Production et exploitation de documents bureautiques

Une interface Perl/OpenDocument

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

EN BREF *Les atouts classiques de Perl en matière de traitement de "texte plat" sont bien connus. Mais à présent, le succès du format* OpenDocument *et le développement d'interfaces de programmation appropriées permettent à Perl de s'attaquer plus facilement au "texte riche", c'est-à-dire aux documents bureautiques. Cet article, après une mise au point sur ce qu'est* OpenDocument *et sur le parti que peut en tirer un programmeur, présente un échantillon des possibilités offertes dans ce domaine par le module* Open OpenDocument Connector *(OpenOffice::OODoc).*

1 Introduction

Le format OpenDocument (qui est notamment celui d'*OpenOffice.org*, de *KOffice* et de quelques autres) n'est pas seulement un format de convergence entre suites bureautiques. C'est aussi un bon allié pour les développeurs d'applications dont la vocation est de produire ou d'exploiter des documents. Ce "standard ouvert" pour employer le vocabulaire d'un débat technico-politique très médiatisé de nos jours, a déjà donné lieu à des interfaces de programmation opérationnelles, comme celle que présente cet article.

Mais d'abord, sachant qu'il vaut mieux enfoncer dix portes ouvertes que d'oublier une porte fermée, il convient de re-préciser ce qu'est OpenDocument et ce qu'il apporte de particulier au développement d'applications d'entreprise.

1.1 Les 6 ans d'histoire d'un format ouvert

À l'époque du lancement d'OpenOffice.org, il y a six ans déjà, la plupart des observateurs du marché se sont contentés de remarquer l'aspect le plus immédiatement visible de cet événement, à savoir l'ouverture du code de StarOffice et, de ce fait, la consécration du plus grand projet de logiciel bureautique libre jamais entrepris.

Mais un événement peut en cacher un autre. En l'occurrence, OpenOffice.org n'était pas seulement une nouvelle suite bureautique, c'était aussi la mise en oeuvre d'un nouveau concept, celui de format documentaire ouvert. En effet, avant même que le logiciel soit disponible dans sa version 1.0, la spécification de son format d'enregistrement natif était rédigée et surtout publiée. Cette innovation a été manifestement sous-estimée sur le moment. On a d'abord vu dans OpenOffice.org un logiciel bureautique gratuit susceptible, peut-être, de rétablir un jour un semblant d'animation sur un marché monopolistique essentiellement marqué par la passivité résignée des utilisateurs, sans plus. L'autre volet de l'affaire n'a pas été clairement évalué au-delà d'un cercle restreint d'observateurs. Or il s'agissait ni plus ni moins que de l'arrivée de ce qui allait devenir ce qu'on appelle maintenant, en Français, un "standard ouvert"[1]. Une innovation exceptionnelle dans le monde de la bureautique.

Un débat sur la notion de "standard ouvert" ou sur les conditions juridiques d'utilisation de la spécification du format OpenOffice.org 1.0 telles qu'elles étaient lors de sa publication nous écarterait de notre sujet. Mais on peut sans trop de risque d'erreur affirmer que, dès son origine et bien avant d'avoir acquis le statut légal de standard, ce format était indiscutablement "ouvert", en ce sens que sa spécification était (et reste) accessible à tous, et que le développement d'applications de toutes natures par des tierces parties était libre de droits. Il ne s'agissait pas encore pour autant d'un standard, car ce format restait l'émanation du projet de suite bureautique dont il portait le nom. Mais, sur ce terrain, les responsables du projet n'ont pas perdu de temps.

1 "On entend par standard ouvert tout protocole de communication, d'interconnexion ou d'échange et tout format de données interopérable et dont les spécifications techniques sont publiques et sans restriction d'accès ni de mise en oeuvre" (*Loi n° 2004-575 du 21 juin 2004, dite "Loi pour la confiance dans l'économie numérique"*).

Dès la fin 2002, des discussions s'engageaient entre Sun et le consortium OASIS[2]. Ce dernier, conscient des ambiguïtés que laissait planer l'homonymie entre le logiciel d'application et le format de fichiers, décida avec sagesse de choisir un nouveau nom pour le second, et apporta quelques retouches à la spécification, sans pour autant remettre en question les principes essentiels. Et le 1er mai 2005, à l'issue d'un processus ayant impliqué des acteurs de différents bords (mais, hélas, pas Microsoft), le consortium mettait à la disposition du public le digne successeur du format OpenOffice.org 1.0, sous la désignation officielle "*OASIS Open Document Format for Office Applications, version 1.0*". Une désignation communément abrégée en *OpenDocument* ou *ODF*. Un nouveau seuil était franchi : la spécification du format appartenait désormais à un organisme de standardisation multipartite, et sortait du giron de Sun. La suite bureautique OpenOffice.org, dont la version 2.0 est sortie officiellement en octobre 2005, fut bien sûr la première à supporter nativement le format OpenDocument. Mais un nouveau principe était consacré et soutenu par l'OASIS : la définition des formats d'enregistrement des documents bureautiques doit être indépendante de tout éditeur de logiciel bureautique.

Mais ce n'était qu'une étape. Une étape décisive, certes, mais il fallait faire encore mieux. Car du point de vue des pouvoirs publics, l'OASIS n'a pas le statut d'instance de normalisation, et les seuls vrais standards internationaux sont ceux de l'ISO[3]. Là non plus, les choses n'ont pas traîné. Au terme d'une procédure de certification commencée en septembre 2005, le format OpenDocument a été validé le 1er mai 2006 en tant que standard international sous la référence ISO 26300. Une consécration essentielle puisqu'elle place l'ODF au même rang que le PDF et le HTML. Ce qui veut dire que le format OpenDocument, qui n'est évidemment pas plus figé qu'un autre, évoluera désormais dans le cadre d'un processus public, formel et débarrassé des contraintes liées à l'agenda d'un fournisseur unique. Les développeurs que nous sommes peuvent donc investir sur lui sans s'interroger à tout instant sur la pérennité de leurs applications.

A l'heure où cet article est rédigé, la version 1.1 d'OpenDocument est déjà arrêtée et devrait prochainement être adoptée comme standard OASIS. Les travaux de la version 1.2 sont quant à eux déjà bien engagés. Le standard, encore jeune, comporte quelques zones grises et mérite d'être complété. Mais la pérennité étant un de ses objectifs majeurs, et l'OASIS n'ayant aucun intérêt à créer des ruptures de compatibilité, les nouvelles versions sont appelées à apporter des extensions et non pas des remises en question. Surtout, aucune nouvelle version ne peut arriver par surprise : les travaux intermédiaires du comité technique chargé, au sein de l'OASIS, de l'élaboration du standard, sont largement diffusés[4].

1.2 Caractéristiques essentielles d'OpenDocument

La documentation consacrée à ce format, à commencer par la spécification[5] elle-même et quelques articles de fond[6], est suffisamment connue, et il serait donc aussi inutile qu'ennuyeux de la commenter en détail ici. Pour la bonne compréhension de la suite, on se bornera à en introduire (ou à en rappeler) les quelques principes essentiels.

1.2.1 Une archive compressée

Un fichier ODF, d'un point de vue technique, est une archive zip parfaitement classique dont le contenu s'extrait sans formalité particulière avec *unzip* ou tout autre utilitaire comparable.

Le contenu de cette archive est multiforme. Certains membres sont des fichiers XML et sont obligatoirement présents. D'autres, optionnels, peuvent être des documents XML ou des fichiers binaires, selon les caractéristiques de l'application. Dans cet article, nous ne parlerons que des trois membres XML les plus intéressants pour le développement d'applications, ainsi que des modalités de stockage des images.

1.2.2 Contenu

Dans une archive ODF, le contenu du document est associé à un et un seul des composants XML, nommé comme il se doit *content.xml*. Encore faut-il s'entendre sur ce qu'on appelle le contenu.

Pour simplifier, disons que le contenu (au sens ODF) correspond à tout ce qui apparaît dans le corps des pages, autrement dit tout sauf les en-têtes, pieds de pages et fonds de page. Ce contenu peut se composer de toutes sortes d'objets (texte plat, tableaux, listes, images, par exemple).

2 Organization for the Advancement of Structured Information Standards (http://www.oasis-open.org)
3 International Organization for Standardisation (http://www.iso.org)
4 cf. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office
5 http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf
6 Voir notamment Wikipedia (http://fr.wikipedia.org/wiki/OpenDocument)

En pratique, cependant, les choses sont un peu plus compliquées.

D'abord, certains éléments peuvent être physiquement stockés ailleurs que dans *content.xml* alors qu'ils apparaissent dans le contenu. Le meilleur exemple est celui des images. La spécification ODF laisse aux applications le choix entre deux modes de stockage. Une image peut être directement encodée en Base64 à sa place en tant qu'élément XML, et dans ce cas elle est incorporée dans *content.xml*. Elle peut aussi être stockée en format binaire (JPEG, PNG, BMP ou tout autre format graphique au gré de l'utilisateur) en tant que membre indépendant non XML, et dans ce cas elle est simplement représentée par un lien dans *content.xml*. Il existe aussi une troisième variante (qui n'est qu'une extension de la seconde), consistant à ne pas embarquer l'image dans le fichier ODF, et à enregistrer simplement son URL (*http*, *ftp*, *file* ou tout autre protocole supporté par l'application). OpenOffice.org pratique la seconde et la troisième variantes (donc ne stocke les images que par référence dans *content.xml*) ; l'interface Perl OpenOffice::OODoc fait de même. Nous ne parlerons donc plus, dans la suite de cet article, de la première option.

Ensuite, certains éléments non visibles pour le lecteur peuvent être stockés dans *content.xml*. C'est le cas notamment de certains descripteurs de styles dits *automatiques*. Pour définir simplement ce qu'est un style automatique, je vais partir d'un exemple tiré du présent paragraphe de cet article : à la fin de la phrase précédente, j'ai mis en italiques le mot "automatiques" ; pour ce faire, je n'ai eu qu'à sélectionner le mot et à cliquer sur le bouton "*I*" dans la barre d'outils d'OpenOffice.org, sans passer par toute la séquence de création explicite d'un style. Pourtant, à travers cette procédure simple, j'ai en réalité provoqué la création d'un style, qui ne sera appliqué qu'une seule fois, n'a pas de nom visible et n'apparaîtra pas dans le gestionnaire de styles du logiciel bureautique. Un style ainsi créé, qui pourrait aussi bien être défini comme un "style sans nom" et qu'on appelle "style automatique" dans le vocabulaire ODF, est représenté par un élément XML distinct de l'objet auquel il s'applique mais stocké dans le même membre de l'archive. Bien entendu, un style automatique, comme n'importe quel style, doit en réalité avoir un nom unique, mais ce nom n'est qu'une clé technique générée par le logiciel et il peut changer d'un enregistrement à un autre.

Il n'en va pas de même pour les styles "*nommés*".

1.2.3 Styles

Les styles "nommés" sont ceux qui peuvent être appliqués, modifiés, créés ou supprimés par un utilisateur final à travers un tableur, un traitement de textes, un support de présentation ou autre outil bureautique classique. Concrètement, ce sont ceux qui apparaissent dans le gestionnaire de styles d'un OpenOffice.org ou d'un KOffice. D'où la nécessité, pour chacun de ces styles, d'avoir un nom visible, unique et persistant.

Pour le programmeur s'attaquant directement aux fichiers ODF, il n'y a pas de différence fondamentale entre styles automatiques et styles nommés, car les éléments XML qui les décrivent ont la même structure. Mais le caractère automatique ou nommé d'un style doit souvent être connu pour déterminer où est stocké le descripteur XML correspondant.

Une première règle est à noter immédiatement : tout style *nommé* est décrit dans un membre de l'archive dédié à cet effet et qui s'appelle, sans surprise, *styles.xml*. Un style qui se "voit" et qui peut se configurer à travers le gestionnaire de styles figure donc, sous la forme d'un élément XML bien entendu, dans ce membre de l'archive.

