il faut bien sûr évoquer la suppression d'un paragraphe. Cependant, à cet égard, nous n'avons pas besoin d'une méthode spécifique pour les objets `odf_paragraph` et `odf_heading` : la classe `ODF::lpOD::Element`, alias `odf_element`, dont héritent tous les objets contenus dans un document, fournit une méthode générique `delete()`, applicable notamment aux paragraphes. La suppression d'un paragraphe ou d'un titre préalablement sélectionné est donc on ne peut plus simple :

```
$paragraphe->delete;
```

Ainsi, l'instruction suivante supprime tous les paragraphes qui contiennent la chaîne « ERREUR » :

```
$_->delete for $doc->get_body->get_paragraphs(content => "ERREUR");
```

4.3 Insertion d'un paragraphe dans un document

Cette rubrique, bien que focalisée sur les paragraphes (dont les titres), va nous permettre de présenter les méthodes génériques d'insertion d'un élément dans un contexte, qui fonctionnent en réalité pour toutes sortes d'éléments dans toutes sortes de contextes.

Nous avons déjà vu comment créer un paragraphe et l'attacher à un document vide. Dans le monde réel, cependant, nous aurons souvent besoin d'insérer des paragraphes au début, à la fin ou au milieu de contenus existants. La méthode `insert_element()`, que nous avons déjà rencontrée dans son application la plus simple, est heureusement paramétrable. Par défaut, elle attache l'objet donné en argument en première position dans le contenu du contexte (on appelle contexte l'élément appelant). Ainsi, de manière générale, l'objet `$o` (quel qu'il soit) devient le premier élément du contenu de `$c` (quel que soit `$c`) après exécution de l'instruction suivante :

```
$c->insert_element($o);
```

Mais nous avons une option `position` qui, tout en utilisant l'élément appelant comme point de repère, permet de choisir entre quatre positions relatives, représentées par les constantes symboliques `FIRST_CHILD`, `LAST_CHILD`, `NEXT_SIBLING`, `PREV_SIBLING`. La valeur par défaut de cette option est `FIRST_CHILD` : elle

spécifie en effet que l'élément inséré doit l'être comme « premier enfant », c'est-à-dire en tête du contenu, de sorte que ces deux instructions sont équivalentes :

```
$c->insert_element($o, position => FIRST_CHILD);
$c->insert_element($o);
```

Si `position` vaut `LAST_CHILD`, l'élément est inséré à la dernière position du contenu. (Évidemment, s'il s'agit d'un contexte vide, la première et la dernière position sont confondues et le paramètre `position` est inutile.)

L'ajout d'un élément (par exemple un paragraphe) à la suite du contexte étant une opération assez courante, il existe un raccourci syntaxique `append_element()` tel que les deux instructions suivantes sont équivalentes :

```
$c->insert_element($o, position => LAST_CHILD);
$c->append_element($o);
```

Si la valeur de `position` est `PREV_SIBLING` ou `NEXT_SIBLING`, l'élément rattaché est placé non plus au début ou à la fin du contexte, mais juste avant ou juste après lui. Naturellement, si le contexte est, par exemple, le corps du document, c'est-à-dire un objet récupéré par `get_body()`, il serait étrange d'insérer un paragraphe en dehors de lui (ce paragraphe serait invisible et la structure du document ne serait plus conforme au standard ODF).

Pour bien illustrer le principe, prenons le cas de l'ajout d'un paragraphe à la fin du contexte (pour simplifier, nous prendrons le corps du document comme contexte, donc disons que nous allons insérer un paragraphe à la fin du document). Nous pouvons l'écrire (au moins) de deux façons.

La première manière consiste à repérer le dernier paragraphe, puis à l'utiliser comme point d'appui pour insérer le nouveau paragraphe *après lui* :

```
$dernier = $doc->get_body->get_paragraph(position => -1);
$dernier->insert_element(
    odf_paragraph->create(text => "C'est le dernier"),
    position => NEXT_SIBLING
    );
```

La seconde manière (qui dans ce cas précis est évidemment la meilleure) consiste à insérer directement le nouveau paragraphe comme le dernier du contexte :

```
$doc->get_body->insert_element(
    odf_paragraph->create(text => "C'est le dernier"),
    position => LAST_CHILD
    );
```

Mais il vaut encore mieux utiliser `append_element()` qui, rappelons-le, est un raccourci pour `insert_element()` avec la position `LAST_CHILD` :

```
$doc->get_body->append_element(
    odf_paragraph->create(text => "C'est le dernier")
    );
```

Comme exercice récapitulatif, le programme ci-après, qui est un convertisseur extrêmement primaire de texte plat en texte ODF, absorbe un fichier de texte dont il transforme chaque ligne en un paragraphe.

```
my $doc = odf_document->create('text');
my $contenu = $doc->get_body;
$contenu->clear;
open SOURCE, '<', $ARGV[0];
while (my $ligne = <SOURCE>) {
    chomp $ligne;
    $contenu->append_element(
          odf_paragraph->create(text => $ligne)
          );
    }
close SOURCE;
$doc->save(target => $ARGV[1]);
```

Ce programme ne produit bien sûr que des paragraphes « plats ». Mais nous verrons plus loin comment ajouter de la décoration dans le texte.

4.4 Copie de paragraphes

Tout objet `odf_element`, et donc notamment tout paragraphe, possède une méthode `clone()` qui, comme son nom l'indique, retourne une copie intégrale de l'appelant. Cette copie est en l'occurrence un paragraphe possédant le même style et le même contenu que l'original, mais elle est dans la même situation qu'un paragraphe nouvellement fabriqué par `create()` : elle n'est pas encore attachée à un document, mais elle peut l'être par les moyens habituels, à savoir `insert_element()` ou `append_element()`. L'exemple suivant illustre une utilisation possible : il alimente un document `$doc2` avec des copies de tous les paragraphes de style « *Code* » trouvés dans un document `$doc1` :

```
$doc2->get_body->append_element($_->clone)
    for $doc1->get_body->get_paragraphs(style => "Code");
```

Cette instruction ne modifie rien dans le contexte source, et les copies de paragraphes une fois créées sont complètement indépendantes des originaux[8]. D'autre part les éléments copiés peuvent être insérés aussi bien dans un autre contexte du même document que dans un autre document.

Les copies par `clone()` ne peuvent être utilisées que pendant la vie du programme en cours d'exécution. Il est toutefois possible avec lpOD d'extraire des copies d'éléments sous forme d'exports XML que les applications peuvent enregistrer dans des fichiers persistants et/ou se transmettre à distance. La méthode `export()` (également appelée `serialize()`) facilite ces opérations, comme dans l'exemple suivant qui enregistre une copie XML du dernier paragraphe d'un document :

```
$p = $doc->get_body->get_paragraph(position => -1);
open(my $f, '>:utf8', "paragraphe.xml");
print $f $p->export;
close $f;
```

Notez bien que, dans cet exemple, le fichier de sortie est ouvert en mode `utf8`. C'est impératif notamment si le texte du paragraphe peut contenir des lettres accentuées. Quant à la méthode `export()`, elle peut être appelée avec une option `pretty` qui, si sa valeur est `TRUE`, produira un XML plus lisible, ce qui peut être utile en phase de test.

Les exports XML générés par `export()`, après stockage et/ou transport par un moyen quelconque, peuvent ensuite être importés via le constructeur standard `create()` de la classe de base `odf_element`. Ce constructeur, en effet, est capable de reconstituer un élément quelconque à partir de sa description XML. Ainsi, la séquence ci-après récupère le paragraphe exporté par l'exemple précédent et le place à la fin d'un document.

8 À ne pas confondre avec les méthodes `move()` et `group()`, qui permettent de *déplacer* des éléments d'un contexte à un autre.

Quand on appelle le constructeur `create()` de la classe de base `odf_element` avec en argument une chaîne terminée par « *.xml* », cet argument est considéré comme le nom d'un fichier contenant la description XML de l'élément à créer, et traité en conséquence :

```
$doc2->get_body->append_element(
    odf_element->create("paragraphe.xml")
    );
```

On peut aussi importer des éléments disponibles en XML dans des dépôts distants :

```
$doc->get_body->append_element(
    odf_element->create('http://quelque.part.ailleurs/paragraphe.xml')
    );
```

À retenir : ces divers moyens de copier un élément d'un contexte à un autre de manière immédiate ou différée, présentés ici à propos des paragraphes, sont communs à toutes sortes d'éléments.

4.5 Création et utilisation de styles dans les paragraphes

4.5.1 Styles de paragraphes

Un paragraphe qui n'a pas de style particulier est représenté d'après un style de paragraphe par défaut, défini une fois pour toutes dans le document. Cependant, il est souvent préférable de choisir un style explicite. Cela peut se faire de deux manières.

Dans tous les cas, l'attribution d'un style à un objet se fait par l'intermédiaire d'un *nom*. Chaque style appartient à une *famille* qui correspond aux types d'objets auxquels il peut s'appliquer et possède un *nom* unique pour cette famille.

On peut tout d'abord affecter un style à un paragraphe dès sa création, grâce au paramètre optionnel `style` :

```
$p = odf_paragraph->create(text => "XYZ", style => "Bordé");
```

Autrement, il est toujours possible de consulter et de changer le style d'un objet existant à l'aide de `get_style()` et `set_style()` :

```
say "Le style actuel est " . $p->get_style;
$p->set_style("Bordé");
```

Pour que ces exemples fonctionnent, il faut que, au moment où le document sera affiché ou imprimé via une application ODF, un style de paragraphe nommé «*Bordé*» ait été défini dans ce document.

Il est recommandé d'utiliser autant que possible des modèles de documents comprenant des styles déjà créés à travers un logiciel bureautique ; cela vaut généralement mieux que de consacrer des centaines de lignes de Perl à définir des styles. Cependant, lpOD permet de créer des styles à partir de rien quand c'est nécessaire.

Pour créer un style, il existe naturellement plusieurs moyens… et plusieurs styles de programmation. Ici, nous allons choisir un exemple de style de paragraphe représentatif sans être trop compliqué, et un style de code bien décomposé à des fins didactiques. Pour aller plus loin, consultez la page de manuel `ODF::lpOD::Style`. Nous prendrons comme objectif un style de paragraphe répondant au « cahier des charges » suivant :

- des marges gauche et droite de *4 mm* et un espacement de *2 mm* au-dessus ;
- une bordure pointillée *bleu marine* d'une épaisseur de *1 mm* ;

– un espace minimal de *5 mm* entre la bordure et le texte ;

– un *arrière-plan jaune* ;

– pour le texte, des caractères *gras* de *12 points* de *couleur bleu marine*.

Créons d'abord un nouveau style de paragraphe. Pour cela, nous allons utiliser le constructeur `create()` de la classe `ODF::lpOD::Style`, alias `odf_style`, en lui spécifiant la famille et le nom du nouveau style :

```
my $s = odf_style->create('paragraph', name => 'Bordé');
```

Remarquez que la famille de style, ici '`paragraph`', est l'argument obligatoire, tandis que le nom propre du style est passé par un paramètre optionnel `name`, car il est permis de créer un style sans lui donner immédiatement de nom (voir le manuel si vous voulez savoir pourquoi).

Nous pouvons ensuite attacher ce style à un document. Attention, n'utilisons surtout pas `insert_element()` ou `append_element()`, car un style ne doit pas être inséré dans un contexte particulier ; il doit être enregistré au moyen de la méthode spéciale `register_style()` définie pour l'objet `odf_document` :

```
$doc->register_style($s);
```

Cette méthode, qui possède un alias `insert_style()`, peut être paramétrée de manière à offrir un contrôle assez fin de la gestion des styles, mais tant que vous en êtes en phase exploratoire avec lpOD le comportement par défaut devrait généralement convenir. Par ailleurs, un objet `odf_style` possède une méthode `register()` qui lui permet de s'enregistrer lui-même dans un document passé en argument, de sorte que l'instruction précédente pourrait être remplacée par ceci :

```
$s->register($doc);
```

Le style est à présent dûment enregistré… mais il n'a encore aucune caractéristique définie. Il reste donc à le paramétrer. Nous avons pour cela une méthode `set_properties()`, mais avant de l'utiliser il faut comprendre qu'un style de paragraphe comprend deux sortes de propriétés bien distinctes correspondant chacune à ce qu'on appelle, dans lpOD, une `area`. La première `area`, nommée '`paragraph`' comme la famille du style, concerne l'enveloppe externe, ou si l'on veut les caractéristiques de la « boîte » (notamment les marges et bordures). La seconde `area`, nommée '`text`', concerne la présentation du texte contenu dans le paragraphe.

Commençons par la décoration « externe » : il faut appeler `set_properties()` en spécifiant l'`area` '`paragraph`' et en donnant les paramètres appropriés :

```
$s->set_properties(
    area             => 'paragraph',
    margin_left      => '4mm',
    margin_right     => '4mm',
    margin_top       => '2mm',
    border           => '1mm dotted #000080',
    padding          => '5mm'
    );
```

Certes, les paramètres ne se devinent pas. Il vous faudra d'abord éplucher la page de manuel `ODF::lpOD::Style` et ensuite, peut-être, investiguer dans la norme OpenDocument elle-même pour connaître toutes les possibilités. Ici, les paramètres relatifs aux marges sont à peu près autodocumentés. '`padding`' spécifie l'espace entre bordure et contenu. L'option '`border`' est plus complexe : sa valeur est une chaîne contenant en réalité trois paramètres, à savoir d'abord l'épaisseur de la ligne, ensuite le style de ligne (par exemple `dotted` pour pointillé ou `solid` pour continu), et enfin le code RVB[9] de la couleur choisie.

9 Code hexadécimal sur 6 positions, préfixé par un '#', et indiquant successivement les quantités de rouge, de vert et de bleu.