Peut-on conclure de ce qui précède que tous les objets visibles (textuels et autres) et tous les styles automatiques sont dans *content.xml* et que *styles.xml* ne contient que les styles nommés ? Non, ce serait trop simple...

Le meilleur contre-exemple nous est donné par le contenu qui apparaît (éventuellement) dans les entêtes et les pieds de pages d'un document. Ce contenu est composé d'éléments de même nature que ceux qui apparaissent dans le corps du document. Ainsi, une ligne de texte et un logo qui se répètent dans une entête sont respectivement décrits par un élément XML de type paragraphe et par un élément XML de type image (OpenDocument est très cohérent avec lui-même sur ce plan comme sur d'autres : la grammaire XML de description d'un objet ne dépend pas du contexte dans lequel cet objet est présenté). Mais le contenu d'une entête fait partie de la définition d'un style de page, et n'appartient donc pas au corps du document. Or les styles de pages sont généralement des styles nommés[7], donc décrits dans *styles.xml*... dans lequel on peut donc trouver des éléments éditables.

Mais ce n'est pas tout. Parmi les objets que contient une en-tête ou un pied de page, certains peuvent faire l'objet de fioritures de présentation qui font appel à des styles automatiques. Or un style automatique est stocké dans le même membre de l'archive que l'objet pour lequel il a été défini. On peut donc trouver aussi dans *styles.xml* des styles automatiques.

7 En pratique, un style de page automatique serait difficilement contrôlable via un logiciel bureautique.

Toujours à propos des styles de pages, j'ai évoqué la présence d'un logo dans une entête. Lorsqu'une image apparaît dans une entête, un pied de page ou un arrière-plan de page, cela veut dire qu'elle fait partie de la définition d'un style de page. Mais à part cela, elle est stockée exactement comme une image du corps du document ; je renvoie donc sur ce point à ce que j'ai dit sur les images à propos de *content.xml*.

Qu'ils soient automatiques ou nommés, les styles se répartissent en catégories très diverses, chacune caractérisée par une structure XML appropriée. Selon qu'un style est applicable à une image, un paragraphe, une cellule de tableau, une liste à puces, une note de bas de page, ou même à une mise en page, on imagine facilement que sa structure de données ne sera pas la même. Pour aggraver cette complexité, il faut ajouter que, très souvent, la présentation d'un objet ne peut être contrôlée que par la combinaison d'un ensemble de styles.

On peut citer en exemple le cas des tableaux. Les caractéristiques communes à l'ensemble du tableau dépendent d'un style de table. Ensuite chaque ligne est soumise à un style de ligne, et chaque colonne à un style de colonne. Les caractéristiques de présentation propres à chaque cellule dépendent, elles, d'un style de cellule. Enfin, sachant que le contenu d'une cellule de texte est un paragraphe ordinaire, ce dernier possède aussi un style de paragraphe... et ainsi de suite. Il faut simplement retenir que la description complète de l'aspect d'un tableau utilise souvent un nombre de styles supérieur au nombre de cellules du tableau, ce qui veut dire que la présentation d'un tableau représente souvent plus de code XML que son contenu[8].

Je n'en dirai pas plus ici sur les aspects théoriques de la gestion des styles en ODF, qui représentent un sujet trop foisonnant pour le format de cet article. Nous reviendrons de manière concrète, avec des exemples, sur la manipulation de certains styles en Perl.

1.2.4 Métadonnées

Tous les outils bureautiques (généralement via une commande comme *Fichier/Propriétés*) offrent à l'utilisateur une interface permettant de consulter et (dans une certaine mesure) de modifier ce qu'on appelle les *métadonnées*, c'est-à-dire certaines propriétés générales utiles pour l'identification, le classement, la recherche ou la gestion de version. Parmi ces propriétés figurent notamment l'auteur, le titre, le sujet, parfois une description longue, une liste de mots-clés, la date de création, et bien d'autres. En format OpenDocument, ces informations sont stockées dans un membre séparé du contenu, nommé *meta.xml*.

Le composant *meta.xml* est généralement très court (quelques dizaines de lignes), et facile à manier. Une application d'archivage ou de gestion documentaire peut l'exploiter très rapidement, puisqu'il peut être traité séparément sans demander beaucoup d'efforts au parseur XML. De plus, l'une des règles du jeu veut que *meta.xml* ne soit, lui, jamais crypté par les applications compatibles OpenDocument, alors que *content.xml* et *styles.xml* sont illisibles si l'utilisateur a choisi de protéger le document par mot de passe[9].

L'interface OpenOffice::OODoc offre une série d'accesseurs permettant de consulter et de modifier très simplement n'importe quel élément des méta-données... et même de faire des choses qu'aucun logiciel bureautique raisonnable ne doit permettre à un utilisateur final (exemple : changer la date de création ou l'auteur de la version initiale, forcer des valeurs fantaisistes dans les statistiques du document, etc). Du fait de leur simplicité d'utilisation, on n'insistera pas sur les méta-données dans la suite de cet article (plutôt focalisée sur le contenu et les styles), sauf à les évoquer incidemment.

1.2.5 Autres informations

Une archive ODF contient presque toujours d'autres membres que *content.xml*, *styles.xml* et *meta.xml*.

Il peut y avoir, bien sûr, des images en format graphique binaire. Celles-ci, le cas échéant, sont toutes consignées dans un répertoire *Pictures*. On peut d'ailleurs très facilement, par exemple, les extraire avec *unzip*, les modifier avec *The Gimp*, puis les remettre en place avec *zip* et enfin recharger le document sous OpenOffice.org pour voir le résultat (c'est une idée de démonstration très visuelle, qui peut être appliquée en réunion pour illustrer les notions de format ouvert et d'interopérabilité). Bien entendu, OpenOffice::OODoc offre des accesseurs qui permettent d'importer et d'exporter les images sans passer par le shell.

8 C'est pourquoi, bien que OpenOffice::OODoc permette de le faire, je déconseille en général de créer par programme des tableaux complexes avec toute leur présentation. Il est bien plus efficace de copier (par programme) des tableaux et des styles de tableaux existants, puis de les modifier, que de tout faire à partir de rien.

9 Bien entendu, rien n'empêche une application *extérieure* au logiciel bureautique de venir crypter tout ce qu'elle veut après l'enregistrement du document. Disons simplement, par exemple, que si on enregistre un document sous OpenOffice.org en choisissant de le protéger par mot de passe, *meta.xml* reste disponible en clair.

L'archive contient éventuellement des scripts ou des macro-instructions (fort heureusement séparées du contenu, des styles et des méta-données). Je n'en dirai pas plus sur ce sujet, car il nous conduirait tout droit dans une zone d'ombre du standard. En effet, ODF n'interdit pas les macros, mais les langages de macros sont eux-mêmes exclus du standard. En d'autres termes, les macros ne sont pas portables, même entre deux applications compatibles ODF[10].

Un autre membre, *settings.xml*, contient des informations qui ne concernent pas directement le document mais qui permettent au logiciel bureautique de reconstituer le contexte de travail de l'utilisateur quand il revient sur le document. (Exemple : quelle était l'imprimante par défaut pour ce document, quelle était la zone visible à l'écran quand il a été enregistré, etc).

L'archive contient aussi un petit composant *mimetype*, un fichier texte d'une seule ligne, qui indique le type MIME. Pour un document ODF-Text, par exemple, ce type est

```
application/vnd.oasis.opendocument.text
```

On peut citer, pour finir et sans épuiser toute la liste, le composant XML dans lequel est consigné l'inventaire des autres membres de l'archive, c'est-à-dire le manifeste. Il se nomme, on s'en serait douté, *manifest.xml* mais, au lieu d'être à la racine de l'archive, il est rangé dans un dossier *META-INF*.

2 Origine, vocation et architecture du module OODoc

Le module Perl OpenOffice::OODoc, alias *Perl Open OpenDocument Connector*, ou pour faire plus court *OODoc*, est sensiblement plus ancien que le format ODF lui-même. Ce module est en effet issu d'un projet initialement privé, basé sur la spécification OpenOffice.org 1.0, développé chez Genicorp[11] en 2002 et destiné à des applications de production automatique de documents. Les premiers utilisateurs appartenaient aux secteurs du droit, de la santé et de l'industrie alimentaire ; depuis, les usages et les domaines d'application se sont multipliés. Le code a été rendu libre, puis diffusé via le CPAN, à partir du début de 2003 (version 1.102). Une refonte technique des "couches basses" a été réalisée en début 2005, de manière à intégrer XML::Twig[12] comme interface de navigation XML et à améliorer les performances en traitement de gros volumes (version 1.301). Enfin, une adaptation majeure a été livrée à la mi-2005 (version 2.001) de manière à prendre en charge le format OpenDocument[13] qui venait tout juste d'être validé par l'OASIS.

Les objectifs fonctionnels de cette boîte à outils sont de deux sortes :

1. Faciliter la recherche, le contrôle et l'extraction automatiques de données contenues dans les documents ;

2. Faciliter la génération ou la mise à jour automatique de documents par des applications non bureautiques, par exemple à des fins de reporting, de publipostage, ou pour garantir le respect de règles strictes en matière de contenu et de *workflow* documentaire.

En résumé, OODoc est une interface entre applications de production et documents bureautiques.

Pour atteindre ses objectifs, le module ne comporte rien de révolutionnaire, mais offre plusieurs sortes de services :

1. D'abord, il évite au développeur d'avoir à gérer explicitement la compression et la décompression des fichiers ; toute la logistique de base est encapsulée dans des "connecteurs" qui permettent, à la limite, d'ignorer le format des fichiers ;

2. Ensuite, il offre une collection importante d'accesseurs prédéfinis, permettant de retrouver et de modifier les éléments de contenu et de présentation les plus utilisés sans avoir à connaître leur position dans la structure XML[14].

10 En fait, OpenOffice::OODoc s'intéresse aux macros... lorsqu'il s'agit de les mettre dans le collimateur ! La distribution CPAN contient en effet un utilitaire (*oofindbasic*) permettant de détecter, d'exporter et/ou de détruire les macros OOoBasic en respectant tout le reste. Cet utilitaire peut servir de base au développement d'un filtre associé, par exemple, à un firewall ou à un serveur de messagerie, et capable d'interdire le passage des macros sans arrêter les documents. Voir "*man oofindbasic*".

11 http://www.genicorp.fr

12 http://search.cpan.org/dist/XML-Twig

13 Remarque : la prise en charge d'OpenDocument s'est faite sans rupture, car l'ancien format OpenOffice.org est encore supporté.

14 Le nombre de ces accesseurs peut lui-même toutefois entraîner une difficulté : l'API OpenOffice::OODoc comporte environ 400 méthodes documentées... d'où un manuel de référence assez volumineux et, disons-le, plutôt rébarbatif quand on l'aborde sans avoir préparé le terrain en lisant, par exemple, un article comme celui-ci.

OODoc est donc une interface d'*accès aux données* contenues dans les documents, mais ce n'est en aucun cas un outil de *transformation* de documents. Ce point mérite d'être précisé, car il s'agit d'une question récurrente. L'accès en lecture et écriture à tout élément de contenu ou de présentation d'un fichier OpenDocument est une chose, la conversion d'un fichier OpenDocument en un fichier PDF, Microsoft Office ou HTML (et vice-versa) en est une autre[15].

OpenOffice::OODoc s'appuie, pour la gestion physique des fichiers ODF, sur l'incontournable *Archive::Zip*, et pour l'accès au contenu XML, sur *XML::Twig.* Les entrées-sorties sont traitées de manière transparente, et les applications n'ont pas, normalement, à faire directement appel aux fonctionnalités d'Archive::Zip. En revanche, toutes les méthodes de XML::Twig sont directement applicables et peuvent être utilisées efficacement en combinaison avec celles d'OpenOffice::OODoc.