Croyez-moi sur parole, le code `#000080` correspond au bleu marine. Cela dit lpOD fournit une fonction utilitaire `color_code()` qui, à partir d'un nom de couleur conventionnel, donne le code RVB (s'il est connu)[10]. Ainsi, la valeur du paramètre `border` pourrait être construite comme dans l'exemple suivant.

```
my $bordure = '1mm dotted ' . color_code('navy blue');
```

Quant aux propriétés de présentation applicables au contenu, elles sont définies dans l'`area` '`text`' :

```
$s->set_properties(
    area             => 'text',
    weight           => 'bold',
    size             => '12pt',
    color            => 'navy blue'
    );
```

On notera ici les options `weight` et `size`, spécifiant respectivement le « poids » et la taille des caractères. L'option `color`, comme d'habitude, accepte au choix un code RVB ou (s'il existe et si on le connaît) le nom symbolique correspondant. Car, pour les options `color` de divers types d'objets comme les styles, lpOD appelle automatiquement `color_code()` pour trouver le code correspondant si la valeur donnée est un nom au lieu d'un code préfixé par '`#`'. Il se trouve que '`navy blue`' fait partie des quelques dizaines de noms de couleurs prédéfinis par défaut.

Concernant la couleur d'arrière-plan, nous pouvons utiliser une méthode générique `set_background()` applicable aux styles de paragraphes comme à d'autres objets pour lesquels la notion d'arrière-plan a un sens. Ici nous devons définir un arrière-plan dont le seul paramètre est la couleur jaune. Facile :

```
$s->set_background(color => 'yellow');
```

La séquence complète de création et d'intégration de notre style de paragraphe peut donc, parmi de très nombreuses possibilités, s'écrire comme ceci :

```
my $s = odf_style->create('paragraph', name => "Bordé");
$s->set_properties(
    area        => 'paragraph',
    margin_left      => '4mm',
    margin_right     => '4mm',
    margin_top       => '2mm',
    border           => '1mm dotted ' . color_code('navy blue'),
    padding          => '5mm'
    );
$s->set_properties(
    area             => 'text',
    weight           => 'bold',
    size             => '12pt',
    color            => 'navy blue'
    );
$s->set_background(color => 'yellow');
$doc->register_style($s);
```

Voici donc un style « *Bordé* » parfaitement défini.

Supposons maintenant que nous ayons besoin d'un style nommé, disons, « *BordéCentré* », possédant toutes les caractéristiques de « *Bordé* », mais avec, en plus, une présentation *centrée*. Dans ce cas il n'est pas nécessaire de réécrire entièrement ce que nous avons déjà spécifié pour le premier style. Il existe entre styles une

10 La table de couleurs par défaut est basée sur le standard *rgb.txt* de l'environnement *Xorg*. Elle peut être personnalisée par l'utilisateur.

possibilité d'héritage permettant de s'appuyer sur la définition d'un style existant. La définition et l'intégration du style « *BordéCentré* » peut se faire de la façon suivante.

```
my $s2 = odf_style->create(
    'paragraph',
    name              => "BordéCentré",
    parent            => "Bordé"
    );
$s2->set_properties(
    area              => 'paragraph',
    align             => 'center'
    );
$doc->register_style($s2);
```

Nous avons ici créé un nouveau style dont l'option `parent` indique qu'il hérite de toutes les propriétés du style « *Bordé* », à l'exception des propriétés qui lui sont spécifiques et qui sont définies par `set_properties()`. On notera au passage que tout ce qui concerne l'alignement du texte (centrage, justification, etc.) est spécifié dans la zone '`paragraph`'.

On pourrait abréger l'exemple ci-dessus, la séquence suivante produisant le même résultat :

```
odf_style->create(
    'paragraph',
    name              => "BordéCentré",
    parent            => "Bordé",
    align             => 'center'
    )->register($doc);
```

Ici, l'option `align` a été passée directement comme paramètre du constructeur sans passer par `set_properties()`. Cette facilité est basée sur le principe suivant : lpOD considère tout paramètre qui n'est pas reconnu (comme `name` ou `parent`) en tant que l'une des caractéristiques « identitaires » du style comme une propriété appartenant à une `area` dont le nom est celui de la famille de style. Donc ici le paramètre `align` est automatiquement affecté à la zone '`paragraph`'.

4.5.2 Styles de texte

Nous avons vu comment créer et utiliser un style applicable globalement à un paragraphe. Mais ce n'est pas toujours suffisant pour un document en texte « riche ».

Prenons comme exemple un paragraphe `$p` dont l'image à l'écran serait celle-ci :

ODF est un *excellent* format documentaire

Il s'agit d'un paragraphe dont *une partie du texte* est présentée en italiques sur fond jaune. Pour obtenir ce résultat, nous devons d'une part définir un style pour zones de texte, qu'on appelle plus simplement un *style de texte* indépendamment du style éventuellement applicable au paragraphe environnant, et d'autre part appliquer ce style à une *portion* du contenu du paragraphe.

La première opération passe par le constructeur de styles habituel mais la famille du style est cette fois '`text`' et non '`paragraph`' :

```
my $st = odf_style->create('text', name => "ItaliqueFondJaune");
```

Nous pouvons ensuite spécifier des caractères italiques avec `set_properties()` .

```
$st->set_properties(style => 'italic');
```

On a utilisé ici le paramètre `style` pour spécifier les italiques. L'existence d'un paramètre nommé `style` dans la définition d'un style est peut-être « confusante » mais c'est un raccourci ; vous pouvez utiliser le nom étendu de cette propriété, soit '`fo:font-style`'[11], si vous préférez.

Mais par ailleurs, aucune `area` n'a été spécifiée. La raison est simple : un style de texte est moins sophistiqué qu'un style de paragraphe et sa définition ne comprend qu'une seule `area` ; le paramètre `area`, même s'il était fourni, serait ignoré.

Il reste à définir la couleur d'arrière-plan. Sachant que lpOD essaie d'être relativement cohérent avec lui-même, on peut simplement écrire ceci :

```
$st->set_background(color => 'yellow');
```

La préparation complète du style de texte pourrait être récapitulée ainsi :

```
my $st = odf_style->create(
    'text',
    name        => "ItaliqueFondJaune",
    style       => 'italic'
    );
$st->set_background(color => 'yellow');
$doc->register_style($st);
```

lpOD est plutôt tolérant et, pour information, voici une forme encore plus compacte et équivalente (attention, le remplacement de `set_background()` par une option `background_color` ne fonctionne pas avec tous les styles) :

```
odf_style->create(
    'text',
    name             => "ItaliqueFondJaune",
    style            => 'italic',
    background_color  => 'yellow'
    )->register($doc);
```

Nous disposons maintenant d'un style de texte. Il reste à l'appliquer. Cette application passe par la méthode `set_span()` qui définit une « étendue de texte » à l'intérieur d'un paragraphe et lui associe un style de texte. Il existe plusieurs moyens de délimiter la zone en question. Vous pouvez les découvrir dans la page de manuel `ODF::lpOD::TextElement`. Ici nous allons aller au plus simple. Dans notre paragraphe, c'est juste le mot « *excellent* », dont il n'existe qu'une seule occurrence, qui est visé, et pour ce type de situation, en supposant que le paragraphe environnant `$p` ait été préalablement sélectionné, la réponse est simple.

```
$p->set_span(filter => "excellent", style => "ItaliqueFondJaune");
```

Pour replacer cet exemple dans un contexte plus large, voici une séquence qui, dans tous les paragraphes dont le style est « *Ordinaire* », applique notre style « *ItaliqueFondJaune* » à toutes les chaînes « XYZ » :

```
$_->set_span(filter => "XYZ", style => "ItaliqueFondJaune")
    for $context->get_paragraphs(style => "Ordinaire");
```

11 Les attributs '*fo:*' sont empruntés au vocabulaire *XSL-FO*, l'un des standards auxquels se réfère la norme ODF.

5 Tableaux

Pour le traitement des tableaux, lpOD fournit une classe `ODF::lpOD::Table`, alias `odf_table`, qui représente aussi bien les feuilles de calcul de tableurs que les tableaux figurant dans les documents de type texte, présentation ou dessin. Bien entendu, les fonctionnalités d'un tableur sont beaucoup plus élaborées, mais du point de vue de lpOD l'interface de programmation est la même.

5.1 Recherche des tableaux

Un tableau existant dans un document peut être recherché de préférence d'après son nom (unique pour le document) via la méthode `get_table()`. L'instruction suivante retourne (s'il existe) le tableau « *Feuille1* » :

```
$t = $doc->get_body->get_table("Feuille1");
```

À défaut de connaître le nom, on peut toujours choisir une table d'après sa position (à partir de zéro) dans l'ordre des tables du contexte :

```
$t = $doc->get_body->get_table_by_position(0);
```

On peut aussi demander la première table dont le contenu d'une cellule au moins est conforme à une expression de recherche donnée :

```
$t = $doc->get_body->get_table_by_content("XYZ");
```

On peut enfin, à partir d'un objet précédemment sélectionné et dont on pense qu'il est à l'intérieur d'une table, retrouver la table à laquelle il appartient :

```
$t = $objet->get_parent_table;
```

D'autre part une application peut extraire la liste complète des tables d'un contexte donné via la méthode `get_tables()`.

5.2 Accès aux éléments d'un tableau

L'opération la plus banale qu'on puisse faire dans un tableau est la sélection d'une cellule en vue de la consulter ou de la modifier, via la méthode `get_cell()`. Celle-ci doit évidemment être renseignée avec les coordonnées de la cellule, qui peuvent être exprimées, au choix, dans l'un des formats suivants :

- un format numérique positif, sous la forme d'une paire de nombres supérieurs ou égaux à 0, avec le numéro de ligne en premier, le 0 désignant la première position ;
- un format numérique négatif permettant de spécifier les coordonnées à partir de la fin, toujours avec le numéro de ligne en premier, la valeur -1 désignant la dernière position ;
- un format alphanumérique (dit « bataille navale »), correspondant à une chaîne de caractères désignant la colonne par une ou plusieurs lettres et la ligne par un nombre entier positif, la première position de colonne étant le « A » et la première position de ligne étant le « 1 », comme dans l'interface graphique d'un tableur.

Ainsi, à partir d'une table `$t`, les deux instructions suivantes sélectionnent la même cellule :

```
$c = $t->get_cell(0, 0);
$c = $t->get_cell("A1");
```

Les deux instructions ci-après, elles aussi, sont équivalentes :

```
$c = $t->get_cell(9, 27);
$c = $t->get_cell("AB10");
```

À propos d'adressage des cellules, attention au piège ! Dans l'interface visuelle d'un tableur, la présentation est faite de manière à donner à l'utilisateur la sensation que chaque table possède une taille illimitée ou presque. Mais lpOD ne s'adresse pas à la visualisation de la table ; il n'a accès qu'à la taille de la structure de données correspondante telle qu'elle a été enregistrée. Si on n'y prend pas garde, on peut récolter un message « *out of range* » pour avoir adressé une cellule hors des limites de la table réelle même si les coordonnées fournies sont très en deçà des limites de la grille apparente.

Une fois acquise par `get_cell()`, une cellule peut être consultée par `get_value()` ou `get_text()`, ou modifiée par `set_value()` ou `set_text()`. Les accesseurs '`value`' visent plutôt la valeur de la cellule alors que les accesseurs '`text`' s'adressent au texte apparent. Il n'y a pas de différence dans le cas des cellules de texte, mais pour les cellules des autres types il vaut mieux utiliser `get_value()` et `set_value()` si on s'intéresse à la valeur interne (indépendamment du format de présentation). Ajoutons au passage qu'il est possible de consulter et de modifier le type lui-même, et de manipuler les formules[12].

Si nous avons seulement besoin de récupérer la valeur contenue dans une cellule, sans idée de modification ultérieure, alors il est plus efficace (surtout pour les très grandes tables) d'utiliser `get_cell_value()`, avec les mêmes principes quant à la syntaxe des coordonnées. C'est ce que fait cette instruction qui capture le contenu de la cellule en bas à droite du tableau nommé « *Compte* » :

```
$total = $doc->get_body->get_table("Compte")->get_cell_value(-1, -1);
```

Il existe aussi des méthodes `get_row()` et `get_column()` qui permettent respectivement, à partir d'une table, d'adresser respectivement une ligne et une colonne. Dans un très grand tableau, si on prévoit d'accéder successivement à plusieurs cellules d'une même ligne, on peut améliorer les performances en sélectionnant d'abord la ligne, puis en adressant les cellules à partir de la ligne. Une ligne de table, dans lpOD, est un objet `ODF::lpOD::Row`, également nommé `odf_row`, et possède lui aussi une méthode `get_cell()` qui, elle, bien sûr, n'a besoin que d'une seule dimension (qui peut être donnée sous la forme d'un groupe de lettres à partir de « A » ou d'un nombre à partir de 0 ou -1).

Remarquez qu'il est également possible d'accéder à une cellule à partir d'une colonne préalablement sélectionnée, car `get_cell()` est également une méthode de colonne (`odf_column`). Mais dans ce cas il vaut mieux éviter de jouer avec les grandes tables, car l'accès à partir des colonnes est beaucoup plus coûteux qu'à partir des lignes (physiquement, selon le modèle de données ODF, les cellules sont contenues dans les lignes et non dans les colonnes).

La sélection d'une ligne peut, comme on l'a vu, se faire par `get_row()` en indiquant simplement le numéro de ligne. C'est la solution la plus performante. Mais il est parfois utile de sélectionner une ligne dont on ne connaît pas forcément la position, d'après le contenu d'une colonne de référence utilisée comme s'il s'agissait de l'index de la table. La méthode `get_row_by_index()` permet de le faire, comme dans cet exemple qui récupère (si elle existe) la première ou la seule ligne de la table appelante qui, en colonne H, contient la valeur « *XYZ* » :

```
my $ligne = $table->get_row_by_index(H => "XYZ");
```

Une fois en possession de la ligne, on peut lui appliquer `get_cell()` ou `get_cell_value()` pour adresser n'importe quelle cellule présente sur la ligne dont la colonne *H* correspondait à la condition donnée. Notez bien que la logique de recherche est basée sur un *smart match*, ce qui veut dire que le critère de recherche peut être une référence de liste de valeurs ou une expression régulière. Attention quand même aux performances avec

12 Attention, lpOD n'interprète pas et ne contrôle pas le contenu des formules. Il appartient au programmeur qui insère ou modifie une formule dans une cellule de connaître le langage de formule du tableur auquel le document est destiné.

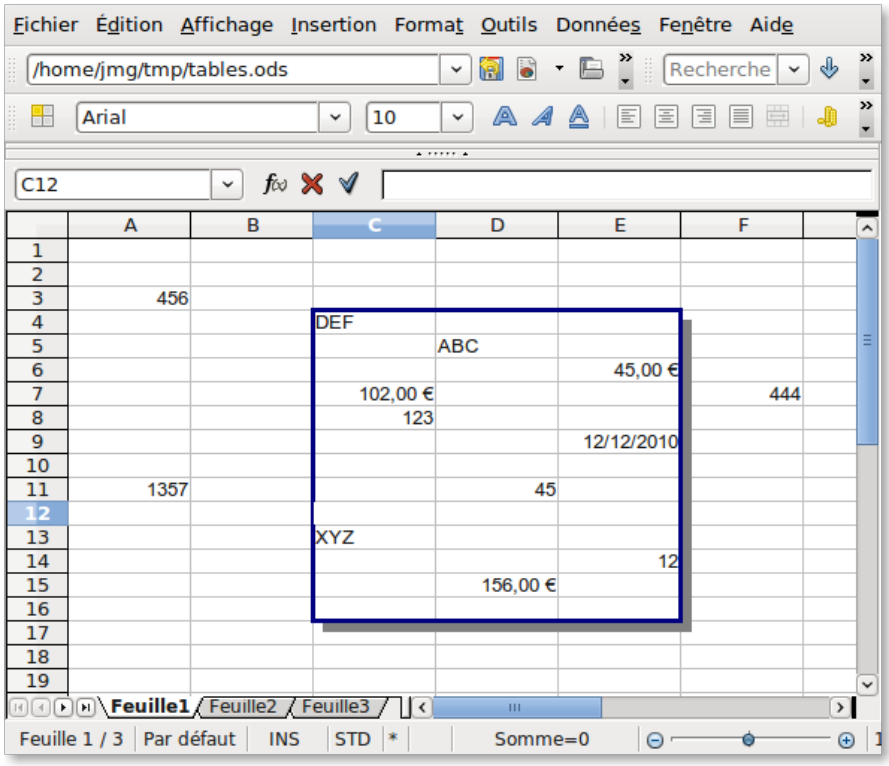
`get_row_by_index()`, car aucune colonne dans une table ODF ne bénéficie d'un accès privilégié, et lpOD effectue en réalité un balayage séquentiel pour trouver la position de la ligne recherchée.

Il existe aussi une forme plurielle de cette méthode, à savoir `get_rows_by_index()`, qui retourne la liste de toutes les lignes dont une certaine colonne est conforme à une certaine expression de recherche.

5.3 Accès à des zones de tableaux

Il est possible d'accéder à des *groupes de cellules* via les méthodes `get_cells()` et `get_cell_values()`.

Imaginons, dans une feuille de calcul arbitraire représentée par la copie d'écran ci-après, une surface rectangulaire allant de *C4* à *E16* (zone encadrée dans l'image).



La méthode `get_cells()` retourne sous la forme d'un tableau Perl à deux dimensions la liste des cellules correspondant à cette surface. Elle s'utilise comme ceci :

```
@cellules = $table->get_cells("C4:E16");
```

Ce tableau est une projection (*mapping*) de la zone sous-jacente, et non pas une collection de copies de cellules.

Ce qui veut dire, dans cet exemple, que les deux instructions suivantes produiront le même effet :

```
$cellules[0][0]->set_text("Ici C4");
$table->get_cell("C4")->set_text("Ici C4");
```

La méthode `get_cell_values()`, qui extrait un tableau de *valeurs* et non un tableau de cellules, comporte un mécanisme un peu plus sophistiqué : elle réclame, comme premier argument, un *type de donnée*. Chaque cellule de table a un certain type, le type par défaut étant `string` (texte). Les autres types sont `date`, `time`, `float`, `currency`, `percentage`, `boolean` (ils correspondent respectivement, dans la version française d'*OpenOffice.org* ou de *LibreOffice*, à ce qu'on appelle les formats « *date* », « *heure* », « *nombre* », « *monétaire* », « *pourcentage* », « *valeur logique* »). Cette méthode retourne les *valeurs* des cellules

correspondant au type indiqué et, dans le tableau 2D résultant, les positions des cellules d'autres types sont marquées par `undef`. L'instruction suivante crée donc un tableau `@valeurs` de 13×3 dont seules trois valeurs seront renseignées (car dans notre copie d'écran seules *E6*, *C7* et *D15* contiennent des valeurs monétaires) :

```
@valeurs = $table->get_cell_values('currency', 'C4:E16');
```

Il reste possible toutefois de récupérer toutes les valeurs quel qu'en soit le type en passant le mot-clé `all` à la place du type :

```
@valeurs = $table->get_cell_values('all', 'C4:E16');
```

Cependant, `get_cell_values()` peut être utilisé en contexte scalaire pour une fonctionnalité très différente. Dans ce cas, ce qui est retourné est la référence d'une liste de trois ou quatre agrégats, selon le type. Si le type recherché est `string`, alors la structure retournée contient le *nombre* de cellules conformes, la valeur *minimale* et la valeur *maximale*. Dans l'exemple ci-dessous (en se basant sur la précédente copie d'écran), `$r->[0]` contient *3*, `$r->[1]` contient « *ABC* », `$r->[2]` contient « *XYZ* », sachant qu'il y a dans la zone explorée trois cellules de texte « *DEF* », « *ABC* », « *XYZ* » :

```
$r = $table->get_cell_values('string', 'C4:E16');
```

Si le type sélectionné est un montant (float ou currency) ou un pourcentage, la structure résultante comprendra le nombre de valeurs du type donné, le minimum, le maximum et, en plus, la somme (ce qui permettra évidemment, si l'utilisateur le souhaite, de trouver facilement la moyenne). Donc dans l'exemple suivant `$r->[0]` vaut *3*, `$r->[1]` vaut *45*, `$r->[2]` vaut *156*, `$r->[3]` vaut *303* et la seconde instruction affiche « *La moyenne est 101* » :

```
$r = $table->get_cell_values('currency', 'C4:E16');
say "La moyenne est " . $r->[3] / $r->[0];
```

Indépendamment de `get_cell()` et `get_cell_values()`, lpOD permet aussi de traiter des plages de cellules nommées, autrement dit des zones rectangulaires désignées non pas par leurs coordonnées mais par des noms symboliques.

Ainsi, la méthode `get_named_range()` de `odf_document` permet de retrouver une zone nommée quelque part dans un document, créée par le tableur ou par un autre programme lpOD, sans même savoir dans quelle feuille elle se trouve :

```
my $z = $doc->get_named_range("MaZone");
```

Inversement, il est possible de *créer* une zone nommée (qui sera visible et utilisable à travers le tableur) avec la méthode `set_named_range()`, en fournissant un nom (unique), le nom de la feuille de calcul de rattachement, et le périmètre de la zone.

Le code suivant crée un espace nommé « *MaZone* » occupant la surface *C4:F12* dans la table « *Feuille4* » :

```
$doc->set_named_range(
    "MaZone", table => "Feuille4", range => "C4:F12"
);
```

Cette zone nommée sera ensuite visible à travers un tableur.

Une zone nommée (qui est un objet de classe `odf_named_range`) possède notamment ses propres méthodes `get_cells()` et `get_cell_values()`, ce qui facilite la manipulation des plages de cellules lorsqu'elles sont nommées. Ainsi, en supposant qu'il existe une zone nommée semblable à celle que définit notre dernier exemple de code, les deux instructions suivantes retournent la même chose :

```
@cellules = $doc->get_named_range("MaZone")->get_cells;
@cellules = $doc->get_table("Feuille4")->get_cells("C4:F12");
```

5.4 Création et redimensionnement de tables

Nous avons déjà vu au début de cet article comment créer un tableau, avec le constructeur `create()` de la classe `odf_table`. Il suffit juste de rappeler que, en admettant que `$hauteur` et `$largeur` correspondent respectivement au nombre de lignes et au nombre de colonnes, l'usage de ce constructeur est de la forme suivante :

```
$table = odf_table->create(
          "MaTable",
          size  => [ $hauteur, $largeur ]
          );
```

ou encore :

```
$table = odf_table->create(
          "MaTable",
          length      => $hauteur,
          width       => $largeur
          );
```

D'autres paramètres sont bien sûr possibles, à découvrir dans le chapitre de manuel `ODF::lpOD::Table`.

Concernant la taille, il convient de faire une remarque importante. Quand un utilisateur de traitement de texte crée un tableau au milieu d'une page, il paraît naturel de définir sa taille avant d'y placer des données. Mais avec un tableur, chaque feuille de calcul semble avoir une taille pour ainsi dire illimitée[13]. Cette grille illimitée est cependant un artifice de l'interface graphique ; la structure de données interne est en réalité limitée, et c'est le tableur qui ajuste sa taille selon la position de la dernière cellule utilisée. Avec lpOD, on travaille sur la structure de données interne, enregistrée en XML dans le fichier, et non avec la grille de présentation du tableur. La taille réelle de cette structure peut être consultée avec la méthode `get_size()`, qui (en contexte de liste) retourne le nombre de lignes et le nombre de colonnes[14]. Et si d'aventure on tentait d'accéder par `get_cell()` à une cellule située au-delà des limites indiquées par `get_size()` on n'obtiendrait rien d'autre qu'un message d'erreur « *out of range* ».

Mais ce qui est transparent pour l'utilisateur d'un tableur peut être obtenu explicitement par programme avec lpOD. Grâce aux méthodes `add_row()` et `add_column()`, on peut insérer des lignes ou des colonnes supplémentaires à tout moment.

L'exemple suivant, qui convertit un fichier CSV[15] en un fichier ODS à un seul onglet, est un exemple d'extension d'une table ligne par ligne alimentée par une source de données externe. Pour traiter les données sources sans réinventer la roue, cet exemple se sert de `Text::CSV`, dont je vous invite à consulter la documentation si vous voulez savoir ce que fait la méthode `getline()`. La lecture du premier enregistrement du fichier avant d'entrer dans la boucle sert uniquement à déterminer le nombre de champs du CSV, de manière à connaître le nombre de colonnes de la table à créer (on suppose qu'il s'agit d'un CSV régulier, où le nombre de champs par enregistrement est toujours le même). La table est initialisée avec une seule ligne et un nombre de colonnes égal au nombre de champs. Ensuite, après chaque lecture d'un nouvel enregistrement dans le fichier, une nouvelle ligne est ajoutée au tableau par `add_row()`. Le traitement s'arrête à la première ligne vide.

13 Ou plus exactement une taille qui dépend des performances du tableur, par exemple 1000000×1024 dans *OpenOffice.org 3.3*.
14 En réalité, cette méthode retourne, en guise de nombre de colonnes, le nombre de cellules de la plus longue ligne du tableau, pour le cas où par malchance on tomberait sur un tableau irrégulier.
15 CSV = *Comma Separated Values*. Format d'enregistrement de tables sous la forme d'un enregistrement par ligne, les champs étant séparés par des virgules ou des points-virgules.

```
use ODF::lpOD;
use Text::CSV;

my $fichier_csv = $ARGV[0];
my $fichier_ods = $ARGV[1];

# accès au fichier, lecture du premier enregistrement
my $source = Text::CSV->new;
open my $FH, "<", $fichier_csv;
my $l = $source->getline($FH);
my @champs = @$l;

# création de la table
my $largeur = scalar @champs;
my $table = odf_table->create(
          "MaFeuille",
          length      => 1,
          width       => $largeur
          );

# sélection de la première ligne de la table
my $ligne = $table->get_row(0);

# boucle de lecture/chargement
while (1) {
    for (my $i = 0 ; $i < $largeur ; $i++) {
          $ligne->get_cell($i)->set_value($champs[$i]);
          }
    $l = $source->getline($FH) or last;
    @champs = @$l;
    $ligne = $table->add_row;
    }
close $FH;

# création du document et rattachement de la table
my $doc = odf_document->create('spreadsheet');
my $contexte = $doc->get_body;
$contexte->clear;
$contexte->append_element($table);
$doc->save(target => $fichier_ods);
```

On remarquera au passage qu'une table peut être créée et chargée *avant* le document auquel elle sera éventuellement rattachée.

Utilisée comme ici sans aucun paramètre, `add_row()` ajoute à la fin de la table une ligne qui est un clone de la dernière ligne existante. Il existe bien sûr d'autres possibilités. En utilisant un paramètre `before` ou `after`, dont la valeur est un numéro de ligne, on peut insérer une nouvelle ligne juste *avant* ou *après* la position indiquée. Dans ce cas la ligne insérée est un clone de la ligne présente à la position indiquée. L'insertion en première position de la table est synonyme d'insertion avant la ligne en position zéro, donc :

```
$table->add_row(before => 0);
```

On peut aussi insérer un nombre quelconque de lignes en une seule instruction, en passant un paramètre number spécifiant le nombre de lignes, comme dans cet exemple qui insère dix lignes après la position 15 (donc après la 16ème ligne) :

```
$table->add_row(number => 10, after => 15);
```

Pour l'agrandissement « en largeur », on dispose de `add_column()` qui fonctionne selon la même logique et accepte les mêmes options que `add_row()`. Ainsi, l'exemple suivant agrandit une table dans les deux dimensions, en ajoutant cent lignes à la position 50 et cinq colonnes après la colonne *B* :

```
$table->add_column(number => 5, after => 'B');
$table->add_row(number => 100, before => 50);
```

Notez que, pour des raisons de performances, quand on agrandit significativement une table il vaut mieux exécuter `add_column()` en premier ; en effet, `add_column()` est plus fortement impacté par le nombre de lignes que `add_row()` par le nombre de colonnes.

Dans certains cas, le fait que `add_row()` et `add_column()` initialisent systématiquement les nouvelles cellules avec les valeurs des cellules de la ligne ou de la colonne située à la position de référence est gênant. Si on souhaite que ces cellules soient créées vides, il suffit de passer une option `empty` avec la valeur `TRUE` :

```
$table->add_column(after => 'C', empty => TRUE);
```

Nous savons donc comment étendre une table… mais comment la réduire ? Évidemment, en supprimant des lignes ou des colonnes, ce qui est possible de deux façons. La première manière, qui suppose une sélection préalable de la ligne ou de la colonne à supprimer, utilise le destructeur générique `delete()` :

```
$ligne = $table->get_row(15);
$ligne->delete;
```

La seconde s'opère directement à partir du contexte de la table et passe par la méthode spécifique `delete_row()` ou `delete_column()`. L'instruction ci-dessous produit le même résultat que l'exemple précédent :

```
$table->delete_row(15);
```

5.5 Styles de tableaux

La décoration des tableaux par programme est certainement l'une des tâches les plus fastidieuses d'une génération de documents, mais lpOD s'efforce d'éviter qu'elle reste un casse-tête insurmontable. Nous nous contenterons ici de donner quelques pistes simples. Le premier principe à retenir est que la majeure partie des caractéristiques de présentation d'une table se définit à travers des *styles de cellules*, mais que la structure dans laquelle les cellules sont présentées est contrôlée par un *style de table*, un ou plusieurs *styles de colonnes*, et un ou plusieurs *styles de lignes*.