L'API OpenOffice::OODoc est (comme celles d'Archive::Zip et de XML::Twig) orientée objet.

Pour accéder à un document, l'application utilisatrice commence par instancier un objet de classe *Document*, que nous pouvons appeler un "connecteur". Chaque connecteur est lié non pas à un fichier OpenDocument, mais à un membre particulier (*content.xml*, *styles.xml*, *meta.xml* ou autre), de sorte que, comme on le verra bientôt à travers des exemples, l'application doit créer deux ou plusieurs connecteurs pour un même fichier si elle veut accéder simultanément au contenu, aux styles nommés et/ou aux métadonnées. Dans la suite de cet article, le mot document, sauf précision contraire, devra être pris comme synonyme de connecteur lié à un membre. On utilisera plutôt le mot *archive* (comme dans Archive::Zip) pour désigner l'ensemble du contenu du fichier. La classe *Document* possède une large gamme de méthodes de recherche, de modification, de création et de suppression d'*éléments*.

Le mot *élément* reviendra souvent dans cet article (et il est omniprésent dans le vocabulaire de l'API). Un *élément* est un composant documentaire de contenu ou de présentation. Physiquement, il correspond à un élément XML du fichier ODF. Dans l'API OODoc, c'est aussi une classe qui possède ses propres méthodes[16]. Un élément peut être, par exemple, un style de page, un champ variable, une ligne de tableau, une section, un paragraphe, ou tout autre objet ayant un sens documentaire immédiat, ou correspondant à un niveau hiérarchique interne (par exemple le corps du document tout entier).

Le module OpenOffice::OODoc au sens strict ne contient pas grand chose. Son rôle se limite à fournir quelques fonctions communes et surtout à intégrer une librairie qui se compose de plusieurs modules, dont chacun correspond à une classe, à savoir :

- *OpenOffice::OODoc::File*, qui gère les opérations en rapport avec les fichiers ODF, en s'appuyant sur Archive::Zip (que les applications n'ont pas à invoquer directement) ;

- *OpenOffice::OODoc::XPath*, qui fournit une interface générale d'accès au contenu XML des documents ; l'implémentation de ce module est basée sur XML::Twig, et fait largement appel à des expressions *XPath* (d'où son nom) ;

- *OpenOffice::OODoc::Document*, qui hérite du précédent et qui apporte un ensemble de méthodes de plus haut niveau (plus conviviales que des requêtes XPath), focalisées sur une sélection d'objets documentaires prédéfinis ; détail important, ce module Document est en fait une composition de trois modules qui ne sont pas normalement invoqués directement, à savoir *OpenOffice::OODoc::Text*, *OpenOffice::OODoc::Styles* et *OpenOffice::OODoc::Image* ;

- *OpenOffice::OODoc::Meta*, lui aussi dérivé d'*OpenOffice::OODoc::XPath*, mais spécialisé dans la manipulation des méta-données.

Les applications n'ont explicitement besoin d'invoquer que les modules *Document*, *Meta* et éventuellement *File*.

Toutefois, chaque module correspond à un chapitre du manuel de référence et il faut savoir que les commandes "`man OpenOffice::OODoc`" et "`man OpenOffice::OODoc::Document`" ne retournent qu'une infime partie de ce manuel. Le manuel d'*OpenOffice::OODoc::Text* (le plus fourni) présente les méthodes de la classe *Document* consacrées à la gestion des conteneurs de textes, ou des conteneurs de haut niveau capables d'héberger des contenus textuels ou non textuels (y compris les tableaux ou les sections). Les pages *OpenOffice::OODoc::Styles* correspondent évidemment à la gestion des styles et enfin le chapitre *OpenOffice::OODoc::Image* décrit les méthodes relatives aux images. Quant aux méthodes communes, utilisables pour traiter tout type d'élément, elles sont présentées dans le chapitre *OpenOffice::OODoc::XPath.* Cette multiplication de

15 On peut cependant noter que le module OpenOffice::OODoc est souvent utilisé pour produire ou enrichir des fichiers au format OpenDocument qui sont ensuite, *avec d'autres outils*, convertis dans d'autres formats.

16 Pour les lecteurs connaissant XML::Twig, la classe OpenOffice::OODoc::Element est directement dérivée de XML::Twig::Elt, dont elle hérite toutes les fonctionnalités.

manuels n'est pas pour simplifier la vie du néophyte. Heureusement, la distribution comprend un chapitre *OpenOffice::OODoc::Intro* qui s'efforce de donner une vision d'ensemble simple, et il existe aussi (hors distribution CPAN) un guide introductif en Français (voir bibliographie).

La connaissance de XML::Twig et de la syntaxe XPath n'est pas nécessaire pour utiliser OpenOffice::OODoc. Cependant cette connaissance peut être très utile pour le développement d'applications avancées ou pour étendre les possibilités actuelles de l'interface.

3 Fonctionnement d'un connecteur de document

Le traitement d'un document commence normalement par la création d'un connecteur OpenOffice::OODoc::Document. Cette création peut s'effectuer à l'aide de la fonction `ooDocument()`. Celle-ci, toutefois, doit être liée à une instance d'archive ODF (correspondant au fichier physique) elle-même crée à l'aide du constructeur `ooFile()`. Mais avant de laisser le lecteur imaginer des complications inutiles, commençons par un exemple.

```
my $archive = ooFile("MonFichier.odt");
my $doc = ooDocument(archive => $archive);
```

La séquence ci-dessus commence par créer un objet `$archive` qui va représenter un accès physique au fichier. Notons au passage que la fonction `ooFile()` crée elle-même une instance d'Archive::Zip à travers laquelle seront gérées de façon transparente toutes les entrées-sorties (et les compressions-décompressions qui vont avec). La seconde instruction est un raccourci qui crée un connecteur de document basé sur l'archive préalablement initialisée. Mais il manque potentiellement quelque chose. En effet, comme on l'a annoncé dans la section précédente, un connecteur permet d'accéder à un et un seul membre de l'archive. L'exemple ci-dessus est syntaxiquement accepté, car le constructeur `ooDocument()` considère que, par défaut, l'utilisateur veut accéder au contenu. Mais pour être parfaitement rigoureux (et rendre le programme plus auto-documenté), il vaut mieux écrire plus précisément :

```
my $doc = ooDocument
      (
      archive => $archive,
      member  => "content"
      );
```

Cette dernière forme spécifie explicitement un accès au corps du document (*content.xml*). Dans une application qui accéderait simultanément aux styles nommés, au corps de document et aux métadonnées, le code serait le suivant :

```
my $archive  = ooFile("MonFichier.odt");
my $contenu  = ooDocument
      (
      archive => $archive,
      member  => "content"
      );
my $styles   = ooDocument
      (
      archive => $archive,
      member  => "styles"
      );
my $meta     = ooMeta
      (
      archive => $archive
      );
```

Après la séquence d'initialisation ci-dessus, le programme dispose de trois connecteurs liés au même fichier ODF par l'intermédiaire de l'objet `$archive`. On notera que les deux premiers, `$contenu` et `$styles`, sont construits par `ooDocument()` et sont donc tous deux des objets de même classe (OpenOffice::OODoc::Document), tandis que le troisième, créé via `ooMeta()` est d'une autre classe (OpenOffice::OODoc::Meta). Ceci parce que le traitement des métadonnées se contente d'un connecteur beaucoup plus simple, alors qu'il y a beaucoup de choses en commun entre la gestion des styles et celle du contenu.

Les connecteurs ainsi créés donnent chacun accès à une vaste collection de méthodes, sur lesquelles nous reviendrons dans les chapitres suivants, et qui permettent de consulter et de modifier tout élément dans l'espace de travail (c'est-à-dire le membre de l'archive) auquel il correspond.

En cas de modification d'un espace de travail (création, suppression ou mise à jour d'éléments), les changements sont réalisés dans l'espace mémoire du connecteur concerné. Pour rendre ces changements persistants, il suffit d'appeler tout simplement la méthode `save()` de l'objet initialisé par `ooFile()` :

```
$archive->save();
```

Par défaut, cette méthode annule l'ancien fichier ODF et le remplace par un nouveau, qui reflète tous les changements effectués à travers le ou les connecteurs de documents. Mais on n'est pas obligé de modifier le fichier source. La méthode `save()`, pour peu qu'on lui donne un nom de fichier en argument, enregistre la version modifiée dans la cible indiquée, et le fichier d'origine reste inchangé, ce qui facilite la réutilisation de documents en tant que modèles :

```
my $archive = ooFile("modele.odt");
# création des connecteurs...
# traitement du document...
$archive->save("cible.odt");
```

Les exemples qui précèdent donnent à penser que, pour commencer à travailler, il faut nécessairement partir d'un fichier ODF existant dont le nom est donné à `ooFile()`. En réalité, la fourniture d'un nom de fichier est obligatoire, mais, moyennant une option appropriée, on peut indiquer au constructeur `ooFile()` qu'il s'agit d'un nouveau fichier, à créer de toutes pièces. L'option requise dans ce cas est `create`, et elle doit être passée avec une valeur spécifiant la classe du document à créer. Exemple :

```
my $classeur = ooFile
      (
      "inventaire.ods",
      create => "spreadsheet"
      );
```

Ce nouvel exemple prépare la création d'un nouveau document de classe "tableur" (*spreadsheet*). Les autres valeurs supportées par OODoc sont *text*, *presentation* et *drawing*. On peut se poser la question de l'utilité de passer en paramètre la classe du document, alors que le suffixe du nom de fichier l'indique par ailleurs. Mais il faut savoir que l'API OpenOffice::OODoc est totalement agnostique quant aux noms des fichiers, qu'elle n'analyse même pas, à la seule exception près des fichiers de la forme "**.xml*", qui sont considérés comme des fichiers XML "plats" (non compressés), qu'elle peut aussi prendre en charge (mais dont nous ne parlons pas dans cet article). Ainsi, une construction comme celle de l'exemple ci-dessous serait parfaitement valide :

```
my $classeur = ooFile
      (
      "archive.zip",
      create => "presentation"
      );
```

Jusqu'à présent, nous en sommes restés à des exemples "canoniques" de manipulation de connecteurs. En fait, OpenOffice::OODoc est une API assez laxiste en termes de syntaxe et sait s'accommoder de raccourcis plus ou moins orthodoxes. Si on veut par exemple créer un connecteur unique sur le contenu d'un fichier ODF, on peut écrire directement :

```
my $doc = ooDocument(file => "MonFichier.odt");
```

Ici, le connecteur `$doc` (qui correspond à *content.xml*, membre par défaut en l'absence de l'option `member`) est activé directement avec un nom de fichier, passé via une option `file` et sans l'intermédiaire d'un objet archive préalablement créé par `ooFile()`. Techniquement, ce n'est qu'un raccourci d'écriture, qui produira un appel automatique de `ooFile()`.

Ensuite, il est parfaitement possible d'écrire une instruction d'enregistrement telle que :

```
$doc->save();
```

car les objets Document possèdent une méthode `save()` qui n'exécute rien par elle-même mais qui appelle automatiquement la méthode `save()` de l'objet archive auquel le connecteur est lié.

On peut même aller encore plus loin (au risque d'un codage plus difficile à interpréter). Le code ci-dessous, qui crée un connecteur `styles` et un connecteur `content` sur le même fichier, parvient à ne pas passer explicitement par `ooFile()` pour initialiser l'archive partagée :

```
my $contenu = ooDocument
                    (
                    file   => "MonFichier.odt",
                    member => "content"
                    );
my $styles   = ooDocument
                    (
                    file   => $contenu,
                    member => "styles"
                    );
```

La seconde instruction de cette séquence est peu orthodoxe car elle utilise l'option `file` non pas pour passer un nom de fichier, mais pour donner la référence d'un connecteur déjà ouvert, retourné par le précédent appel de `ooDocument()`. Elle fonctionne quand même, car le constructeur, reconnaissant que la valeur de l'option `file` est une référence d'objet *Document* au lieu d'une chaîne de caractères, comprend qu'il ne s'agit pas d'un nom de fichier et qu'il doit créer un nouveau connecteur sur la même archive que le précédent. Détail à noter : après une telle séquence d'initialisation, les instructions `$contenu->save` et `$styles->save` seront synonymes, puisque la méthode `save()` est en réalité exécutée par l'objet archive sous-jacent et non par le connecteur de document.