5.5.1 Style de table (au sens strict)

Un style de table s'applique à un objet `odf_table` et concerne, avant tout, son enveloppe extérieure, c'est-à-dire sa taille physique (à ne pas confondre avec son nombre de lignes et de colonnes), ses marges, ses bordures, son arrière-plan, sa visibilité, etc. Pour en savoir plus, je vous renvoie au chapitre `ODF::lpOD::Style` du manuel, sachant que l'exemple suivant donne l'idée générale. La création du style s'appuie sur le constructeur habituel, en choisissant la famille `table`.

Dans l'instruction suivante, les paramètres donnés correspondent à une table qui sera centrée et occupera 80 % de la largeur de la page.

```
odf_style->create(
    'table',
    name => "TableauCentré", width => "80%", align => "center"
    )->register($doc);
```

Notez bien que, en général comme dans cet exemple, la création d'un style de table n'a pas de sens dans un document de tableur où, par définition, une table occupe toute la surface disponible. On suppose ici que ce style concerne un tableau dans un autre type de document, notamment un document de texte.

On peut appliquer ce style à une table dès sa création, via une option `style` :

```
$table = odf_table->create(
                "Mon Tableau",
                size  => [100, 75],
                style => "TableauCentré"
                );
```

On peut aussi l'appliquer à une table existante :

```
$table->set_style("TableauCentré");
```

5.5.2 Style de colonne

Parmi les fonctions d'un style de colonne, la plus importante est la largeur. Celle-ci peut se définir de manière relative ou absolue. Pour d'autres propriétés des styles de colonnes, voir `ODF::lpOD::Style`.

Un style de colonne est créé de la manière habituelle, avec un nom de famille qui est '`table column`', et se paramètre ensuite avec `set_properties()`. Parmi les propriétés, la largeur correspond à un paramètre `width` et une valeur relative se représente par un nombre suivi d'un astérisque. Pour illustrer le propos, nous allons prendre comme exemple un tableau à trois colonnes dont nous voulons attribuer 50 % de la largeur à la première colonne, 20 % à la seconde et 30 % à la troisième. Nous allons créer à cet effet trois styles de colonnes nommés, par commodité, « *C0* », « *C1* » et « *C2* ». Avant de montrer comment faire cette création, débarrassons-nous de la question la plus simple, à savoir comment, après avoir créé une table de trois colonnes, attribuer chacun de ces styles à une colonne via l'habituelle méthode `set_style()` :

```
$table = odf_table->create("Mon tableau", size => [100, 3]);
$table->get_column($_)->set_style("C$_") for (0..2);
```

Imaginons un nombre arbitraire, disons *1000*, comme représentant la largeur totale de la table (ce nombre n'est associé à aucune unité de mesure, il ne correspond pas à une largeur, mais à un nombre de « parts » de la largeur quelle qu'elle soit). Compte tenu de la répartition choisie, nous devons respectivement attribuer *500*, *200* et *300* « parts » à chacun des trois styles. Le caractère relatif du paramètre est spécifié par la présence d'un astérisque final au lieu d'une unité de longueur :

```
odf_style->create('table column', width => "500*", name => "C0")
    ->register($doc);
odf_style->create('table column', width => "200*", name => "C1")
    -register($doc);
odf_style->create('table column', width => "300*", name => "C2")
    ->register($doc);
```

Remarquez que, théoriquement, on devrait obtenir le même résultat avec des valeurs *50**, *20** et *30** ou même *5**, *2** et *3**. puisque c'est le *rapport* entre les valeurs qui compte, et non pas les valeurs elles-mêmes. En pratique, et indépendamment de la norme OpenDocument, les résultats visibles semblent plus fiables avec des nombres à trois chiffres.

Notez qu'on a évité ici l'appel explicite de `set_properties()` en passant directement l'option `width` au constructeur, qui sait quoi en faire ; en l'occurrence il n'est pas nécessaire de préciser une `area`.

Les styles de colonnes, comme les autres, sont intégrés au document par `register_style()`. L'exemple récapitulatif suivant montre comment définir une table et les styles de colonnes correspondants à partir d'une table de paramètres.

```
my %l = (
    C0    => "500*",
    C1    => "200*",
    C2    => "300*"
    );
my $largeur = scalar keys %l;
foreach my $n (keys %l) {
    odf_style->create(
          'table column',
          name  => $n,
          width => $l{$n}
          )->register($doc);
    }
my $table = $doc->append_element(
    odf_table->create(
          "MaTable",
          length      => 100,
          width       => $largeur
          )
    );
$table->get_column($_)->set_style("C$_") for 0..$largeur-1;
```

Dans un document de type texte (`text`), où chaque table a en principe une largeur bien définie, il est recommandé d'attribuer des tailles relatives à toutes les colonnes, comme dans cet exemple. Dans un document de type classeur (`spreadsheet`), où le nombre de colonnes est présenté comme « infini », il est plus simple d'attribuer des largeurs absolues à certaines colonnes choisies. La syntaxe est la même, hormis la présence d'une unité de longueur (`mm`, `cm`, etc.) et non d'un astérisque. L'exemple suivant attribue une largeur absolue de *six centimètres* à la colonne *B* d'une feuille de calcul.

```
odf_style->create(
    'table column',
    name  => "StylePourB",
    width => "6cm"
    )
    ->register($doc);
$table->get_column("B")->set_style("StylePourB");
```

5.5.3 Style de ligne

On s'en tiendra, dans cette présentation, au rôle le plus important d'un style de ligne : le contrôle de la *hauteur*, bien que d'autres propriétés soient accessibles.

La hauteur se spécifie via un paramètre `height` et, ce détail mis à part, la création et l'utilisation d'un style de ligne (dont le nom de famille est '`table row`') fonctionnent de manière similaire à ce que nous avons vu à propos des styles de colonnes, comme on le voit dans la séquence ci-après qui affecte une hauteur de *six centimètres* à la ligne *5* d'une table.

```
odf_style->create(
    'table row',
    name        => "GrandeHauteur",
    height      => "8cm"
    )->register($doc);
$table->get_row(5)->set_style("GrandeHauteur");
```

En dehors du style de ligne proprement dit, lpOD permet aussi d'associer à une ligne un *style de cellule par défaut*. Il s'agit d'un style de cellule comme les autres, dont l'utilisateur souhaite qu'il soit appliqué à toutes les cellules de la ligne qui n'ont pas de style spécifié. Ce style de cellule est associé par `set_default_cell_style()`. L'instruction suivante attribue le style « *Bleu Ciel* » (que nous définirons plus loin) à toutes les cellules sans style particulier de la ligne *5* :

```
$table->get_row(5)->set_default_cell_style("Bleu Ciel");
```

Attention, l'attribution d'un style par défaut ne signifie pas que toutes les cellules visibles dans la grille du tableur seront affectées jusqu'à l'infini ; le style par défaut ne s'applique qu'aux cellules du tableau *réel*, seules connues de lpOD, et non aux cellules présentées dans la grille mais non définies.

5.5.4 Style de cellule

Les paramètres de présentation d'une cellule sont essentiellement ceux qu'on peut imaginer à propos de toute espèce de cadre rectangulaire de présentation de données. Il s'agit de couleur de fond, de bordures, de formats numériques, de polices de caractères, etc. La page de manuel `ODF::lpOD::Style` indique la syntaxe générale et les attributs les plus indispensables, mais pour en avoir la liste exhaustive la référence ultime est la spécification *OpenDocument* elle-même. Ici, nous allons introduire l'idée générale en créant le style « *Bleu Ciel* » déjà cité.

Il s'agit d'un style dont le nom de famille est (vous l'auriez deviné) `'table cell'` ; nous allons lui donner un arrière-plan bleu ciel et un encadrement bleu marine.

```
my $bleu_marine = color_code('navy blue');
odf_style->create(
    'table cell',
    name             => "Bleu Ciel",
    background_color  => "sky blue",
    border           => "1mm solid $bleu_marine"
    )->register($doc);
```

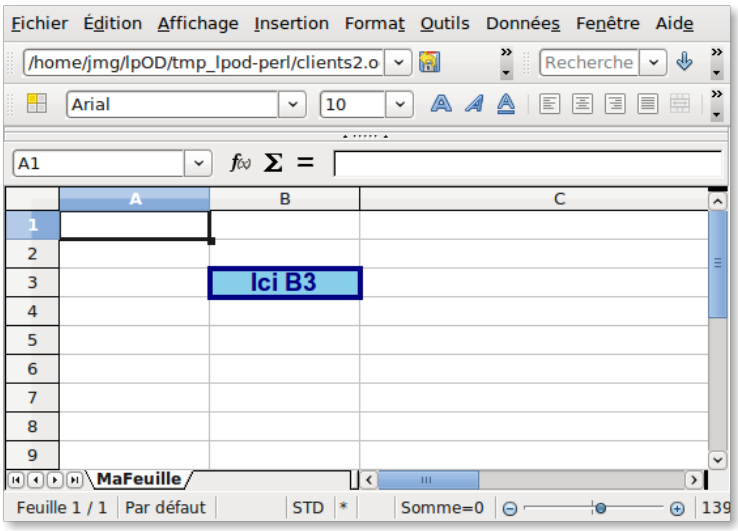
Si nous voulons, en plus, que le contenu des cellules concernées par ce style soit *centré* (ce qui est une propriété de *paragraphe*) et présenté en *caractères gras de couleur bleu marine* (ce qui est une propriété de *texte*), nous devons compléter la définition du style par deux appels de `set_properties()` sur des zones différentes, et donc réécrire le précédent exemple, entre autres possibilités, de la façon suivante :

```
my $sc = odf_style->create(
    'table cell',
    name             => 'Bleu Ciel',
    background_color  => 'sky blue',
    border           => '1mm solid #000080'
    );
$sc->set_properties(
    area  => 'paragraph',
    align => 'center'
    );
$sc->set_properties(
    area       => 'text',
    weight     => 'bold',
    color      => 'navy blue'
    );
$doc->register_style($sc);
```

On peut ensuite appliquer ce style à une cellule particulière :

```
$c = $doc->get_body->get_table("MaFeuille")->get_cell('B3');
$c->set_text("Ici B3");
$c->set_style('Bleu Ciel');
```

Ce qui donne à la cellule *B3* l'aspect présenté dans l'image ci-dessous :



6 Cadres, images et pages de présentation

Les cadres (au sens large) sont des zones rectangulaires pouvant héberger des contenus variés, images ou textes. Les « *pages de dessin* » (*draw pages*) sont des cadres spéciaux qui représentent chacun une page et qui sont utilisables dans les documents de type *présentation* ou *dessin*.

Ce chapitre est un bref aperçu sur la manipulation des cadres d'image, des cadres textuels, et des pages de présentation.

6.1 Cadre d'image

À l'exception des pages de dessin qui représentent un cas particulier, les cadres sont, dans le vocabulaire ODF, des « *draw frames* » (littéralement « *cadres de dessin* ») et, dans lpOD, des instances de la classe `ODF::lpOD::Frame`, également nommée `odf_frame`. Il existe bien entendu un constructeur `create()` pour cette classe. Un cadre d'image peut se construire (au minimum) avec un paramètre `image` spécifiant le chemin d'accès à un fichier graphique :

```
my $cadre = odf_frame->create(image => '/dossiers/images/logo.png');
```

lpOD fournit un alias pour ce constructeur :

```
my $cadre = odf_create_image_frame('/dossiers/images/logo.png');
```

Mais le résultat obtenu avec aussi peu de paramètres sera rarement très utile. En effet, la seule indication d'une ressource graphique externe à intégrer dans le document ne dit pas *comment* et *où* sera présentée l'image.

Il est d'abord préférable de spécifier, via un paramètre `size`, la taille d'affichage de l'image dans le document. Si cette taille n'est pas indiquée, lpOD va essayer de déterminer la taille originale qui, si tout va bien[16], sera appliquée dans le document. La valeur de `size` doit donner la largeur et la hauteur sous la forme d'une chaîne de caractères (avec virgule séparatrice) ou d'une référence de liste contenant les deux valeurs. Les deux formes ci-dessous sont acceptées pour définir une taille de 5×12 cm :

```
          size  => "5cm, 12cm"
          size  => [ "5cm", "12cm" ]
```

Il est recommandé de fournir un paramètre `name` attribuant au cadre un nom unique (qui pourra servir à le retrouver plus tard).

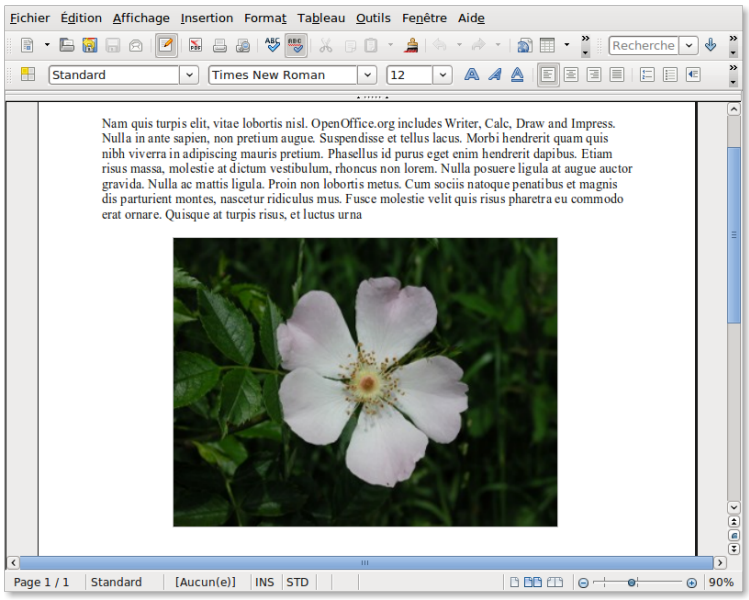
Commençons par un cas d'utilisation simple : l'ancrage d'une fenêtre à la suite d'un texte par l'intermédiaire d'un paragraphe, comme dans l'exemple suivant :

```
$base = $doc->get_body;
$para = $base->append_element(odf_paragraph->create);
$para->append_element(
    odf_frame->create(
          image => '/dossiers/images/eglantine.png',
          name  => "Églantine"
          )
    );
```

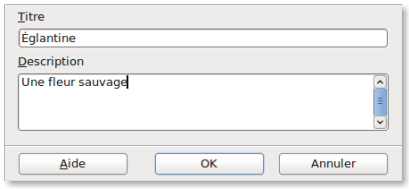
Dans cette séquence, nous commençons par ajouter un paragraphe vide à la fin du texte, puis nous attachons à ce paragraphe un cadre d'image nommé « *Églantine* ». L'objet `odf_frame`, dans ce cas, réalise l'ancrage d'une image externe au paragraphe. L'image est « flottante », c'est-à-dire qu'elle va « descendre » ou « monter » si on insère ou supprime des paragraphes situés avant son paragraphe d'ancrage. De plus, l'image disparaîtra automatiquement si ce paragraphe d'ancrage est supprimé.

Perfectionnons un peu l'exemple : nous allons présenter ce cadre en position *centrée* de manière à obtenir un résultat semblable à ce que montre la copie d'écran suivante.

16 C'est-à-dire si la ressource est accessible en lecture et si l'image est dans l'un des formats supportés par `Image::Size`, qui est utilisé par lpOD.



Pour centrer l'image, sachant qu'elle est attachée à un paragraphe, nous pouvons tout simplement centrer le paragraphe lui-même, au moyen, bien entendu, d'un style de paragraphe, que nous gratifierons aussi d'une marge supérieure (`margin_top`) de *5mm* pour faire une séparation nette entre l'image et le paragraphe précédent.



Pour enrichir l'exemple, nous allons aussi « documenter » l'image au moyen d'un titre et d'une description (via les paramètres `title` et `description`), dont nous pourrons vérifier l'existence, par exemple, avec le traitement de texte dans la fenêtre « *Description* » (reproduite ci-dessus) associée à l'image[17] :

```
odf_style->create(
        'paragraph',
        name            => "Centré",
        align           => 'center',
        margin_top      => '5mm'
        )->register($doc);
my $base = $doc->get_body;
my $para = $base->append_element(
        odf_paragraph->create(style => "Centré")
        );
$para->append_element(
    odf_frame->create(
          image       => '/dossier/images/eglantine.jpg',
          name        => "Fleur",
          title       => "Églantine",
          description  => "Une fleur sauvage"
          )
    );
```

Pour être vraiment complet, il faudrait associer au cadre (comme à tout objet visible) un style. Mais pour l'instant nous allons faire l'impasse là-dessus en supposant que le style par défaut applicable aux cadres nous convient dans ce contexte précis. Attention, ce ne sera pas toujours le cas !

Au lieu d'attacher l'image à un paragraphe afin que sa position soit relative au texte, nous pouvons décider de lui donner une position fixe par rapport à une *page* de texte. Dans ce cas le cadre ne peut plus être attaché à un paragraphe. Il est généralement rattaché directement au corps du document[18], mais il doit comporter des paramètres de position supplémentaires, à savoir, au minimum, un paramètre `page` spécifiant le numéro de la page d'ancrage, et un paramètre `position` indiquant les coordonnées. Reprenons le même exemple mais avec un placement de l'image aux coordonnées *6×12 cm* dans la page *1*.

17 Cette fenêtre est accessible par un clic droit dans l'image, en choisissant l'entrée « *Description* ».
18 D'autres contextes de rattachement sont possibles mais ne sont pas abordés dans cet article.

```
$doc->get_body->insert_element(
    odf_frame->create(
          name         => "Fleur",
          image        => '/dossier/images/eglantine.jpg',
          title        => "Églantine",
          description  => "Une fleur sauvage",
          page         => 1,
          position     => "6cm, 12cm"
          )
    );
```

Mais là, dans la plupart des cas, une mauvaise surprise nous attend : l'image apparaît généralement à une position arbitraire qui n'est pas celle qu'on a spécifiée. Ce n'est pas un bug : la norme ODF prévoit en effet que les coordonnées d'un cadre peuvent être *interprétées* selon le style de ce cadre. Ici, nous voulons que les coordonnées soient appréciées à partir du coin supérieur gauche de la page et, pour le dire, il nous faut un style. En l'occurrence, les styles appropriés pour les cadres sont de la famille `graphic`. Reprenons l'exemple en conséquence :

```
$doc->get_body->insert_element(
    odf_frame->create(
          name         => "Fleur",
          image        => '/dossier/images/eglantine.jpg',
          title        => "Églantine",
          description  => "Une fleur sauvage",
          page         => 1,
          position     => "6cm, 12cm",
          style        => "CadreBasique"
          )
    );
odf_style->create('graphic', name => "CadreBasique")->register($doc);
```

Curieusement, cet exemple fonctionne alors que le style « *CadreBasique* » semble avoir été créé sans aucun paramètre autre que son nom de famille et son nom propre, donc sans aucune indication quant à l'interprétation des coordonnées. Rassurez-vous, il n'y a pas de mystère : lors de la création d'un style graphique, des valeurs par défaut sont automatiquement fournies par lpOD pour certains paramètres importants, dont le mode d'interprétation des coordonnées, et il se trouve que ces paramètres correspondent à notre cahier des charges.

Il reste cependant un dernier point à traiter avant d'en finir avec les images. Dans notre exemple, l'image est une ressource externe désignée par un chemin d'accès dans un système de fichiers local. Mais si le document ainsi traité est ensuite exploité sur un site distant le fichier graphique ne sera plus accessible et le cadre sera vide[19]. Une solution consiste à utiliser un lien internet et à rendre l'image accessible en ligne, mais l'utilisateur ne peut pas ou ne veut pas forcément être toujours connecté. En conséquence, la solution la plus « portable » consiste à embarquer physiquement l'image dans le document au moyen d'une méthode `add_image_file()` de la classe `odf_document`. Cette méthode reçoit en argument le chemin d'accès d'un fichier externe et retourne deux valeurs : un lien spécial qui correspond à l'adresse de l'image importée dans le conteneur du document, et la taille de l'image[20]. Dans ce cas, le chemin d'accès qui, dans la construction du cadre, désigne l'image, doit être remplacé par le lien spécial retourné par `add_image_file()`. Voici notre exemple réécrit dans cette perspective.

19 ou agrémenté d'un message « *Erreur de lecture* ».
20 Pour cela lpOD utilise `Image::Size` ; voir la documentation de ce module pour connaître la liste des formats graphiques supportés.

```
My ($lien_image, $taille_image) =
    $doc->add_image_file('/dossier/images/eglantine.jpg');
$doc->get_body->insert_element(
    odf_frame->create(
          name         => "Fleur",
          image        => $lien_image,
          title        => "Églantine",
          description  => "Une fleur sauvage",
          page         => 1,
          position     => "6cm, 12cm",
          size         => $taille_image,
          style        => "CadreBasique"
          )
    );
odf_style->create('graphic', name => "CadreBasique")->register($doc);
```

On remarquera que cette fois le paramètre `size` a été explicitement renseigné et qu'on n'a pas laissé lpOD le deviner. On a profité ici d'une fonctionnalité de `add_image_file()` qui, utilisée en contexte de liste, retourne d'abord une chaîne de caractères qui sera l'adresse interne de l'image quand elle aura été chargée, et ensuite la *taille* de l'image.

On pourrait légèrement raccourcir ce code en créant d'abord un cadre vide sans taille et en le garnissant ensuite avec la méthode `set_image()` qui permet, en une instruction, de demander le chargement du fichier et de fixer la taille originale de l'image :

```
$cadre = odf_frame->create(
    name         => "Fleur",
    title        => "Églantine",
    description  => "Une fleur sauvage",
    page         => 1,
    position     => "6cm, 12cm",
    style        => "CadreBasique"
    );
$doc->get_body->insert_element($cadre);
$cadre->set_image('/dossier/images/eglantine.jpg', load => TRUE);
odf_style->create('graphic', name => "CadreBasique")->register($doc);
```

Dans cet exemple, on appelle `set_image()` avec une option `load` qui force l'embarquement de l'image dans le document ; si cette option était absente ou `FALSE`, l'image ne serait pas chargée et serait gérée comme une ressource externe.

6.2 Cadres de texte

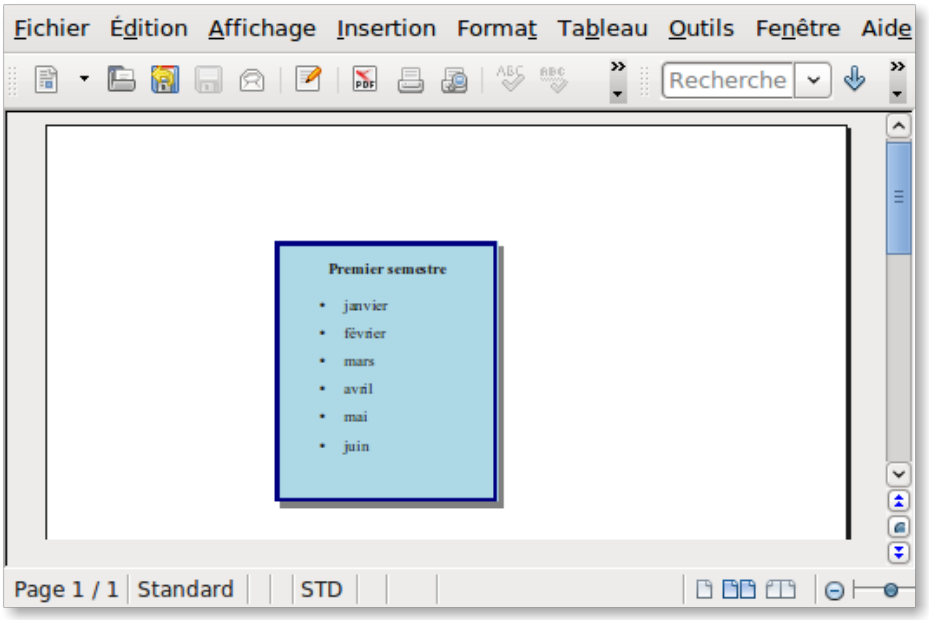
Les règles de construction sont les mêmes pour les cadres de textes que pour les cadres d'image, la seule différence étant liée, bien évidemment, au contenu. Il s'agit toujours d'objets `odf_frame`. Ce qui a été dit dans la rubrique précédente concernant l'ancrage, la position, la taille et le style est applicable. Quant à la désignation du contenu, elle peut se faire de deux manières (et à deux moments différents). Voyons d'abord la plus simple, qui implique un simple paramètre `text` dans l'appel du constructeur. Nous allons tout simplement reprendre l'exemple du début de cet article sous une nouvelle forme : l'affichage du message « *Bonjour le monde !* » dans une boîte de texte de *8×2 cm* ancrée à la première page d'un document textuel vide et placée en position *6×12 cm*.

On notera que, pour les mêmes raisons que dans l'exemple précédent, nous avons besoin d'un style graphique pour la bonne interprétation des coordonnées. Bien entendu, ce style, que nous avons nommé « *CadreBasique* », n'aurait à être défini qu'une fois pour toutes dans un document et pourrait être réutilisé par plusieurs cadres de texte et/ou d'image.

```
use ODF::lpOD;

my $doc = odf_document->create('text');
my $contexte = $doc->get_body;
$contexte->insert_element(
    odf_frame->create(
          text        => "Bonjour le monde !",
          name        => "Salut",
          size        => "8cm, 2cm",
          position    => "6cm, 12cm",
          page        => 1,
          style       => "CadreBasique"
          )
    );
odf_style->create(
    'graphic',
    name         => "CadreBasique"
    )->register($doc);
$doc->save(target => "bonjour.odt");
exit;
```

En pratique on aura souvent à créer des boîtes de texte présentant des contenus plus sophistiqués qu'une simple ligne de texte. Dans ce cas, au lieu du paramètre de construction `text` (qui n'est pas plus obligatoire que le paramètre `image`), on commencera par créer un cadre vide, et on utilisera ensuite la méthode `set_text_box()` pour le garnir. Cette dernière méthode accepte en entrée une liste dont les éléments sont, au choix, des chaînes de caractères ou des objets `odf_element`. Chaque chaîne de caractères donnera lieu à création automatique d'un paragraphe dont elle sera le texte et qui sera ajouté au contenu. Chaque `odf_element` sera inséré tel quel, ce qui veut dire qu'on peut intégrer dans une boîte de texte toutes sortes de contenus structurés, et notamment des paragraphes possédant déjà chacun son propre style, ou des listes (à points ou à numéros). L'application la plus fréquente des cadres de texte concerne les supports de présentation (abordés plus loin), où chaque cadre est généralement utilisé pour présenter une liste à points[21], mais les cadres peuvent contenir (et mélanger) plusieurs sortes de structures et sont parfaitement utilisables dans des pages de documents textuels.



Notre prochain exemple va construire un cadre de texte semblable à celui de l'image ci-dessus. Il contiendra un paragraphe centré en caractères gras, avec une marge inférieure de 4 millimètres, suivi d'une liste à points. Afin d'être bien visible, le cadre aura un fond bleu clair et une bordure d'un millimètre en bleu foncé, avec une ombre grise décalée en bas à droite de 1,5 millimètre. Et cette bordure sera écartée du contenu d'un espace d'au moins 4 millimètres.

21 car nous vivons dans un monde où la communication professionnelle est largement tributaire du « *point de pouvoir* »…

Toute cette décoration implique, on s'en doute, un style graphique un peu plus riche que le précédent. Mais comme cet article d'initiation n'a pas vocation à devenir un roman-fleuve, la construction de styles graphiques élaborés, permettant de régler les bordures, la transparence, les couleurs, la luminosité, l'adaptation au texte, et bien d'autres détails, fait partie des nombreux thèmes qui ne sont pas traités ici ; ce style est donc présenté sans commentaire dans l'exemple (mais peut cependant suggérer quelques pistes). Et comme il y aura un paragraphe centré, il faut aussi définir un style de paragraphe.