Cependant, l'option `file` est redoutable pour un programmeur distrait, car elle cache un piège potentiellement fatal, dans lequel tombe l'exemple suivant :

```
my $contenu = ooDocument
                    (
                    file   => "MonFichier.odt",
                    member => "content"
                    );
my $styles   = ooDocument
                    (
                    file   => "MonFichier.odt",
                    member => "styles"
                    );
# traitement...
$contenu->save;
$styles->save;
```

Ici, en effet, les connecteurs `$contenu` et `$styles` sont créés en liaison avec le même fichier, mais sans aucune connexion explicite l'un avec l'autre. Dans ces conditions, l'API ne peut pas savoir qu'il s'agit de deux accès au même fichier. Si des modifications sont faites à travers les deux connecteurs, seuls, dans cet exemple, seront conservés les changements effectués via `$styles`, car le dernier appel de `save()` écrase le fichier créé par le premier. Pour corriger cet exemple, il suffit de remplacer le nom de fichier par l'objet `$contenu` dans le deuxième appel de `ooDocument()` et (pour éliminer des entrées-sorties redondantes) de supprimer l'un des deux `save()`, au choix, car ils sont équivalents.

4 Opérations simples sur les conteneurs de texte

Ayant présenté les fonctions globales d'accès aux documents, nous pouvons aborder les opérations les plus courantes et les plus simples par lesquelles, via un connecteur de document, on peut agir sur les éléments textuels.

L'élément textuel le plus communément utilisé est le paragraphe. Attention, un paragraphe n'est pas seulement l'élément de base d'un texte littéraire dit "non structuré". Ce mot ne doit pas être pris dans le sens du langage courant. Il faut savoir que presque tout ce qui contient du texte passe par des paragraphes. Une liste à puces ou un tableau, par exemple, est composé, entre autres, de paragraphes.

Pour présenter l'opération la plus simple dans ce domaine, et respecter une vieille tradition, commençons pas écrire un programme complet, bref mais opérationnel, qui crée un nouveau document textuel et y place l'inévitable paragraphe "*Hello World !*".

```
use OpenOffice::OODoc;
my $ar = ooFile("essai.odt", create => "text");
my $doc = ooDocument(archive => $ar, member => "content");
$doc->appendParagraph(text => "Hello World !", style => "Standard");
$ar->save;
```

Chacun est invité à vérifier le bon fonctionnement de ce programme en ouvrant le fichier *essai.odt* avec son traitement de textes favori... compatible OpenDocument, cela va de soi. On constate immédiatement que le fichier ainsi créé présente, après une première ligne vide, un paragraphe contenant le texte donné via l'option `text` de la méthode `appendParagraph()`. La première ligne s'explique simplement parce que tout document de classe `text` créé par `ooDocument()` avec l'option `create` est initialisé avec un premier paragraphe vide.

S'il s'agissait d'ajouter le nouveau paragraphe à la fin d'un document existant, le même script serait utilisable, en supprimant seulement l'option `create` dans l'appel de `ooFile()`.

Notons au passage que la méthode `appendParagraph()` ne permet pas seulement de placer le nouveau paragraphe en fin de document. C'est son comportement par défaut, mais elle peut aussi, avec un paramétrage approprié, attacher le paragraphe à un conteneur de texte donné (par exemple une section ou une cellule de tableau).

Connaissant `appendParagraph()`, nous pouvons l'utiliser de manière répétitive, comme dans le script suivant qui convertit en format OpenDocument un flot de texte plat en provenance de l'entrée standard :

```
use OpenOffice::OODoc;
my $ar = ooFile("essai.odt", create => "text");
my $doc = ooDocument(archive => $ar, member => "content");
while (my $ligne = <STDIN>)
      {
      chomp $ligne;
      $doc->appendParagraph(text => $ligne, style => "Standard");
      }
$doc->save;
```

J'entends une première objection : ce convertisseur fonctionne mais l'esthétique du résultat ne sera pas au goût de tout le monde. La réponse est simple : ici nous créons des paragraphes en leur affectant le style "*Standard*", de sorte que c'est selon un style par défaut, dont les caractéristiques dépendent du logiciel de présentation, qu'ils sont affichés. Mais nous verrons plus loin comment fabriquer nos propres styles de paragraphes.

Il n'y a évidemment pas que la méthode `appendParagraph()` pour créer un nouveau paragraphe. Nous disposons aussi de `insertParagraph()`, un peu plus sophistiquée parce qu'elle permet de placer un paragraphe juste avant ou juste après un autre élément déjà présent dans le document.

Par ailleurs, il existe une méthode `getParagraph()` qui permet de sélectionner un paragraphe par son numéro d'ordre dans le document. La logique d'adressage est semblable à celle des tableaux Perl, c'est-à-dire que les numéros commencent à zéro, donc pour extraire le premier paragraphe d'un document il suffit d'écrire

```
my $p = $doc->getParagraph(0);
```

Comme les éléments d'un tableau, les paragraphes peuvent être désignés par des nombres négatifs, le numéro `-1` étant celui du dernier paragraphe.

Voici donc comment, en combinant `getParagraph()` et `insertParagraph()`, on peut insérer un nouveau paragraphe qui deviendra l'avant-dernier du document :

```
my $repere = $doc->getParagraph(-1);
$doc->insertParagraph
     (
     $repere,
     position       => "before",
     text           => "Hello, World !",
     style          => "Standard"
     );
```

On remarquera que le premier argument de `insertParagraph()` est l'élément devant (ou derrière) lequel sera placé le nouveau paragraphe. L'élément `$repere` pourrait être de n'importe quel type, et pas nécessairement un paragraphe. L'option `position`, évidemment, permet de choisir si l'insertion doit se faire avant ou après le point de repère.

L'usage de `getParagraph()` n'est cependant pas exclusivement destiné à capturer des points de repère en vue d'insérer de nouveaux éléments. Dès lors qu'un paragraphe est sélectionné, on peut traiter son contenu.

La méthode générique `getText()`, applicable à un très grand nombre de conteneurs de textes, fonctionne évidemment avec un paragraphe, donc elle extrait le texte, assurant automatiquement la conversion entre l'encodage UTF-8 d'OpenDocument et le jeu de caractères local de l'utilisateur (soit par exemple iso-8859-1 ou iso-8859-15). Ainsi, la capture du texte d'un paragraphe donné dans une variable Perl se fait très simplement :

```
$para = $doc->getParagraph($numero);
$texte = $doc->getText($para);
```

Il existe aussi un raccourci `getParagraphText()`, qui donne directement le texte à partir du numéro de paragraphe, mais qui ne permet pas de mise à jour puisqu'elle ne fournit pas la référence du paragraphe.

Si on dispose d'un paragraphe préalablement extrait par `getParagraph()` ou autrement, on peut aussi lui affecter un contenu via la méthode générique `setText()`, Exemple :

```
$doc->setText($para, "Bonjour le monde !");
```

`setText()` est applicable à d'autres éléments. Cette méthode annule tout contenu antérieur et le remplace par un texte donné. Cette méthode réalise l'encodage inverse de celui de `getText()`, à savoir la conversion en UTF-8 du texte chargé, qui doit être donné dans le jeu de caractères local.

Pour insérer du texte à une certaine position sans supprimer le texte existant, on doit utiliser une autre méthode, `extendText()`.

L'essentiel de ce que nous venons de voir à propos des paragraphes s'applique également aux titres. Un titre (*heading*, dans la spécification ODF) est une espèce particulière de paragraphe, qui possède, en plus d'un contenu et d'un style comme tout paragraphe, d'un attribut numérique qui indique son niveau dans la hiérarchie des titres. Pour créer ou extraire un titre, on utilise `appendHeading()`, `insertHeading()`, `getHeading()`. dont le mode d'emploi ressemble fortement à celui des méthodes `xxxParagraph()` déjà présentées. À deux détails près. D'abord, un paramètre `level`, correspondant au niveau hiérarchique, doit être fourni en plus. Ensuite, l'utilisation d'un paramètre `style`, préférable pour la création d'un paragraphe, est pratiquement impérative pour celle d'un titre.

L'interface offre par ailleurs des méthodes de recherche et de remplacement de texte dans tout ou partie du document, agissant aussi bien sur les paragraphes que sur les titres. On ne peut pas toutes les présenter ici. La plus générale est `selectElementsByContent()` : cette méthode est à double détente, sachant que, d'une part, elle sélectionne la liste des éléments textuels de tous types dont le contenu correspond à une expression régulière donnée, et d'autre part elle peut, pendant sa recherche, déclencher l'exécution d'un traitement donné (substitution de texte, appel de fonction externe, etc) à chaque fois qu'elle rencontre un élément conforme au filtre.

Quant à la suppression d'un élément de texte, elle se fait à l'aide de la méthode `removeElement()` qui supprime tout élément qu'on veut bien lui donner en argument. Attention, cette méthode est redoutable, car elle fait disparaître (avec tout son contenu) un objet structuré de haut niveau aussi bien qu'un simple paragraphe. Mais on peut aussi détacher un élément de quelque part pour le raccrocher ailleurs dans le document. La méthode `cutElement()` facilite de telles opérations de copier-

coller. La séquence suivante, sur le thème "les derniers seront les premiers", coupe le dernier paragraphe d'un document et le place en première position :

```
$dernier = $doc->cutElement($doc->getParagraph(-1));
$premier = $doc->getParagraph(0);
$doc->insertElement($premier, $dernier, position => "before");
```

Ce nouveau bout de code nous a permis d'introduire `insertElement()`, qui est en quelque sorte une version générique de `insertParagraph()`, et qui permet d'insérer n'importe quel élément (préalablement créé ou prélevé ailleurs) devant ou derrière n'importe quel autre. Ce qui nous amène à conclure cette section en précisant que la plupart des méthodes disponibles pour la manipulation d'éléments simples sont en réalité utilisables pour tout autre espèce d'éléments.

L'interface facilite aussi l'insertion de champs variables (dont le contenu sera renseigné par le logiciel bureautique au moment de présenter le document) dans les conteneurs de textes. Sans donner la liste complète des possibilités, on se contentera ici d'une illustration simple. Soit un paragraphe `$p`, préalablement sélectionné dans un document `$doc`, dont le texte est le suivant :

"Il est exactement HeureIci chez nous et HeureLàbas à Montréal."

On veut remplacer "HeureIci" et "HeureLàbas" par des zones variables affichant respectivement l'heure courante locale et l'heure courante avec un décalage négatif de 6 heures (ce qui donne l'heure de Montréal si le document est ouvert à Paris). Il suffit d'écrire les deux instructions suivantes :

```
$doc->setTextField
     ($p, "HeureIci", "time");
$doc->setTextField
     ($p, "HeureLàbas", "time", "time-adjust" => "-PT06H");
```

La même méthode `setTextField()` est utilisable pour tous les types de champs de textes variables prévus par le standard... à condition de connaître les mots-clés correspondants et les paramètres optionnels (comme ici `time-adjust`) qui s'y appliquent[17].

5 Manipulation des tableaux

Les tableaux sont tout particulièrement visés par les applications de traitement automatique de documents, parce qu'ils peuvent fournir ou recevoir des données dont la structure est très proche de celle des bases de données d'entreprise. Le stockage des tableaux (compte tenu surtout de leurs nombreux paramètres de présentation et de typage des données) est assez complexe dans le schéma OpenDocument, mais OpenOffice::OODoc permet de masquer une grande partie de cette complexité.

5.1 Accès global à un tableau

Une première précision s'impose ici : quand on parle de table ou de tableau en ODF, on ne fait pas de différence a priori entre classes de document. Cela signifie qu'une feuille de calcul (ou un onglet) de tableur (ODS[18]) est un objet de même type que, par exemple, un tableau incorporé dans un document de classe textuelle (ODT[19]).

Avant d'accéder au contenu (c'est-à-dire aux cellules), il est préférable de récupérer la référence du tableau. Cette sélection se fait, de préférence, en utilisant le nom logique du tableau (car tout tableau possède un nom unique, choisi par l'auteur du document ou attribué par défaut par le logiciel bureautique), à l'aide de la méthode `getTable()`. Cette méthode reçoit un premier argument, qui est le nom du tableau, et deux arguments numériques optionnels, indiquant les dimensions (hauteur et largeur) de la zone du tableau, comptée à partir de la cellule supérieure gauche, qu'on veut adresser par la suite.

```
my $table = $doc->getTable("Tableau1", 36, 72);
```

17 La référence en la matière est le chapitre 6 de la spécification, intitulé "*Text Fields*".
18 *OpenDocument Spreadsheet*
19 *OpenDocument Text*

On peut se demander ici à quoi servent les arguments dimensionnels, notamment si on veut tout simplement adresser l'ensemble du tableau. En fait, ces arguments sont optionnels et leur rôle est simplement d'initialiser un espace de travail à l'intérieur duquel les fonctions d'accès direct aux cellules à partir de leurs coordonnées cartésiennes seront opérationnelles. Compte tenu de certaines caractéristiques de représentation interne des tables dans la norme ODF, que nous ne pouvons pas développer ici, cette initialisation est parfois nécessaire, parfois non. Elle n'est jamais nécessaire, par exemple, si le tableau a été créé par un programme Perl à l'aide de la méthode `appendTable()` ou `insertTable()` et si le document n'a pas été entretemps édité et ré-enregistré par un autre logiciel. On constate aussi qu'elle est inutile dans le cas d'un tableau appartenant à un document de classe text (ODT) produit par OpenOffice.org Writer ou KWord, du moins dans leurs versions actuelles. Elle est en revanche absolument indispensable si le tableau visé est une feuille de tableur provenant d'OpenOffice.org Calc ou de KSpread.

Attention, `getTable()`, si on lui fournit un argument entièrement composé de chiffres décimaux, recherche la table non pas par son nom mais par sa position dans la liste des tables du document, selon la même logique que `getParagraph()`. La bonne nouvelle, c'est qu'on peut accéder aux tables sans connaître leurs noms. La mauvaise, c'est que `getTable()` ne permet pas de retrouver une table dont le nom se compose uniquement de chiffres (sauf si, par une heureuse coïncidence, le nom correspond à la position).

Nous venons d'introduire incidemment les deux méthodes de création de tables. La plus simple, `appendTable()`, attache un nouveau tableau au document dans les mêmes conditions que `appendParagraph()` attache un nouveau paragraphe. L'application doit cependant, au minimum, fournir un nom (unique), un nombre de lignes et un nombre de colonnes (et peut ajouter des paramètres de style). La valeur retournée par cette méthode est une référence de tableau utilisable dans les mêmes conditions que si elle avait été récupérée par `getTable()`. L'instruction suivante crée un tableau de 10x15 à la fin d'un document :

```
my $table = $doc->appendTable("Feuille de Paie", 10, 15);
```

La méthode `insertTable()` permet, avec d'autres paramètres, d'insérer un tableau n'importe où dans le document, devant ou derrière un autre élément donné comme point de repère.

Notons bien que ces méthodes ne sont pas réservées aux feuilles de tableurs, mais s'appliquent aussi bien à toute espèce de tableau contenu dans un document de classe quelconque.

D'autre part, les méthodes générales de suppression et de copie d'éléments s'appliquent aux éléments complexes comme les tableaux aussi bien qu'aux simples paragraphes.

5.2 Accès au contenu d'un tableau

La méthode `getTable()` prépare l'accès à une table et retourne une référence utilisable à cet effet.

Mais le but de la manoeuvre est évidemment d'accéder au contenu des cellules. Cet accès, dès lors que nous avons la référence de la table, se fait très simplement par `getCell()` :

```
my $table = $doc->getTable("Feuille1", 50, 26);
my $cellule = $doc->getCell($table, 6, 15);
```

La méthode `getCell()` est capable de s'adapter à plusieurs modes d'adressage de cellules. Dans la forme utilisée ci-dessus, on donne, après la référence de la table, le numéro de ligne puis le numéro de colonne. Attention, tout comme les numéros de paragraphes, ces numéros commencent à zéro et peuvent aussi être comptés en valeurs négatives à partir de la fin (à condition que la notion de fin de table ait un sens, ce qui n'est pas forcément le cas avec une feuille de tableur).

Que la cible soit une feuille de tableur ou tout autre tableau, `getCell()` supporte aussi la syntaxe d'adressage de style "bataille navale" utilisée dans les formules de calcul bureautiques ; il faut alors se souvenir que la colonne doit être spécifiée en premier sous forme de lettre(s), et que les numéros ne commencent pas à zéro. Les deux instructions suivantes sont donc équivalentes :

```
my $cellule = $doc->getCell($table, "C7");
my $cellule = $doc->getCell($table, 6, 2);
```

Lorsqu'on veut traiter plusieurs cellules dans une même ligne, on peut optimiser l'accès en sélectionnant une fois pour toutes la ligne (*row*) et en utilisant sa référence (au lieu de celle de la table) pour accéder à chaque cellule. Il faut savoir, à ce sujet, que les tableaux stockés en ODF sont en réalité des structures arborescentes et non plus des tables, et que les cellules sont contenues dans les lignes et non dans les colonnes (qui ne portent que des indications de présentation). L'exemple suivant, qui se contente d'exporter à plat sur la sortie standard le contenu de chacune des cellules d'une ligne donnée, montre comment sélectionner, disons, la 6ème ligne d'un tableau avec `getRow()` et l'utiliser dans `getCell()`.

```
my $table = $doc->getTable("Feuille1", 15, 12);
my $ligne = $doc->getRow($table, 5);
for (my $i = 0 ; $i < 12 ; $i++)
     {
     my $cellule = $doc->getCell($ligne, $i);
     print $doc->getText($cellule) . "\n";
     }
```

On voit que `getCell()` est capable de reconnaître le type de son premier argument et, s'il s'agit d'une ligne et on d'une table, de sélectionner dans cette ligne la cellule dont la position est indiquée par l'argument suivant. Mais, dès lors qu'il s'agit de traiter successivement toutes les cellules d'une ligne, on peut encore optimiser avec `getRowCells()`, qui extrait d'un coup la liste des cellules d'une table (dont on peut alors se dispenser de connaître la taille :

```
my $table = $doc->getTable("Feuille1", 15, 12);
my @cellules = $doc->getRowCells($table, 5);
foreach my $cellule (@cellules)
     {
     print $doc->getText($cellule) . "\n";
     }
```

Et si on voulait appliquer le même traitement à toutes les lignes du tableau, on pourrait extraire celles-ci par `getTableRows()`...

Mais nous ne pouvons pas décrire ici toutes les méthodes d'accès aux tables. Il est temps de dire quelques mots sur le traitement d'une cellule.

Dans notre dernier exemple, on a utilisé la méthode `getText()`, déjà rencontrée à propos des paragraphes, pour exporter le contenu éditable d'une cellule. On s'en contentera souvent pour exploiter un contenu documentaire existant.

Nous avons vu comment lire le contenu d'une cellule. Mais comment peut-on, en sens inverse, placer une valeur dans une cellule ? S'il s'agit d'une cellule de texte (c'est-à-dire sans type numérique), rien n'est plus simple : on peut se servir de la méthode `setText()` applicable à tous les conteneurs de textes, comme dans l'exemple suivant, qui crée un tableau et y charge le contenu d'un fichier de données en format CSV simplifié. Pour ne pas trop compliquer l'exercice, on suppose que la virgule est le séparateur de champ, que nous n'avons pas à gérer de guillemets, que chaque ligne du fichier source correspond à une ligne du tableau cible, et que nous connaissons à l'avance les dimensions à donner au tableau pour qu'il puisse digérer le contenu du fichier source[20].

20 Cet exemple est inspiré de l'utilitaire *text2table* qui est fourni avec la distribution OpenOffice::OODoc.

```
open SOURCE, "<", "data.txt";
my $table = $doc->appendTable("Rapport", $hauteur, $largeur);
my $entree = undef;
my @lignes = $doc->getTableRows($table);
for (my $i = 0 ; $i < $hauteur && ($entree = <SOURCE>) ; $i++)
      {
      chomp $entree;
      my $ligne = $lignes[$i];
      my @valeurs = split ',', $entree;
      my @cellules = $doc->getRowCells($ligne);
      for (my $j = 0 ; $j < $largeur && @valeurs ; $j++)
            {
            $doc->setText($cellules[$j], shift @valeurs);
            }
      }
close SOURCE;
```

Mais si la cellule visée est de type numérique, il vaut mieux s'appuyer sur les méthodes de lecture-écriture spécialisées telles que `updateCell()` ou `cellValue()`.

Il faut en effet savoir que, pour obtenir des résultats corrects avec une cellule numérique, on doit mettre à jour simultanément deux ou plusieurs paramètres, et pas seulement son contenu visible. Une telle cellule contient deux valeurs, soit une valeur calculable et une valeur visible. Par exemple pour une cellule de type monétaire dont la valeur visible est "123,45 €", la valeur calculable est "123.45". La valeur calculable est indépendante de la présentation, tandis que la valeur visible tient compte du séparateur décimal en vigueur, du style de la cellule et d'autres paramètres tels que la devise et le type de données.

Prenons comme exemple un cas volontairement compliqué, mais réaliste : la mise en place d'un montant monétaire en euros dans une cellule qui n'est pas encore typée. Il faut donc typer cette cellule, et pas seulement lui donner un contenu. En l'occurrence, il faut lui donner le type "*currency*", la devise "*EUR*", et un format de présentation. Ce dernier dépend d'un style, et nous n'avons pas ici la place de présenter la création d'un style de cellule monétaire ; nous allons donc un peu simplifier le problème en supposant que, dans le même tableau, il existe déjà une cellule monétaire dont la présentation nous convient et dont nous allons tout simplement emprunter le style. Notre tableau s'appelle "Comptes", la cellule cible est "D9" et nous voulons y inscrire un montant de 456,78 euros selon le même format qu'une cellule monétaire existante "C4".

```
# accès à la table
my $table = $doc->getTable("Comptes", 10, 5);
# récupération du style de la cellule C4
my $style = $doc->cellStyle($table, "C4");
# accès à la cellule D9
my $cible = $doc->getCell($table, "D9");
# affectation type, style et format
$doc->cellValueType($cible, "currency");
$doc->cellCurrency($cible, "EUR");
$doc->cellStyle($cible, $style);
# inscription valeurs éditable et visible
$doc->cellValue($cible, "456.78", "456,78 €");
```

Cet exemple auto-documenté présente les différentes méthodes de paramétrage d'une cellule. Ces méthodes ont une syntaxe assez élastique, puisqu'elles acceptent plusieurs modes de désignation de la cellule concernée. Ici, le premier appel de `cellStyle()` adresse la cellule par ses coordonnées absolues (tableau et position dans le tableau), tandis que le second agit sur une cellule désignée par sa référence préalablement extraite par `getCell()`. On pourrait aussi, dans toutes les méthodes `cellXxx()`, identifier une cellule par une référence de ligne récupérée par `getRow()` suivie d'une position dans la ligne. Enfin, on peut noter les deux valeurs inscrites dans la cellule par la dernière instruction `cellValue()` ; la seconde est la valeur visible.