Commençons par la définition des styles :

```
# style de paragraphe
$doc->register_style(
        odf_style->create(
                'paragraph',
                name            => "Centré",
                align           => 'center',
                margin_bottom   => '4mm'
                )
        )->set_properties(
                area    => 'text',
                weight  => 'bold'
        );

# style de cadre
$doc->register_style(
        odf_style->create(
                'graphic',
                name             => "Cadre élaboré",
                border           => '1mm solid #000080',
                padding          => '4mm',
                background_color => 'light blue',
                shadow           => '#808080 1.5mm 1.5mm'
                )
        );
```

Passons maintenant au contenu.

Nous découvrirons au passage un exemple de création et d'ajout d'articles dans une liste à points (encore un sujet sur lequel on trouvera des informations dans `ODF::lpOD::StructuredContainer` mais qui n'est pas développé ici).

Cette fois le cadre sera créé sans option `text`, donc vide, le contenu étant fourni ultérieurement par `set_text_box()`.

```
# création du cadre vide
my $frame = odf_frame->create(
        name            => "F123",
        size            => "6cm, 4cm",
        position        => "6cm, 3cm",
        page            => 1,
        style           => "Cadre élaboré"
        );

# insertion du cadre dans le document
$context->insert_element($frame);

# création du paragraphe
my $paragraphe = odf_paragraph->create(
        text    => "Premier semestre",
        style   => "Centré"
        );

# création et peuplement de la liste à points
my $liste = odf_list->create;
$liste->add_item(text => $_)
          for qw(janvier février mars avril mai juin);

# insertion du paragraphe et de la liste dans la boîte
$frame->set_text_box($paragraphe, $liste);
```

6.3 Pages de présentation

Un support de présentation est un document dont la particularité est d'avoir un contenu entièrement constitué d'une suite de « *pages de dessin* », c'est-à-dire, du point de vue de lpOD, d'objets `ODF::lpOD::DrawPage`, alias `odf_draw_page`. Une page de dessin, en soi, est un objet relativement simple, mais compte tenu des rasisons d'être habituelles d'un document de présentation, où la forme compte autant sinon plus que le contenu, la construction et l'organisation des styles applicables sont très complexes. Il est donc recommandé d'éviter systématiquement d'avoir à créer de tels documents par programme, et d'utiliser Perl pour exploiter des documents existants ou créer des présentations à partir de modèles convenablement stylés. On s'en tiendra donc ici à quelques possibilités de manipulation, sans aborder le traitement des styles de page de présentation.

La séquence suivante permet de retrouver une page d'après son nom (tel qu'il apparaît dans un logiciel interactif) :

```
$base = $doc->get_body;
$page = $base->get_draw_page_by_name("Ma Page");
```

La méthode `get_draw_page()` donne plus de chances de succès : elle essaie d'abord de trouver une page dont l'*identifiant interne* est conforme à l'argument, et en cas d'échec elle recherche une page dont le *nom* correspond à l'argument :

```
$page = $base->get_draw_page("XYZ");
```

Une page (avec tout son contenu) peut être supprimée du document sans formalité particulière :

```
$page->delete;
```

Elle peut également être copiée, et la copie insérée ailleurs, comme dans l'exemple suivant qui copie une page nommée « *Introduction* », renomme la copie « *Conclusion* » et la place à la fin du document, le tout en utilisant des méthodes génériques.

```
$page = $base->get_draw_page("Introduction")->clone;
$page->set_name("Conclusion");
$page->set_id("ID001");
$base->append_element($page);
```

Il faut noter que la copie d'une page de présentation peut se faire au sein d'un même document ou entre documents. Dans le second cas il faut cependant veiller à ce que les styles utilisés par la page copiée soient présents à l'identique dans les deux documents.

La création d'une page de dessin est assez simple (seuls les styles sont complexes). La forme minimale recommandée est représentée par l'exemple ci-dessous. Le premier argument est l'identifiant interne (non visible pour l'utilisateur final), et le nom est passé via un paramètre `name` :

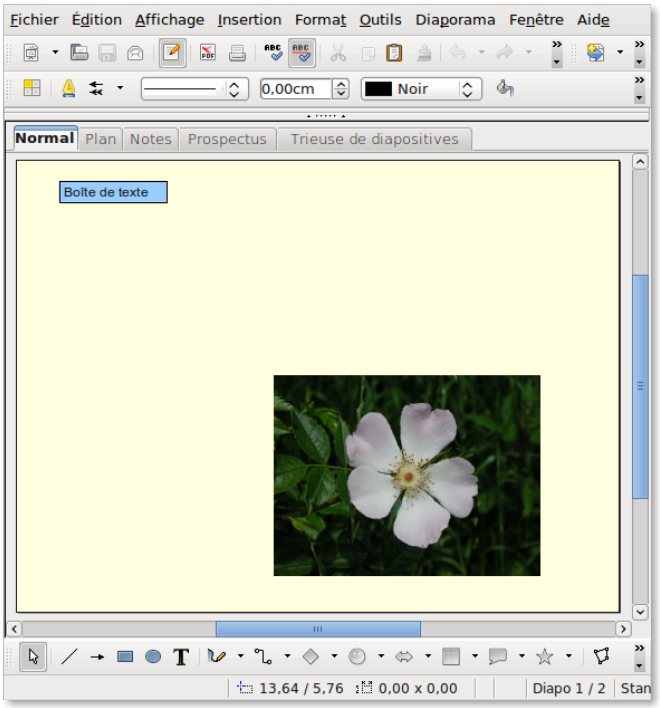
```
$page = odf_draw_page->create('id1234', name => "Ma Page");
```

Il est conseillé d'affecter à la page un *style* de la famille '`drawing page`' (du moins si on en connaît un qui convient), via un paramètre `style`.

La définition du contenu ne pose pas de problème spécifique puisque les pages de présentation sont essentiellement garnies de cadres de texte et d'image tels que ceux que nous avons déjà abordés dans ce chapitre. Pour placer un cadre dans une page, il suffit d'appeler `insert_element()` ou `append_element()` à partir de cette page. Sans oublier d'indiquer la position.

L'exemple suivant, qui crée un support de présentation très simple d'une page, avec une boîte de texte et une boîte d'image, montre comment combiner un objet `odf_drawing_page` avec des objets `odf_frame` et indique aussi comment créer un style de page de dessin très sommaire (ici, sa seule propriété sera la couleur de remplissage jaune pâle).

```
my $doc = odf_document->create('presentation');
my $base = $doc->get_body;
$base->clear;
my $page = odf_draw_page->create(
    'id1234', name => "Ma Page", style => "Dessin"
    );
$base->insert_element($page);
$p->append_element(
    odf_frame->create(
          text        => "Boîte de texte",
          size        => "5cm, 1.5cm",
          position    => "2cm, 1cm"
          )
    );
my $fr = odf_frame->create(
    name => "Églantine", position => "12cm, 10cm"
    );
$p->append_element($fr);
$fr->set_image('/dossiers/images/eglantine.jpg', load => TRUE);
$doc->register_style(
    odf_style->create(
          'drawing page',
          name        => "Dessin",
          fill_color  => 'light yellow'
          ),
    automatic => TRUE
    );
$doc->save(target => "presentation.odp");
```



On s'étonnera peut-être de la présence d'un paramètre booléen `automatic`, dont la valeur est `TRUE`, utilisé avec `register_style()`. Ce paramètre optionnel permet de demander à lpOD d'enregistrer ce style de page en tant que « *style automatique* ». Pour en savoir plus sur ce que cela signifie, reportez-vous à la page de manuel `ODF::lpOD::Style`. Ici, on se limitera à signaler que cette directive (dans le cas précis de notre exemple) est nécessaire pour que ce style produise effet en l'état. Si tout va bien, le résultat de ce programme, dans la configuration par défaut de lpOD, devrait ressembler à l'image de gauche.

Le fond bleu de la boîte de texte en haut à gauche provient du style de cadre par défaut de lpOD dans les présentations (alors que dans les textes ce fond par défaut est blanc). Quant au cadre d'image, nous avons simplement repris le même que celui de la section précédente. Il est important de savoir qu'un même code peut servir à produire un cadre destiné à être attaché aussi bien à une page de texte, à un paragraphe, à une cellule de tableur, etc.

6.4 Exercice récapitulatif

Pour conclure ce chapitre, nous allons écrire un programme sans utilité ni prétention artistique, mais (espérons-le) de nature à démontrer d'autres manières de manipuler les pages de présentation, les cadres d'image (et au passage, implicitement, les styles graphiques). Notre objectif est de construire une page dont l'aspect est représenté ci-contre. On y remarque, sur un fond de couleur en dégradé axial incliné à 45°, un empilement de cinq exemplaires de la même image, avec un décalage de position permettant de voir toute la pile et de constater que la coloration s'estompe à mesure qu'on va vers le dessous de la pile. L'effet d'estompage est obtenu par une augmentation de luminosité et un ajout de couleur bleue.

Commençons par préparer le terrain. Après avoir initialisé le document, la séquence ci-dessous crée un style de page de dessin puis, comme ce style utilise un dégradé de couleur pour le remplissage, un *style de dégradé* (ou « *gradient* », présenté ici sans commentaire). Ensuite elle charge un fichier d'image et récupère son lien d'accès interne et la taille de l'image en utilisant `add_image_file()`. Enfin elle crée et insère dans le document une page de dessin (*draw page*) utilisant le style de page créé au début.

```
my $doc = odf_document->create('presentation');
$doc->register_style(
    odf_style->create(
          'drawing page',
          name               => "Perso",
          fill               => 'gradient',
          fill_gradient_name => "AxialJaune"
          ),
        automatic => TRUE
    );
$doc->register_style(
    odf_style->create(
          'gradient',
          name              => "AxialJaune",
          style             => 'axial',
          angle             => 450,
          start_color       => '#ffffcc',
          end_color         => '#ffcc99',
          start_intensity   => '100%',
          end_intensity     => '95%'
          )
    );
my ($image, $taille) = $doc->add_image_file(
    '/dossier/images/eglantine.jpg'
    );
my $page = odf_draw_page->create(
    'p001',
    name        => "MaPage",
    style       => "Perso"
    );
$doc->get_body->insert_element($page);
```

Ces bases étant posées, il reste à définir les cinq représentations de l'image, ce qui implique cinq cadres. Mais comme chaque représentation possède des caractéristiques de coloration distinctes, et sachant que de telles caractéristiques sont contrôlées par l'intermédiaire des styles, il faut donc définir autant de styles que de cadres. Comme personne n'aime la redondance de code, nous allons développer les cadres et les styles correspondants dans une boucle à cinq itérations de la manière suivante (qui bien sûr n'est pas la seule et peut-être pas la meilleure).

```
for my $i (2..6) {
    my $x = ($i * 2); my $y = ($i * 1.5);
    my $lum = ($i * 10) . '%';
    my $bleu  = ($i * 5) . '%';
    $doc->register_style(
          odf_style->create(
                'graphic',
                name            => "Style$i",
                luminance       => $lum,
                blue            => $bleu
                ),
          automatic    => TRUE
          );
    $p->insert_element(
          odf_frame->create(
                image           => $image,
                name            => "Cadre$i",
                position        => [$x, $y],
                size            => $taille,
                style           => "Style$i"
                )
          );
}
```

Dans la boucle (qui fait varier un compteur `$i` de *2* à *6*), la première ligne calcule à partir du compteur les coordonnées `$x` et `$y` qui seront appliquées aux cadres, de manière à obtenir un décalage de position, le premier cadre étant présenté en position *[4, 3]*. Les variables `$lum` et `$bleu`, respectivement destinées à doser la correction de luminosité et la correction de couleur bleue, sont calculées également à partir du compteur, de manière à obtenir un décalage de coloration d'un cadre à l'autre (un signe *%* est ajouté à la fin, car il s'agit de corrections en pourcentage). Le même compteur est utilisé pour définir le nom unique de chaque style graphique et de chaque cadre. À chaque itération, un style de cadre `Style$i` est créé avec des paramètres `luminance` et `blue` correspondant à la correction de couleur calculée. De même, un cadre `Cadre$i` est créé avec la position calculée `[$x,$y]` (on notera que les unités de longueur, telles que `cm` ou `mm`, ne sont pas spécifiées, mais lpOD sélectionne automatiquement le centimètre par défaut, ce qui convient ici).

Il ne reste plus qu'à terminer le traitement en enregistrant le document ainsi créé :

```
$doc->save(target => "cadres.odp");
```

7 Mise en page des textes

Les documents textuels (comme le présent article) peuvent nécessiter eux aussi des styles de page, mais il s'agit de styles très différents de ceux des pages de présentation. Une page de texte n'est pas un cadre comme une page de dessin : le texte y est « flottant », il peut passer d'une page à une autre selon les insertions et les suppressions de paragraphes.

7.1 Organisation générale

Un style de page de texte est dénommé « *master page* ». La traduction littérale en français par « *page maîtresse* » est trompeuse, car il s'agit d'un style de mise en page et non pas d'une page ; il vaut mieux dire « *modèle de page* ». Dans lpOD, un modèle de page est un objet `ODF::lpOD::MasterPage`, alias `odf_master_page`, et c'est un style de la famille '`master page`'. Son rôle consiste :

- à définir un *entête de page* (si nécessaire) ;
- à définir un *pied de page* (si nécessaire) ;
- à spécifier un *format de présentation*, défini ailleurs.

Un format de présentation de page est un objet `ODF::lpOD::PageLayout`, alias `odf_page_layout`. Il définit les caractéristiques graphiques de la page, telles que les dimensions, l'orientation (portrait/paysage), les marges, les bordures, l'arrière-plan. Un tel style s'inscrit dans la famille '`page layout`'.

Chaque style `master page` ou `page layout` se définit par le constructeur habituel `create()` de `odf_style`. Le lien entre les deux se fait par une option `page_layout` de `odf_master_page`. Un même `page layout` peut être utilisé par plusieurs `master pages`. Quant aux entêtes et pieds de page, ils sont traités comme des prolongements des `master pages`, et non comme des styles.

7.2 Création et utilisation d'un modèle de page

La création d'un modèle de page (`master page`) est extrêmement simple… tant qu'il n'y a ni entête ni pied de page :

```
my $mp = odf_style->create(
    'master page',
    name        => "ModèlePage",
    page_layout => "PrésentationPage"
    )->register($doc);
```

Ce modèle est minimaliste : sa seule propriété utile est le nom (identifiant) du style de présentation (`page layout`) applicable.

La question primordiale est : comment l'utiliser ? Avec les pages de présentation, c'est facile : toute page est un cadre possédant un attribut `style`, mais dans un document de texte les pages n'ont pas d'existence statique (le nombre de pages du document n'est même pas forcément connu du programmeur). En fait, l'approche est tout à fait différente : l'activation d'un style de page se fait par l'intermédiaire d'un *style de paragraphe*. Il existe en effet, dans la définition d'un style de paragraphe, un paramètre optionnel `master_page`. Attention, il ne s'agit pas d'employer systématiquement ce paramètre, nous allons expliquer pourquoi, à partir de l'exemple suivant :

```
odf_style->create(
    'paragraph',
    name        => "Saut",
    master_page => "ModèlePage",
    parent      => "CorpsDeTexte"
    )->register($doc);

$doc->get_body->append_element(
    odf_paragraph->create(text => "Bonjour", style => "Saut")
    );
```

Cette séquence ajoute un paragraphe dont le style, nommé « *Saut* », fait référence au modèle de page précédemment défini. Le seul fait d'appliquer ce style a deux conséquences :

1) le paragraphe concerné sera le *premier* d'une nouvelle page (autrement dit il sera toujours précédé d'un saut de page) ;

2) la page au début de laquelle il apparaîtra, et toutes les pages suivantes jusqu'au prochain paragraphe dont le style portera une option `master_page`, auront une structure et une présentation déterminées par le modèle de page « *ModèlePage* » et son style de présentation associé « *PrésentationPage* ».

On n'a donc pas à se demander quels sont les numéros des pages auxquelles le style s'appliquera ; la question, comme on le voit, ne se pose pas dans ces termes-là. Il suffit de se souvenir qu'un style de page entre en action à chaque fois qu'il est « appelé » par le style d'un paragraphe et reste en vigueur jusqu'à ce qu'un autre style de page soit « appelé » à l'occasion d'un nouveau saut de page. Corollaire : deux paragraphes dont le style possède une option `master_page` ne peuvent pas apparaître dans la même page.

7.3 Entête et pied de page

Les deux annexes optionnelles d'un modèle de page, à savoir l'entête et le pied, se définissent simplement à l'aide des méthodes `set_header()` et `set_footer()` à partir d'un `master page` déjà créé, comme celui de la section précédente. Chacune de ces méthodes retourne un objet qui pourra servir de contexte pour insérer du contenu dans l'entête ou le pied :

```
my $mp = odf_style->create(
    'master page',
    name        => "ModèlePage",
    page_layout => "PrésentationPage"
    );
$doc->register_style($mp);
my $entete = $mp->set_header;
my $pied = $mp->set_footer;
```

Après cette séquence, le modèle « *ModèlePage* » possède un entête et un pied vides, mais représentés par des objets `$entete` et `$pied` qui peuvent être utilisés, via `insert_element()` ou `append_element()`, pour attacher presque n'importe quels objets, notamment des paragraphes, des tables, des cadres, etc.

Occupons-nous d'abord de l'entête. Nous allons prendre pour modèle un entête de page conforme à l'image ci-contre[22]. Cet entête comporte un cadre d'image du côté gauche et du texte à droite. Pour obtenir cette disposition, le moyen le plus simple implique de commencer par placer un tableau à une ligne et deux colonnes, à placer le cadre d'image dans la cellule de gauche et le texte dans la cellule de droite.

22 L'image provient de la communauté *OASIS OpenDocument XML* (*http://opendocument.xml.org*). Le modèle de page utilisé dans cet exemple est celui du document automatiquement généré par l'utilitaire de test d'installation de `ODF::lpOD`.

Tout cela est réalisé par le code suivant. On remarquera au passage l'insertion, dans la cellule de gauche (« *A1* ») du tableau, d'un paragraphe vide dans lequel sera inséré le cadre d'image (l'accrochage direct du cadre à la cellule, sans passer par un paragraphe, ne fonctionne pas, même si lpOD permet de le faire). Dans la cellule de droite, le texte de l'un des deux paragraphes contient un caractère `\n`, spécifiant un saut de ligne sans changement de paragraphe.

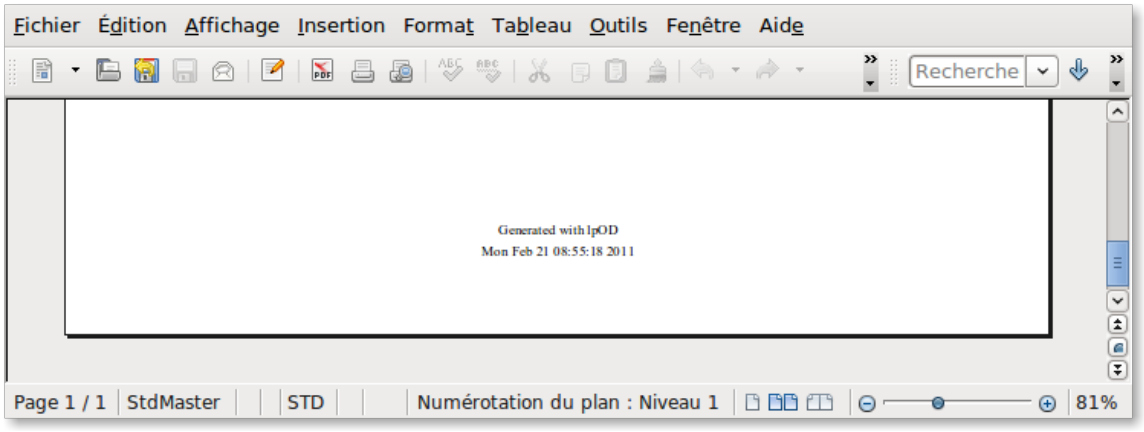
```
my $table = $entete->insert_element(
    odf_table->create("TableEntete", size => "1, 2");
my ($logo, $taille) = $doc->add_image_file('/images/logo_oasis.png');
$table->get_cell("A1")
    ->insert_element(odf_paragraph->create)
    ->insert_element(
          odf_frame->create(image => $logo, size => $taille)
          );
my $b1 = $table->get_cell("B1");
$b1->append_element(
    odf_paragraph->create(
          text  => "The lpOD Project",
          style => "GrasBleuCentré"
          )
    );
$b1->append_element(
    odf_paragraph->create(
          text  => "Open Document processing\nwith Perl",
          style => "ItaliqueCentré"
          );
```

Pour respecter notre cahier des charges, il faut aussi définir les styles des deux paragraphes placés dans la cellule de droite, qui ont des présentations différentes (dont aucune ne correspond au style de paragraphe par défaut). Pour que l'exercice soit complet, nous supposons qu'aucun des styles nécessaires n'est déjà enregistré dans le document, et nous devons donc les définir.

```
$doc->register_style(
        odf_create_style(
                'paragraph',
                name            => "GrasBleuCentré",
                align           => 'center',
                margin_top      => '0cm',
                margin_bottom   => '1cm'
                )
        )->set_properties(
                area            => 'text',
                size            => '200%',
                weight          => 'bold',
                color           => 'navy blue'
                );
$doc->register_style(
        odf_create_style(
                'paragraph',
                name            => "ItaliqueCentré",
                align           => 'center'
                )
        )->set_properties(
                area            => 'text',
                size            => '120%',
                style           => 'italic'
                );
```

Voilà pour l'entête.

Pour le pied de page, nous allons prendre pour modèle la copie d'écran suivante :



Ce pied contient deux paragraphes centrés, en caractères dont la taille est égale à 70 % de la taille par défaut. Le premier est un texte en dur, le second contient la date courante (le document qui a servi de modèle a été généré le 21/02/2011 comme on le voit sur l'image, mais ce n'est pas cette date-là qui importe). Nous devons donc créer et insérer ces paragraphes dans le contexte du pied de page, et bien sûr définir leur style (cette fois les deux paragraphes ont le même style).

```
$pied->append_element(
    odf_create_paragraph(
          text    => "Generated with lpOD",
          style   => "PetitCentré"
          )
    );
$pied->append_element(
    odf_create_paragraph(
          text    => scalar localtime,
          style   => "PetitCentré"
          )
    );
$doc->register_style(
    odf_style->create(
          'paragraph',
          name    => "PetitCentré",
          align   => 'center'
          )
    )->set_properties(
          area    => 'text',
          size    => '70%'
    );
```

À présent le modèle de page est entièrement spécifié. L'un des points importants à retenir est le fait que la définition d'un style de page comprend la création d'objets graphiques et textuels ordinaires correspondant à tout ce qui sera répété dans les entêtes et pieds de page. Il y a donc du *contenu* dans certains *styles*.

Mais tout n'est pas encore fini car, au début de ce programme, on a vu que ce modèle de page (`master page`) fait référence à une présentation de page (`page layout`), nommée « *PrésentationPage* », et qui reste à définir. La variété des options est infinie, de la couleur de fond au séparateur de notes de bas de page en passant par la disposition des entêtes et des pieds de pages. Mais ici nous en resterons à une description simple et classique, en spécifiant seulement un format *A4* (21×29,7 cm) et des marges de *16 millimètres*.

```
odf_style->create(
    'page layout',
    name        => "PrésentationPage",
    size        => "21cm, 29.7cm",
    margin      => "16mm"
    )
    ->register($doc);
```

C'est terminé (enfin). Cet exemple avait un double but : illustrer les possibilités de contrôle des styles de page de texte par lpOD, mais aussi démontrer que la mise en œuvre de ces possibilités implique un effort de codage substantiel et qu'il vaut donc généralement mieux réutiliser des styles de page créés avec un bon traitement de texte interactif que de s'acharner à tout fabriquer en Perl.

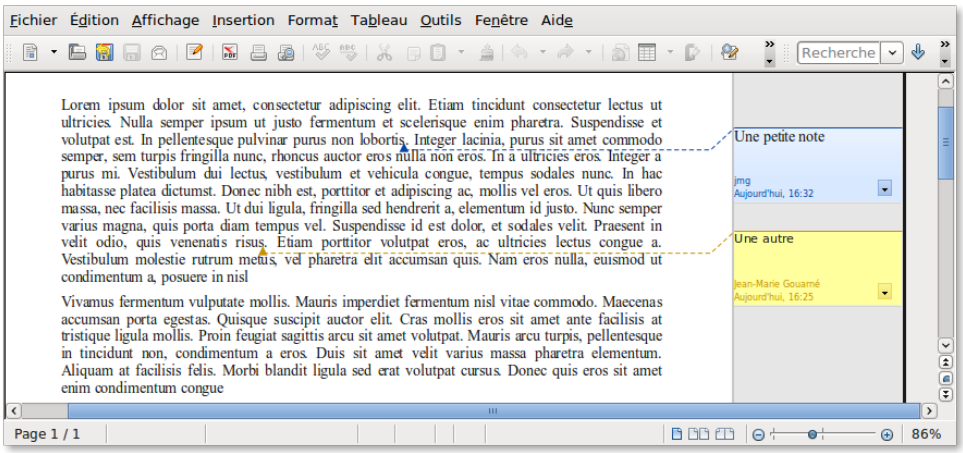
On trouvera une variante de cette construction de style dans le programme `lpod_test` déjà cité.

8 Repères, notes et autres balises

Un document peut être parsemé de diverses marques, dont certaines sont visibles et d'autres non, et qui peuvent être placées à diverses positions à l'intérieur du contenu. Ce sont par exemple des notes, des entrées bibliographiques, des entrées d'index, des signets, des champs variables, des commentaires[23].

8.1 Principe de placement des balises

Tous ces objets peuvent être placés selon une logique commune, en appelant des méthodes `set_xxx()`, où `xxx` dépend bien sûr du type de balise. Ces méthodes proviennent souvent de paragraphes ou de titres, parfois de cellules de tableaux ou d'autres conteneurs, selon l'objet auquel on veut attacher la balise.



Prenons comme exemple initial les *commentaires* ou *annotations* qu'on peut inscrire par exemple dans la marge d'un texte, et dont chacune a un point d'ancrage dans le texte, comme dans l'illustration ci-contre.

Le contexte de placement est ici un paragraphe, qu'on est supposé avoir sélectionné et à partir duquel on appelle la méthode `set_annotation()`. Du seul fait que cette méthode est appelée à partir d'un paragraphe, l'objet sera attaché à ce paragraphe. On pourrait aussi l'appeler, par exemple, à partir d'une cellule de table. Dans le cas du paragraphe, on peut aussi préciser la position du point d'ancrage dans le texte, à l'aide d'options permettant de choisir une distance (en nombre de caractères à partir du début ou de la fin du paragraphe) ou une expression de recherche dans le texte. Certains paramètres sont propres au type de balise : par exemple une annotation possède notamment un *texte* (`text`), une *date* (`date`) et un *auteur* (`author`). Mais ceux qui concernent la sélection du point d'ancrage dans le texte sont communs à toutes les méthodes qui placent des balises dans des zones de texte.

23 Ce sujet est documenté dans le chapitre `ODF::lpOD::TextElement`.

La première balise, ancrée juste après le mot « *lobortis* », peut avoir été placée par l'instruction suivante :

```
$paragraphe->set_annotation(
    text        => "Une petite note",
    after       => "lobortis"
    );
```

Ici, le texte seul est spécifié ; dans ce cas lpOD attribue par défaut la date courante comme date de l'annotation, et le nom de l'utilisateur du processus (selon le système d'exploitation) comme auteur.

Dans cet exemple l'option `after` spécifie la position immédiatement après le dernier caractère de la chaîne de recherche. Si cette chaîne n'est pas trouvée, la balise n'est pas insérée. On peut utiliser `before` plutôt que `after` pour placer l'ancrage avant et non pas après l'expression recherchée. Il est également possible de passer un paramètre `offset` indiquant la position numérique du point d'ancrage (si la valeur de `offset` est négative alors la position est comptée à partir de la fin). On peut même combiner `offset` avec `before` ou `after`, par exemple de la manière suivante :

```
$paragraphe->set_annotation(
    text        => "Une petite note",
    offset      => 100,
    before      => "Etiam"
    );
```

Dans l'instruction ci-dessus, la combinaison de paramètres indique que le point d'ancrage sera devant le premier mot « *Etiam* » apparaissant après le *100ème* caractère. Si `offset` était absent, ce serait devant la première apparition du mot « *Etiam* » (car il y en a deux dans le paragraphe).