La méthode `cellFormula()` permet de consulter et de modifier une formule contenue dans une cellule (ou simplement de détecter la présence d'une formule). L'exemple suivant applique à la cellule "H8" de notre table "Comptes" une formule qui affiche la somme des cellules de "C4" à "D9" :

```
$doc->cellFormula("Comptes", "H8", "oooc:=SUM([.C4:.D9])");
```

Mais attention, OpenOffice::OODoc n'interprète pas le contenu des formules, qui n'est d'ailleurs pas couvert par le standard OpenDocument[21], de sorte que la validité d'une formule insérée via `cellFormula()` est de la responsabilité du programmeur et ne s'apprécie que par rapport au logiciel à travers lequel le document sera visualisé. De plus, les formules doivent être injectées selon une syntaxe interne qui ne correspond pas à ce que voit l'utilisateur final dans son tableur ; ainsi, la formule ci-dessus s'écrirait et s'afficherait "`=SOMME(C4:D9)`" sous OpenOffice.org (version française). Cela dit, certaines formules sont relativement portables, comme celle de l'exemple ci-dessus qui fonctionne aussi bien avec KSpread qu'avec OpenOffice.org Calc.

Avec OpenOffice::OODoc, on peut agrandir l'espace d'une cellule (sans modifier par ailleurs la structure du tableau) en lui permettant de recouvrir plusieurs colonnes et/ou plusieurs lignes. Ainsi, l'instruction suivante fait s'étendre la cellule B4 d'un tableau donné sur 3 colonnes, de sorte qu'elle recouvre C4 et D4 :

```
$doc->cellSpan($table, "B4", 3);
```

Nous avons vu comment consulter et modifier des cellules existantes. Mais peut-on aussi créer des cellules qui n'existent pas, c'est-à-dire pratiquement étendre la surface d'un tableau ? La réponse est oui, car l'interface comporte notamment des méthodes `insertRow()` et `insertColumn()` qui, comme leurs noms l'indiquent, permettent d'étendre un tableau verticalement et horizontalement. À utiliser avec modération (car elles sont coûteuses en temps de calcul) elles permettent d'ajuster les tableaux en fonction des besoins, sachant que les méthodes `deleteRow()` et `deleteColumn()`, qui font le travail inverse, sont également disponibles.

En revanche, l'apparente facilité d'utilisation des méthodes `appendTable()` et `insertTable()` ne doit pas être considérée comme un encouragement à créer systématiquement des tables à partir de rien. En effet, produire une table n'est rien, mais construire sa présentation entièrement par programme est un travail de fourmi, compte tenu de la quantité de styles à définir. Il est nettement plus indiqué de se servir de modèles de tableaux fabriqués avec un bon outil bureautique interactif. L'interface contient en effet tout ce qu'il faut pour favoriser la réutilisation d'éléments, y compris en-dehors des documents dans lesquels ils ont été créés.

6 Illustrer les documents

J'ai dit quelques mots dans l'introduction de cet article au sujet des modalités de stockage des images dans les fichiers ODF. Nous allons voir comment on peut concrètement agir sur ces images à travers notre interface de programmation.

Une image est pour nous un élément parmi d'autres, avec cependant une caractéristique particulière qui le distingue d'un conteneur de texte : il porte une référence à un objet binaire, parfois de grande taille, et stocké ailleurs. D'autre part, une image est elle-même attachée à un conteneur (appelé *frame* dans le vocabulaire ODF) dont le rôle est de servir de support à toutes sortes d'objets rectangulaires incrustés dans les documents, dont les images mais aussi les "boîtes de textes" (*text boxes*) très utilisées dans les supports de présentation mais dont nous ne parlons pas dans cet article. Et il ne faut pas oublier qu'une image, comme tout autre élément visible, doit avoir un style. Cependant, le maniement de tout cet attirail est relativement simple avec OODoc. Pour s'en convaincre, voyons comment on peut attacher, disons à la page 2 d'un document de classe texte (ODT), une image importée du système de fichiers local.

```
$doc->createImageStyle("quelconque");
$doc->createImageElement
     (
     "Ailleurs",
     description    => "Quelquepart la nuit",
     import         => "/usr/share/wallpapers/alien-night.jpg",
     page           => 2,
     position       => "3cm, 4.5cm",
     size           => "8cm, 6cm",
     style          => "quelconque"
     );
```

La première instruction a créé, en tant que style automatique (nous n'avons pas besoin qu'il soit visible pour l'utilisateur final), un style d'image par défaut. Il est en effet préférable (même si ce n'est pas strictement obligatoire) que chaque image

21 Affirmation vraie au moment où cet article est rédigé. Mais l'OASIS a institué un comité technique "*OpenFormula*" chargé d'étendre la spécification dans ce domaine.

ait un style. La méthode `createImageStyle()` produit, par défaut, un style d'image qui convient dans la plupart des cas (mais elle est bien sûr paramétrable). La seconde instruction est plus intéressante. Son premier argument est le nom (identifiant, donc unique pour le document) qui sera donné à l'image. Vient ensuite un paramètre `description`, optionnel, qui permet d'associer un commentaire à l'image. Le paramètre `import` indique le chemin d'accès au fichier qui est censé contenir une image dans un format supporté par le logiciel bureautique qui affichera le document ; attention, ce fichier n'est pas chargé immédiatement, il ne l'est que quand les changements sont physiquement enregistrés par une instruction `save()`. L'option `page` va forcer l'attachement de l'image à la page 2 (disons tout de suite que, si le document n'a qu'une page, l'image ne sera pas affichée). Le paramètre `position` (obligatoire si on ne veut pas que l'image soit collée dans un coin) donne l'abscisse et l'ordonnée de l'image par rapport à la page. Quant à l'option `size`, il ne faut pas croire qu'elle est facultative et que, en son absence, l'image sera affichée à sa "taille d'origine". Même si la notion de "taille d'origine" existe dans certains logiciels bureautiques, elle n'est pas inscrite dans le fichier ODF et c'est au programme de la renseigner[22]. Enfin, le paramètre style nous permet d'appliquer le style d'image précédemment défini.

Les paramètres géométriques (position et taille) méritent une petite précision. Ce sont des chaînes de caractères et non pas des valeurs numériques. Ils contiennent à la fois les valeurs et les unités de mesures. Ici, on a utilisé des centimètres (`cm`), mais d'autres unités sont supportées ; il peut s'agir de millimètres (`mm`) ou d'unités anglo-saxonnes non métriques ; on peut aussi utiliser le point (`pt`). Par ailleurs, il n'est jamais trop tard pour ajuster ces paramètres, car d'autres méthodes permettent de déplacer ou de retailler facilement une image en place. Comme dans cette séquence qui déplace de 2cm à droite l'image que nous venons d'insérer et divise sa taille par 2 :

```
$doc->imagePosition("Ailleurs", "5cm, 4.5cm");
$doc->imageSize("Ailleurs", "4cm, 3cm");
```

À partir de cet exemple, on peut développer de nombreuses variantes.

D'abord, le chemin d'accès au fichier peut être introduit par une option `link` au lieu d'une option `import`. Dans ce cas, le résultat apparent sera le même, mais aucun fichier ne sera importé. Il faudra, bien sûr, que l'image liée soit accessible par le logiciel qui affichera le document. En pratique, cette option `link` pourra être utilisée pour lier des images distantes (*http://...*) aussi bien que des images locales. Dès lors qu'il n'y a pas d'importation physique, `createImageElement()` accepte sans contrôle n'importe quel URL, à charge pour le programmeur de spécifier des cibles qui seront effectivement joignables en temps voulu par le logiciel de présentation.

Si, au lieu d'attacher l'image à une position fixe dans une page donnée, on veut la lier à la position d'un élément quelconque, il suffit de remplacer l'option `page` par une option `attachment` qui, elle, pointe sur la référence de l'élément d'ancrage. Il peut s'agit d'un paragraphe, d'une cellule, ou de tout autre chose. Attention, il faut dans ce cas repenser les coordonnées (paramètre `position`), car elles leur origine est alors celle du point d'attachement et non plus un coin de page. Ce paramètre peut être omis si on veut simplement placer l'image à la position du point d'attachement.

Une image déjà présente dans un document se recherche (de préférence) à peu près dans les mêmes conditions qu'un tableau, à l'aide d'une méthode `getImageElement()`. L'élément retourné par cette méthode peut faire l'objet des opérations de base communes à tous les éléments (copie, suppression, etc).

Lorsque (comme c'est le cas avec OpenOffice.org) les images importées sont stockées en tant que membres indépendants dans l'archive ODF (et non pas imbriquées dans le contenu), il est très facile de les exporter. Ainsi, la méthode `exportImage()` permet directement d'extraire une image désignée par son nom logique dans le document à destination d'un fichier externe.

```
$doc->exportImage("Ailleurs", "/home/jmg/images/alien-night.jpg");
```

Et, pour exporter d'un coup toutes les images d'un document, on dispose d'une méthode `exportImages()`.

Quant aux caractéristiques de présentation des images (autres que la taille et la position) elles sont contrôlées par des styles.

22 On peut utiliser le module Perl *Image::Size* pour rechercher la taille d'origine d'une image, mais OpenOffice::OODoc ne le fait pas automatiquement pour nous.

7 Opérations simples sur les styles

Nous avons évoqué au début de cet article la distinction entre styles automatiques et styles nommés. Cette distinction est déterminante pour savoir où un style est stocké (*content.xml* ou *styles.xml*) mais, à ce détail près, elle n'est pas fondamentale. Il existe d'autres distinctions qui, elles, déterminent la structure de données (c'est-à-dire la structure XML) qui décrit chaque style. C'est un sujet vaste et foisonnant, aussi on se limitera dans cet article à la présentation générale des styles de textes et des styles d'images. On laissera volontairement de côté d'autres styles tout aussi importants, mais qui nous entraîneraient dans de trop longs développements, comme par exemple les styles de pages.

La première difficulté, pour le programmeur, vient du grand nombre d'attributs de présentation pouvant intervenir dans un document bureautique simple. Seule la spécification ODF de l'OASIS peut prétendre en donner la liste complète[23]. Fort heureusement, une application "ordinaire" n'en utilise qu'une petite partie.

7.1 Styles de paragraphes

Notre premier exemple va consister à créer un style de texte nommé, disons le style "EnCouleurs", qui sera défini comme un style de paragraphe, et qui aura pour effet de colorer le texte en bleu sur fond jaune, et de forcer à 14 points la taille des caractères. Ce style utilisera des attributs du domaine *FO*[24], dont le nom commence par "`fo:`". En tant que style nommé (donc défini dans *styles.xml*), il devrait être ensuite visible sous OpenOffice.org ou autre traitement de textes ODF dans la liste des styles disponibles pour l'utilisateur final. Mais dans notre exemple, nous allons l'affecter immédiatement, par programme, à un nouveau paragraphe créé dans le contenu du document (*content.xml*), ce qui va nous amener à créer deux connecteurs sur le même fichier.

```
my $ar = ooFile("essai.odt");
my $styles = ooDocument(archive => $ar, member => "styles");
my $contenu = ooDocument(archive => $ar, member => "content");
$style->createStyle
      (
      "EnCouleurs",
      parent => "Standard",
      family => "paragraph",
      properties     =>
             {
             -area                  => "text",
             "fo:background-color"  => odfColor("yellow"),
             "fo:color"             => odfColor("blue")
             }
      );
my $para = $contenu->appendParagraph
             (
             text    => "Le texte du nouveau paragraphe",
             style   => "EnCouleurs"
             );
$ar->save;
```

La méthode `createStyle()` exige toujours d'abord un nom (unique pour une famille de styles). On peut aussi lui passer une option `parent` qui permet d'hériter les propriétés d'un style existant (ici on s'appuie sur le style *Standard*. Tous les styles nommés ont le privilège de pouvoir servir de base à la définition d'autres styles, de sorte que le style créé ici pourrait ensuite être lui-même mentionné via l'option `parent` dans la définition d'une autre style. La famille (`family`) est un paramètre important qui détermine à quels types d'objets un style est applicable ; un style de la famille `paragraph` comme celui-ci est utilisable notamment pour les paragraphes et pour les titres. Viennent ensuite les propriétés au sens strict ; on y accède par le hachage `properties`.

Dans un style de paragraphe, il existe deux zones de propriétés (appelées *areas*), à savoir les propriétés "`paragraph`" portant sur l'enveloppe externe du paragraphe (retraits et espacements, marges, alignement, etc) et les propriétés "`text`" qui s'appliquent aux caractères contenus. L'option "`-area`" qui figure ici dans la liste n'est pas vraiment une *property* ; c'est un drapeau qui signifie que les properties s'appliquent à la zone "`text`". La méthode `createStyle()` ne permet pas d'accéder aux deux zones en une seule instruction. Mais ce n'est jamais bloquant : s'il fallait définir des propriétés "`text`" *et* des propriétés "`paragraph`", il suffirait d'utiliser `updateStyle()` ou `styleProperties()` après `createStyle()` pour terminer le travail. Par

23 Le manuel de référence du module OpenOffice::OODoc, en particulier, ne décrit pas ces attributs, sauf quelques uns à travers des exemples.
24 *Form Objects*, l'un des dialectes de présentation standardisés par ailleurs et intégré dans le vocabulaire OpenDocument.

exemple, si nous voulons que le paragraphe soit centré et (compliquons un peu) muni d'une bordure noire continue de 0,5mm, nous pourrions revenir sur lui comme ceci :

```
$styles->styleProperties
      (
      "EnCouleurs",
      -area                 => "paragraph",
      "fo:text-align"       => "center",
      "fo:border"           => "0.5mm solid #000000"
      );
```

On voit que la syntaxe des attributs FO est assez variée et pas nécessairement intuitive. Le dernier attribut utilisé ci-dessus, "`fo:border`", contient en une seule chaîne trois paramètres, soit l'épaisseur de la bordure, le style de trait (`solid`, c'est-à-dire continu) et le code hexadécimal RVB de la couleur (facile à retenir ici, puisque, pour le noir, tout est à zéro) précédé du caractère "#". À propos de couleurs, dans le précédent exemple, nous avons utilisé une fonction `odfColor()` avec des noms de couleurs significatifs. En fait, les paramètres de couleurs, dans le format ODF, doivent être fournis au format hexadécimal "#RRVVBB". `odfColor()` est une fonction utilitaire fournie dans OODoc et qui convertit des noms symboliques de couleurs dans le format attendu en utilisant une table de correspondance RVB au choix de l'utilisateur (déclarée une fois pour toute lors de l'installation du module ou sélectionnée au gré des applications). Cela peut être le fichier *rgb.txt* généralement présent dans les environnements *X-Window* ou apparentés (*XFree*, *Xorg*)[25].

7.2 Styles de textes

Un style de texte se distingue d'un style de paragraphe en ce sens qu'il peut être appliqué à une partie du contenu d'un paragraphe.

Supposons un paragraphe dont le contenu est "J'écris cet article avec un logiciel compatible OpenDocument", et dont l'aspect est conforme à l'image ci-dessous.

Proposons-nous de mettre d'une part les mots "article" et "logiciel" en italiques grasses et d'autre part le mot "OpenDocument" en bleu sur fond jaune, sans rien changer au style du paragraphe. Nous allons d'abord, pour cela, définir deux styles que nous appellerons "ItaliqueGras" et "BleuSurJaune" :

```
$styles->createStyle
      (
      "ItaliqueGras",
      family => "text",
      properties     =>
             {
             "fo:font-style"        => "italic",
             "fo:font-weight"       => "bold"
             }
      );
$styles->createStyle
      (
      "BleuSurJaune",
      family => "text",
      properties     =>
             {
             "fo:color"             => odfColor("blue"),
             "fo:background-color"  => odfColor("yellow")
             }
      );
```

Ici, nous avons un peu abrégé (l'indicateur `-area` a été omis, parce que l'area par défaut dans les styles de texte est la zone "`text`", et c'est la seule qui nous intéresse ; et nous n'invoquons pas de `parent` parce que le `parent` par défaut est "*Standard*" et cela nous convient).

Reste à savoir comment affecter ces styles, sachant que nous ne pouvons pas les attribuer au paragraphe. Revenons vers le contenu du document et supposons que `$para` est la référence de notre paragraphe. Avec la méthode `setSpan()`, nous pouvons sélectionner une sous-chaîne dans un conteneur de texte et lui attribuer un style. Cette méthode demande 3

25 Dans une distribution Linux, ce fichier est généralement sous /usr/lib/X11, /usr/share/X11 ou /usr/X11R6/lib/X11.

arguments : le conteneur (ici, le paragraphe), la chaîne à "styler" (les caractères jokers et autres expressions régulières sont admis), et enfin le nom du style de texte à appliquer.

```
$contenu->setSpan($para, "article|logiciel", "ItaliqueGras");
$contenu->setSpan($para, "OpenDocument", "BleuSurJaune");
```

L'image ci-dessous donne une idée de ce que produirait cet exemple de code.

Détail important : `setSpan()` est une méthode répétitive, ce qui veut dire que, en un seul appel, elle applique le style donné à *toutes* les occurrences éventuelles de la chaîne recherchée.

7.3 Styles d'images

On a déjà entrevu `createImageStyle()` dans le cadre de la création d'une image. Cette méthode est essentiellement destinée à abréger les efforts d'un développeur qui n'a besoin que d'un style passe-partout, acceptable dans la plupart des cas. Pour contrôler plus finement la présentation d'une image, on peut utiliser la méthode générale `createStyle()`, assortie bien entendu de paramètres très différents de ceux qui conviennent à un style de texte.

Un style d'image est normalement défini dans la famille "`graphic`" et son style parent est "`Graphics`". Les attributs possibles (`properties`) sont assez nombreux et peuvent se répartir en plusieurs catégories.

Il y a d'abord les attributs de position, qui indiquent comment doivent être appliquées les coordonnées des images portant ce style. Par exemple si l'attribut "`style:vertical-pos`" est égal à "`from-top`", alors la position verticale de l'image est comptée de haut en bas.

On peut aussi utiliser des attributs applicables au cadre de présentation de l'image. Certains sont des attributs FO : ainsi, on retrouve un "`fo:background-color`" qui contrôle la couleur de fond à utiliser si l'image est transparente ou si elle n'occupe pas toute la surface du cadre, ou un "`fo:border`" qui permet de choisir le style, l'épaisseur et la couleur de la bordure, tandis que "`fo:clip`" permet de découper une zone de l'image. Un autre, "`style:shadow`", détermine un effet d'ombre autour du cadre.

Enfin, on dispose de tout un attirail de propriétés de retouche photographique qui, sans modifier le contenu graphique stocké dans l'archive ODF, appliquent des filtres de présentation. Ces filtres agissent sur la luminosité (`draw:luminance`), la correction gamma (`draw:gamma`), le contraste (`draw:contrast`), l'opacité (`draw:image-opacity`), les couleurs (`draw:color-inversion`, `draw:red`, `draw:green`, `draw:blue`).

Ces exemples sont loin d'être exhaustifs, et il faut se reporter à la spécification ODF pour une présentation complète. D'autre part, rien ne vaut l'examen direct du XML généré par un logiciel compatible pour découvrir, dans des documents concrets, comment ces attributs sont utilisés. Attention, les styles d'images sont souvent des styles automatiques à rechercher dans *content.xml*, et non dans *styles.xml*, sachant que l'utilisateur bureautique qui modifie interactivement la présentation d'une image passe rarement par la création d'un style d'image réutilisable.

Nous terminerons cette section par un exemple de création effective d'un style d'image présentant les caractéristiques suivantes :

- les coordonnées seront relatives à la page et comptées de haut en bas et de gauche à droite ;
- l'image sera présentée avec 20% de transparence sur un arrière-plan bleu (RVB `#0000ff`) ;
- elle sera encadrée par une bordure invisible (de même couleur que l'arrière-plan), continue, d'un dixième de millimètre d'épaisseur, dont elle sera distante de 0,25cm, et portera une ombre grise (RVB `#808080`) de 2mm ;
- on retouchera les couleurs à raison de moins 5% sur le rouge et plus 2% sur le vert, on ajoutera 5% de contraste et autant de luminosité, avec une correction gamma de 1,1.

```
$doc->createStyle
            (
            "Retouches",
            family         => "graphic",
            parent         => "Graphics",
            properties     =>
                   {
                   'style:vertical-pos'   => 'from-top',
                   'style:horizontal-pos' => 'from-left',
                   'style:vertical-rel'   => 'page',
                   'style:horizontal-rel' => 'page',
                   'fo:background-color'  => '#0000ff',
                   'fo:padding'           => '2.5mm',
                   'fo:border'            => '0.1mm solid #0000ff',
                   'style:shadow'         => '#808080 2mm 2mm',
                   'draw:luminance'       => '5%',
                   'draw:contrast'        => '5%',
                   'draw:gamma'           => '1.1',
                   'draw:image-opacity'   => '80%',
                   'draw:red'             => '-5%',
                   'draw:green'           => '2%',
                   'draw:blue'            => '0%'
                   }
            );
```

En reprenant cet exemple comme un modèle, et en jouant sur les valeurs affectées aux différentes propriétés, on peut déjà couvrir la plus grande partie des besoins.

On peut donc utiliser OpenOffice::OODoc pour régler assez finement l'illustration de documents tels que des catalogues commerciaux, des trombinoscopes, des manuels techniques ou des albums de famille. Cela dit, restons raisonnable : OpenDocument n'est pas fait pour jouer dans la même cour que *The Gimp* et *ImageMagick*...

8 Intégration de documents

Dans cette dernière section, nous allons indiquer quelques possibilités de combinaison de documents, en commençant pas la mise en oeuvre de la notion de "document-maître" résultant de l'intégration de plusieurs documents, pour évoquer ensuite l'usage d'un document comme d'une base de données de styles ou d'éléments de contenu destinés à être réutilisés.

8.1 Sections et documents liés

Une section est un conteneur de haut niveau auquel il est possible d'attacher toute espèce de contenu ; tout contenu qui peut être directement rattaché à la racine du document peut aussi bien être rattaché à une section.

La méthode `getSection()` permet de sélectionner une section d'après son nom (qui est unique pour le document). L'objet qu'elle retourne peut ensuite être utilisé soit pour rechercher quelque chose, soit pour ajouter quelque chose dans la section.

```
$section = $doc->getSection("Annexes");
$doc->currentContext($section);
$para = $doc->getParagraph(2);
```

Le code ci-dessus récupère la référence d'une section "Annexes" et, via une méthode `currentContext()`, réduit le contexte de recherche dans le document à cette section, de sorte que l'appel de `getParagraph()` qui vient ensuite va rechercher le paragraphe 2 (c'est-à-dire le 3ème paragraphe) dans la section "Annexes". Cette méthode `currentContext()`, que nous n'avions pas encore présentée, peut en fait être appliquée à toutes sortes de conteneurs (tableaux, boîtes de textes, listes, pages de présentation, etc). De plus, dans le cas de `getParagraph()`, on aurait pu écrire :

```
$para = $doc->getParagraph(2, $section);
```

car, comme beaucoup d'autres, cette méthode accepte un contexte de recherche en argument optionnel. Mais `currentContext()` permet de fixer un contexte jusqu'à nouvel ordre. Attention, il faut penser à exécuter un `resetCurrentContext()` pour rétablir le contexte par défaut (qui correspond à l'ensemble du document).