Certaines balises peuvent être placées non seulement *devant* ou *derrière* une chaîne donnée, mais aussi *à sa place*. Ce n'est pas le cas des annotations, mais c'est par exemple le cas des champs. Prenons le cas d'un champ indiquant l'heure courante. Les champs se placent avec la méthode `set_field()` dont le premier argument est le type de champ. Si on veut que tous les mots « *Etiam* » du paragraphe soient remplacés par un affichage de l'heure, il faut écrire :

```
$paragraphe->set_field('time', replace => "Etiam");
```

Mais si l'objectif est de faire ce remplacement une et une seule fois, il faut fournir un `offset` (éventuellement à zéro). Combiné avec `replace`, `offset` signifie que le remplacement doit être fait pour la *première occurrence* de la chaîne indiquée en cherchant *à partir de la position donnée*. Si on s'intéresse à la première occurrence de tout le paragraphe il suffit de passer un `offset` égal à zéro.

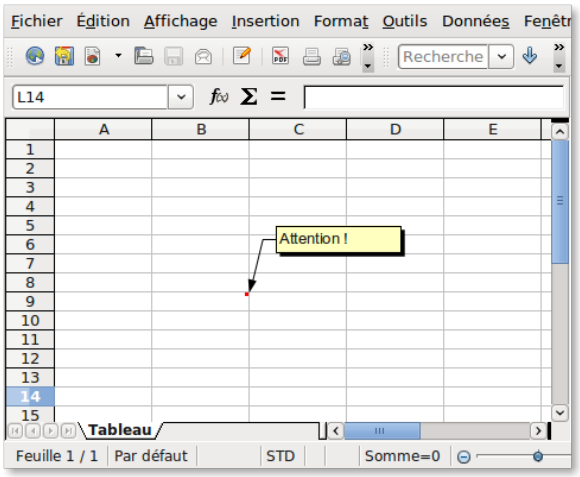
Nous n'irons pas plus loin ici sur les possibilités avancées de placement[24], et nous allons passer en revue à titre d'exemples quelques balises choisies parmi beaucoup d'autres.

8.2 Notes et commentaires

Nous venons de faire connaissance avec l'usage des *commentaires* ou *annotations* dans le contexte d'un paragraphe. Ce sont des objets visibles dans la fenêtre du traitement de texte mais qui sont placés hors des pages et non destinés à être imprimés. Un commentaire peut aussi être ancré à une cellule de table, comme dans l'exemple suivant.

24 On peut se référer, dans la page de manuel `ODF::lpOD::TextElement`, à la rubrique *Bookmarks*. Cette rubrique documente des paramètres de placement des *repères de texte* dont la majorité s'applique aussi aux autres balises.

```
$table->get_cell("B2")->set_annotation(
    text        => "Attention !",
    display     => TRUE
    );
```



L'option `display` sert ici à faire en sorte que le commentaire soit visible par défaut.

Le résultat de cette instruction est représenté par la copie d'écran ci-contre.

Les *notes*, malgré quelques caractéristiques communes et une parenté étymologique avec le mot *annotation*, ne doivent pas être confondues avec les étiquettes posées par `set_annotation()`. Une note s'installe par `set_note()` ; elle possède un point d'ancrage dans un bloc de texte et un contenu placé ailleurs. Mais le point d'ancrage est représenté dans le texte par un numéro ou une chaîne de caractères convenue, et le contenu de la note se place dans le document, soit en bas de la page, soit dans une page *ad hoc* en fin de document. Pour un usage classique, les paramètres sont les mêmes que pour les commentaires, mais un identifiant de note (unique) est requis comme premier argument. Ainsi, l'instruction suivante placera une note de bas de page automatiquement numérotée en séquence, avec un renvoi de note sous forme de numéro dans le texte (à l'image des notes de bas de page du présent article) :

```
$paragraphe->set_note(
    'note_001',
    text        => "Ce concept mérite une explication",
    after       => "Etiam tincidunt consectetur"
    );
```

Au-delà de cet usage simple, il existe des moyens de changer la présentation du renvoi de note (par exemple de remplacer les numéros par des astérisques, ou de modifier la logique de numérotation). Une option `class`, dont les valeurs possibles sont '`footnote`' et '`endnote`', permet de choisir entre note de bas de page et note de fin de document. Par défaut, lpOD choisit '`footnote`'. Autre point, concernant aussi bien les commentaires que les notes : le paramètre `text` utilisé ici pour spécifier un contenu, peut être remplacé par un paramètre `body`, permettant d'utiliser comme contenu une liste de paragraphes existants (généralement copiés d'ailleurs).

8.3 Champs variables

Les champs variables, qui s'installent dans les paragraphes avec `set_field()`, servent à insérer dans un texte des valeurs qui ne sont pas nécessairement figées et connues de l'auteur du texte. Par exemple, si nous souhaitons, dans un paragraphe, faire apparaître le numéro de page courant juste après le mot « page », nous pouvons écrire :

```
$paragraphe->set_field('page number', after => "page ");
```

Le premier argument de `set_field()` est un type de champ. Les types possibles sont assez nombreux[25]. Certains sont liés à des informations externes (exemples : `date`, `time`), d'autres à des variables ou à des métadonnées globales du document (exemples : `title`, `subject`, `creator`), d'autres au contexte (exemples : `page number`, `chapter`).

25 On peut produire la liste des types de champs supportés par lpOD via la méthode `odf_text_field->types`.

Il existe des paramètres optionnels propres à certains types de champs. Par exemple le type '`page continuation`', qui est destiné à afficher le numéro d'une page adjacente, demande un paramètre `select` indiquant s'il s'agit de la page précédente ou, comme c'est le cas ci-dessous, de la page suivante :

```
$paragaphe->set_field('page continuation', select => 'next');
```

Autre exemple de personnalisation : pour un champ `time`, le code suivant inscrit une date explicite, en l'occurrence la date et l'heure d'exécution, et active une option `fixed` signifiant que cette valeur ne bougera plus :

```
$paragaphe->set_field(
    'time',
    time_value  => iso_date,
    fixed       => TRUE,
    after       => "document généré à "
    );
```

lpOD propose un type, le type '`variable`', qui permet de mettre un ou plusieurs champs en relation avec une *variable* définie par l'utilisateur dans le contexte global du document. Tous les champs associés à une variable affichent la valeur de cette variable. Un champ de ce type doit avoir un paramètre `name` indiquant le nom (unique) de la variable. Il peut s'agir d'une variable déjà présente dans le document, mais lpOD permet aussi, à l'aide de la méthode `set_variable()` de la classe `odf_document`, de la créer. L'exemple suivant enregistre une variable « *Client* » dans le document, et place dans un paragraphe un champ destiné à l'afficher.

```
$doc->set_variable("Client", type => 'string', value => "Jules");
$paragraphe->set_field(
    'variable', name => "Client", after => "Le client se nomme "
    );
```

La variable est ici créée avec une valeur initiale spécifiée par le paramètre `value`. Mais il est possible de lire ou de modifier une variable, après l'avoir sélectionnée par son nom grâce à `get_variable()`. Une variable de document possède, comme une cellule de table, un type de donnée, une valeur, et supporte les méthodes `get_value()` et `set_value()`. L'exemple suivant recherche la variable « *Client* », puis affiche et change son contenu s'il la trouve, ou la crée si elle n'existait pas.

```
$client = "Jim";
$var = $doc->get_variable("Client");
if ($var) {
    say "Ancienne valeur : " . $var->get_value;
    $var->set_value($client);
    }
else      {
    say "Création de la variable";
    $var = $doc->set_variable(
          "Client", type => 'string', value => $client
          );
    }
```

Comme les cellules de tables, les champs peuvent être associés à des styles permettant de choisir les formats de dates ou de nombres (ces formats ne sont pas traités dans cet article).

8.4 Repères de texte

Un texte peut être marqué de balises invisibles permettant de signaler certaines positions ou zones pour des traitements ultérieurs. Les exemples les plus connus sont les *repères de texte* (ou *signets*) et les *marques*

d'index (ces dernières n'étant pas abordées ici, car leur présentation exigerait de parler aussi des différents types d'index, ce qui nous emmènerait trop loin).

On s'intéressera ici au cas le plus simple et le plus courant, à savoir les repères de texte positionnels (appelés *position bookmarks* dans la documentation de lpOD). Comme son nom l'indique, un repère de position est destiné à marquer une place. D'abord pour permettre à un utilisateur final (par exemple à travers le *navigateur* du traitement de texte) de retrouver un point particulier du texte. Mais aussi pour permettre à une application lpOD d'affecter une sorte d'identifiant à un élément de texte, de manière à pouvoir le sélectionner facilement. En effet, l'une des options de la méthode `get_paragraph()` permet de retrouver un paragraphe d'après le nom d'un repère de texte qu'il est censé contenir. Ainsi, l'instruction suivante sélectionne le paragraphe contenant le signet « *MaPlace* », s'il existe :

```
$para = $contexte->get_paragraph(bookmark => "MaPlace");
```

On peut aussi retrouver un signet par son nom et sélectionner un objet de plus haut niveau auquel il appartient, comme dans l'exemple suivant qui, après avoir trouvé un signet par son nom, vérifie s'il est situé à l'intérieur d'une cellule de table et dans ce cas affiche la position de la cellule dans sa table, et si ce n'est pas le cas se contente du paragraphe auquel il appartient.

```
my $conteneur;
my $marque = $doc->get_body->get_bookmark("MaPlace");
if ($marque)          {
    $conteneur = $marque->get_parent_cell;
    if ($conteneur)   {
          my ($table, $ligne, $colonne) = $conteneur->get_position;
          say "Trouvé dans $table à la position $ligne $colonne";
    }
    else              {
          $conteneur = $get_parent_paragraph;
          say $conteneur ?
                "Trouvé au milieu d'un paragraphe" :
                "Bizarre, un signet hors paragraphe ???";
          }
    }
else                  {
    say "Signet introuvable";
    }
```

L'installation d'un repère de texte se fait normalement à partir du paragraphe conteneur avec `set_bookmark()`. Le seul argument obligatoire est le nom unique du signet, donc l'instruction suivante suffit à placer le marqueur « *MaPlace* » dans un paragraphe préalablement choisi :

```
$paragraphe->set_bookmark("MaPlace");
```

Il est permis de poser un signet à une position particulière dans le texte d'un paragraphe. Pour ce faire, les options de position (notamment `offset`, `after`, `before`) sont exactement les mêmes que pour une annotation. Ainsi cette instruction place un signet cinq caractères avant la fin du paragraphe :

```
$paragraphe->set_bookmark("MaPlace", offset => -5);
```

9 Conclusion

Le choix des sujets traités et non traités ici est sans doute plus ou moins arbitraire. On peut cependant espérer qu'il sera utile à ceux qui débutent dans le traitement de documents bureautiques en Perl, ou tout simplement à ceux qui veulent savoir si ODF::lpOD a une chance d'être adapté à leurs besoins sans avoir à plonger dans le manuel.

Sachant que cet article n'est pas un manuel, il n'a pas répondu à une question légitime : que peut-on faire, et que ne peut-on pas faire, avec cette bibliothèque ?

La réponse n'est pas simple ; elle est à tiroirs multiples. Au sens le plus strict, lpOD est une caisse à outils permettant tout et n'importe quoi, notamment de fabriquer, de rechercher, de modifier ou de supprimer toute espèce d'élément (même non prévu) dans un document. lpOD embarque un moteur d'accès *XPath*[26] permettant de naviguer librement dans la structure des documents. L'utilisateur a donc de ce fait « tous les pouvoirs », mais l'usage des fonctionnalités de bas niveau exige une certaine familiarité avec *XML*, *XPath* et la spécification *ODF*. Si, à l'autre extrémité, on veut s'en tenir aux méthodes de haut niveau, ignorer complètement *XML* et *XPath*, s'abstenir de toute investigation relative à la norme *ODF*, et ne rien lire d'autre sur ce sujet que la documentation embarquée dans la distribution ODF::lpOD, le périmètre est plus circonscrit (quoique largement suffisant pour beaucoup d'applications, notamment s'il s'agit d'exploiter des documents existants ou de travailler à partir de modèles, et non de créer de toutes pièces des documents complexes). Chacun placera le curseur là où il veut entre ces deux extrêmes, selon ses objectifs et sa façon de travailler.

Pour un usage avancé, il peut être utile de se référer ponctuellement à la norme *ODF*[27]. Par exemple pour connaître toutes les valeurs possibles d'une certaine propriété d'un certain objet, ou pour savoir si on a le droit d'insérer tel objet dans, avant ou après tel autre objet. Actuellement, la version *1.1* de cette norme est probablement la mieux appropriée dans un premier temps[28]. Pour la mise au point d'applications avancées, les *addicts* du XML peuvent aussi s'amuser à décortiquer les fichiers générés via lpOD et les comparer aux fichiers ODF que peut produire un logiciel bureautique populaire comme *OpenOffice.org*, *LibreOffice* ou *Microsoft Office*. À cet égard il faut être vigilant car aucun logiciel bureautique ne respecte totalement la norme *ODF*, donc dans les situations limites, même quand on est sûr d'avoir généré un document « légal », il vaut mieux s'assurer que le résultat est compatible avec l'outil de travail de l'utilisateur.

Bon courage…

26 Fourni par `XML::Twig`. Voir la documentation de `XML::Twig` pour en connaître la couverture fonctionnelle et les limites.

27 La documentation officielle sur ODF est disponible (notamment en format ODF) via http://www.oasis-open.org/committees/office.

28 La version *1.2* apporte surtout des *extensions* par rapport à la version *1.1*, dont elle ne remet pas en question l'essentiel. Un document produit au format *ODF 1.1* sera en principe repris sans perte par un logiciel compatible *ODF 1.2*, tandis que l'inverse n'est pas certain. De plus, la normalisation ayant une tendance inflationniste, le texte de la spécification *ODF 1.1* est nettement plus abordable !