Pour créer un élément dans une section, on utilise les mêmes méthodes que pour le créer dans le corps principal du document. Avec les méthodes `insertXxx()`, comme `insertParagraph()`, `insertHeading()` ou `insertTable()`, on n'a même pas à le savoir, puisque chacune d'elles crée le nouvel élément immédiatement avant ou après un élément de référence donné, donc le cas échéant dans la même section. Avec les méthodes `appendXxx()`, il faut en revanche préciser une option `attachment` pointant sur la section cible :

```
$doc->appendParagraph
     (
     text           => "Suite de la section",
     style          => "Standard",
     attachment     => $section
     );
```

Notons bien que l'attachement peut porter sur autre chose qu'une section. On peut par exemple attacher un nouveau paragraphe (ou autre chose) à une cellule de tableau.

Les méthodes `appendSection()` et `insertSection()` créent et installent de nouvelles sections. Pour créer une section, il faut au minimum lui donner un nom unique, et optionnellement lui attribuer un style (mais nous n'abordons pas ici la question des styles de sections).

On peut faire migrer vers une section une liste d'éléments préalablement sélectionnés. L'exemple suivant (qui n'est qu'un cas d'école) extrait pêle-mêle tous les conteneurs dont le texte contient le mot "document" et les transfère vers une section créée à cet effet :

```
@liste = $doc->selectElementsByContent("document");
$section = $doc->appendSection("Fourre-tout");
$doc->moveElementsToSection($section, @liste);
```

Dans un texte déjà structuré par des titres, la méthode `getChapterContent()`, capable d'extraire tous les éléments subordonnés à un titre de chapitre donné, est assez commode quand on veut transformer un chapitre en section. Dans l'exemple qui suit, on sélectionne le 3ème titre de niveau 1, on extrait son texte, on utilise ledit texte comme nom d'une nouvelle section qu'on insère juste avant la position du titre, puis on sélectionne la liste des éléments qui dépendent de ce titre, et enfin on transfère cette liste d'éléments, précédée de l'élément titre lui-même, à l'intérieur de la section.

```
$element_titre = $doc->getHeading(2, level => 1);
$texte_titre = $doc->getText($element_titre);
$section = $doc->insertSection
     ($element_titre, $texte_titre, position => "before");
@liste = $doc->getChapterContent($element_titre);
$doc->moveElementsToSection($section, $element_titre, @liste);
```

Si on souhaite interdire à l'utilisateur final de modifier le contenu d'une section, il suffit d'appeler la méthode `lockSection()`, optionnellement assortie d'une clé de protection s'il est nécessaire, en plus, d'interdire le déverrouillage :

```
$doc->lockSection("Nom de section", "gzwbxzn");
```

D'un autre côté, la méthode `unlockSection()` déverrouille d'autorité n'importe quelle section, qu'elle soit protégée ou non par un mot de passe, et quelle que soit l'application par laquelle elle a été verrouillée :

```
$doc->unlockSection("Nom de section");
```

Il existe d'ailleurs une méthode `unlockSections()`, sans argument, qui déverrouille toutes les sections protégées. Ces méthodes sont utiles pour débloquer des sections protégées dont on a perdu les mots de passe[26].

Les sections ne sont pas seulement destinées à modulariser un document monolithique. Elles servent également de support à l'intégration de documents multiples sous le contrôle d'un document maître. Utilisée dans ce dernier mode, une section ne possède aucun contenu direct, mais sert simplement de support à une référence qui pointe sur un document externe. Comme les images liées qui ont été évoquées dans cet article, les documents liés doivent être désignés par des chemins d'accès ou des

26 Précisons, pour éviter tout malentendu, qu'OpenOffice::OODoc ne fournit ni n'utilise aucune technique cryptographique. Le verrouillage des sections est de même nature que celui des cellules dans une feuille de calcul. Il s'agit d'une d'aide à la bonne gestion des documents, facile à neutraliser en agissant directement sur le fichier, et non pas d'un mécanisme de sécurité. En revanche, la protection globale d'un document par mot de passe sous OpenOffice.org provoque un véritable cryptage du document et, dans ce cas, OpenOffice::OODoc ne peut accéder ni au contenu ni même aux styles.

URL correspondant à des protocoles qui seront supportés par le logiciel d'édition. On peut noter que les documents liés ne sont pas obligatoirement eux-mêmes au format OpenDocument ; en pratique ils peuvent être dans tout format supporté par le logiciel d'édition. À titre d'exemple, la séquence suivante crée un document de classe *text*, y place un premier paragraphe puis, par l'intermédiaire de deux sections, lui attache deux documents, dont un document ODT local et un document *Microsoft Word* distant, et termine par un dernier paragaphe.

```
$doc = ooDocument
      (
      file           => "maitre.odt",
      member         => "content",
      create         => "text"
      );
$doc->appendParagraph
      (
      text           => "Ceci est une introduction",
      style          => "Standard"
      );
$doc->appendSection
      ("Annexe I", link => "/home/jmg/documents/a1.odt");
$doc->appendSection
      ("Annexe II", link => "http://labas.auloin.net/a2.doc");
$doc->appendParagraph
      (
      text           => "Ceci est la conclusion",
      style          => "Standard"
      );
$doc->save;
```

Cet assemblage est très simpliste. Une application plus réaliste, au lieu de partir d'un document vide, utiliserait un modèle de document avec une table des matières et une collection de styles bien étudiée.

8.2 Documents-réservoirs

Avec une interface de programmation telle qu'OpenOffice::OODoc, on peut créer et gérer des documents qui ne sont même pas destinés à être vus par des utilisateurs. Il est en effet très facile de construire et d'exploiter des archives compressées, formellement compatibles OpenDocument, mais dont le seul rôle est de stocker des définitions de contenu ou de style en vue de les utiliser pour la génération automatique de documents.

En matière de contenu, on peut citer notamment des blocs de textes correspondant à des formules obligatoires ou routinières (mentions légales, formules de politesse, articles types). Ces blocs de texte peuvent d'ailleurs contenir des champs variables ou encore des balises convenues qui seront automatiquement remplacées, au moment de la génération de chaque document et via une méthode comme `selectElementByContent()` ou `substituteText()`, par des données de l'application. On peut aussi mutualiser et réutiliser des illustrations, représentées par des éléments images. Autre possibilité intéressante, la mise à disposition de collections de pages de présentations, permettant de construire par programme, en quelques secondes, un support de présentation qui contiendra un sous-ensemble de cette collection sélectionné selon un ou plusieurs critères.

La copie d'un élément quelconque de contenu d'un document à un autre est très simple. Prenons comme exemple le cas d'un document, disons un courrier commercial, dans lequel nous voulons injecter deux paragraphes tirés d'un document-dictionnaire. Le premier est une introduction, à placer immédiatement après un paragraphe commençant par "Cher Client". Le second est une formule de politesse, à placer à la fin du corps de lettre. Nous supposons ici que le programme a déjà créé un connecteur `$lettre` correspondant au contenu de la lettre à composer, et un autre connecteur `$dico` correspondant au document-dictionnaire. Nous allons profiter de cet exemple pour introduire `selectElementByBookmark()` qui permet de capturer n'importe quel élément d'après un repère de texte (placé à l'aide du logiciel bureautique ou par un programme ayant utilisé `setBookmark()`, une autre méthode d'OpenOffice::OODoc). Plutôt que de sélectionner les paragraphes du dictionnaire avec `getParagraph()` d'après des numéros non significatifs, il est plus intelligent de se servir de repères de textes dont les noms ont bien sûr été choisis à bon escient. Au passage, nous verrons aussi une méthode `appendBodyElement()` qui permet d'ajouter un élément quelconque à la fin d'un corps de document.

```
$repere = $lettre->selectElementByContent("^Cher Client");
$intro = $dico->selectElementByBookmark("Intro lettre commerciale");
$lettre->insertElement($repere, $intro->copy, position => "after");
$fin = $dico->selectElementByBookmark("Formule fin lettre commerciale");
$lettre->appendBodyElement($fin->copy);
```

On a remarqué l'invocation systématique de la méthode `copy` sur les éléments extraits du dictionnaire. Il s'agit d'une méthode d'élément (et non d'une méthode de document) ; elle provient directement de XML::Twig. Son utilisation ici fait que ce sont des *répliques* des éléments extraits qui sont insérées dans le document cible. Rien n'interdit d'insérer les références originales elles-mêmes... à condition de savoir exactement ce qu'on fait en plaçant le même élément à plusieurs endroits, un sujet sur lequel nous ne pouvons pas nous étendre ici.

Il est tout aussi facile de consigner dans une même archive un nombre illimité de styles de toutes natures (notamment les styles de paragraphes et de pages les plus susceptibles d'être réutilisés), ainsi que des déclarations de polices de caractères. Cette solution, qui permet d'utiliser l'interface WYSISWYG d'un logiciel bureautique pour définir en quelques clics les styles les plus complexes, ce qui est certainement plus efficace que de développer et de maintenir des centaines d'instructions `createStyle()` assortie chacune de dizaines de paramètres dont la syntaxe n'est pas intuitive. La propagation d'un style d'un document à un autre est très facile. L'exemple ci-dessous montre comment on peut créer un style "Corps de lettre" en utilisant comme modèle un style du même nom tiré d'un autre document.

```
# accès au dictionnaire de styles
$librairie = ooDocument(file => "mes_styles.odt", member => "styles");
# accès aux styles du document cible
$lettre = ooDocument(file => "lettre.odt", member => "styles");
# création du nouveau style d'après le modèle
$lettre->createStyle
     (
     "Corps de lettre",
     prototype      => "Corps de lettre",
     source         => $librairie
     );
```

9 Conclusion

Dans cet article, nous avons balisé des débuts de pistes, sans faire le tour complet des possibilités. Le premier objectif était de démontrer que, en creusant un peu le sujet, on peut aller très loin dans la mise en relation de documents bureautiques enfin standardisés avec des applications de production de toutes natures. Le second était de permettre au lecteur d'évaluer l'adéquation de Perl et plus particulièrement de l'interface OpenOffice::OODoc à ses propres besoins actuels et futurs en matière de traitements documentaires.

Dans l'avenir, il est probable que cette interface évoluera, d'abord pour rester en phase avec les améliorations du standard, et ensuite pour prendre en considération de nouveaux besoins et/ou intégrer des contributions communautaires. De plus, on peut souhaiter voir apparaître d'autres interfaces, basées ou non sur OpenOffice::OODoc, spécialisées dans des domaines fonctionnels précis.

Bien entendu, pour tout besoin d'assistance technique ou de formation concernant les applications du format OpenDocument ou plus particulièrement la mise en oeuvre du module OpenOffice::OODoc, n'hésitez pas à prendre contact avec

jmgdoc@cpan.org

BIBLIOGRAPHIE

Spécification du format OASIS OpenDocument 1.0 :
http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf

Article *Wikipedia* OpenDocument :
http://fr.wikipedia.org/wiki/OpenDocument

Forum OpenDocument de l'OASIS :
http://opendocument.xml.org

Site *CPAN* du projet OpenOffice::OODoc :
http://search.cpan.org/dist/OpenOffice-OODoc

Forum de discussion d'OpenOffice::OODoc (en Anglais) :
http://www.cpanforum.com/dist/OpenOffice-OODoc

"Open Office Document Connector", *Dr.Dobb's Journal*, novembre 2005
http://www.ddj.com/dept/lightlang/184416230

"The Perl OpenDocument Connector", *The Perl Review*, Volume 3, Issue 1, décembre 2006
http://www.theperlreview.com

"OpenOffice::OODoc, Guide de prise en main",
http://jean.marie.gouarne.online.fr/doc/oodoc_guide.pdf

La référence du présent document est http://jean.marie.gouarne.online.fr/doc/perl_odf_connector.pdf