Qt et XML pour les fichiers de paramètres

Cette page est en XHTML1 et utilise les CSS2, il vous faut un browser compatible: netscape 6.2, Internet Explorer 6, Mozilla. Avec d'autres (notamment Netscape 4.x) la mise en page risque d'être aléatoire.

Cette page explique les bases de l'utilisation du module XML de Qt. Elle suppose que vous connaissez les bases de XML et de Qt, même si quelques rappels seront effectués. Elle ne se veut pas exhaustive mais *pragmatique*. L'objectif est d'utiliser XML pour les tâches suivantes :

- fichiers de configuration d'une application;

En effet, nous développons des algorithmes dépendant de pleins de paramètres qu'il faut spécifier à nos programmes (de tests, ou application finale). Or :

- on veut avoir un certains nombres de *jeu de paramètres tests*;
- le nombre de paramètres peut évoluer et il faut maintenir une *compatibilité ascendante*;
- on n'a pas de temps à perdre sur cet aspect de la programmation.

Pour cela nous utilisons la ligne de commande et des *fichiers de paramètres*. La ligne de commande n'est pas abordée ici mais (pub perso) vous pouvez aller voir la librairie ArgParser.

Cette page aborde le problème des fichiers de paramètres, c'est à dire comment écrire et lire des données dans un fichier. Ce problème comporte deux aspects, qui sont la façon de *représenter* les données et comment parser un fichier pour en extraire des paramètres.

Pourquoi XML?

Pour lire des fichiers de paramètres il y a plusieurs chemins que je voie (malheureusement) certains emprunter :

- parser "*à la main*" le fichier;
- écrire un parser avec lex et yacc;
- utiliser un format/une librairie de haut niveau (par exemple XML)

La première voie est une hérésie. Il faut se battre avec les commandes type strtok et autre token_iterator. Au final on perd son temps à réinventer la roue, et on le fait moins bien que les spécialistes *es* roues.

La deuxième solution est plus élégante mais nécessite un bonne connaissance de deux outils dont l'API archaïque place une grosse charge cognitive sur le

programmeur. En outre leur aspect profondemment C empêche la réutilisabilité et encourage de mauvaises pratiques de programmation.

En outre, ces approches ne résolvent pas le problème de la représentation des données. Chacun développe ainsi son format de données, et on se retrouve avec des soucis de conversion, d'évolutivité et de maintenance.

Faisons confiance aux spécialistes!

En effet, le problème de la représentation de données a été traité par des gens qui s'y connaissent. Ils ont donc déjà réfléchi à ce que doit être une bonne solution et ont mis en place des standards. En plus, ils fournissent les outils (librairies dans différents langages et utilitaires pour différentes plateformes) pour travailler confortablement!

Évolutivité

Supposons que vous développez un programme de rendu de pierres précieuses. Vous débutez un sujet de recherche que d'autres après vous vont continuer. Votre souhait le plus cher est évidemment que ce que vous mettez au point soit réutilisable par vos successeurs! Passons sur la nécessité de programmer aux normes pour garantir la portabilité et de concevoir avec sagesse pour assurer la réutilisabilité des composants, et regardons le problème des fichiers de paramètres.

Supposons que vous avez développé un programme de rendu de pierre précieuse. En sortie vous avez de belles images de synthèses, et en entrée vous prenez des fichiers de descriptions de pierres précieuses. Voilà par exemple les paramètres qui y sont spécifiés :

paramètre	example de fichier
nom de la pierre	name = diamond
facettes	nbFacets = 4 nbVertex = 3 Vertex = 0 0 0 Vertex = 0 1 0 Vertex = 1 1 0 nbVertex = 3 Vertex = 1 1 0 Vertex = 1 0 0 Vertex = 0 0 0 ...
couleur	color = 255 255 255
commentaires	# This is a comment until end of line

Un an après, vous n'êtes plus là mais votre programme compile et s'éxécute toujours. Votre successeur travaille sur un logiciel de cassage de pierre et souhaite définir de nouveaux paramètres. Par exemple :

paramètre	example de fichier
solidité	solidity = 23
pureté	purity = 10

Il récupère les 100 fichiers de pierre que vous lui avez laissé et les enrichi de ces paramètres. Son programme marche très bien mais patatrac! Voilà qu'il veut obtenir des images avec votre programme. Malheureusement, celui-ci refuse de lire les fichiers de paramètres "enrichis" car vous n'aviez pas prévu cette possibilité

d'extension. Seule solution, maintenir plusieurs fichiers de configuration, ou ajoute ces nuoveaux paramè tres dans des lignes de commentaires (à la Bright).

XML, comme c'est dit dans son nom (*Xtensible Markup Language*) est extensible et ré soud ce problè me.

API disponible

Pour é crire et lire vos fichiers de description de pierre, vous avez passé trois semaines à coder et dé boguer une librairie. Sur deux mois de stage, c'est beaucoup de temps perdu. Et si votre successeur dé cide de passer à Java ou caml, il doit ré ecrire une librairie.

Avec les librairies XML disponibles dans de nombreux langages, c'est l'affaire d'une heure comme peuvent en té moigner certains qui travaillent sur les pierres pré cieuses!.

Outils externes

Vous avez é crit toute une bibliothè que de fichiers de configuration dé crivant les nombreuses pierres existantes. Vous souhaitez les rendre consultables en ligne sous formes de belles pages web. Hu, Hu?!?

Avec XSLT, le langage de transformation de fichiers XML, il est facile de produire automatiquement des pages web qui, si elles sont bien faites, sont des fichiers XML dont l'apparence est en outre facilement configurable par des feuilles de style!

Tâ ches de base

- Écriture
- Lecture
- Édition externe

Si vous avez lu le paragraphe pré cé dent (*Pourquoi XML?*) vous ê tes à ce stade convaincus d'utiliser XML. Nous reprenons l'exemple de ce paragraphe (fichier de description de pierre pré cieuses) et dé taillons comment effectuer en C++ avec la librairie Qt les tâ ches de base.

Voici un exemple de fichier de description au format XML. Ce fichier dé montre toutes les possibilité s de repré sentation de donné es. Pour une discussion sur le choix de la repré sentation, voir la section *Style des fichiers*.

```
<!DOCTYPE stone
        PUBLIC "-//XADECK//DTD Stone 1.0 //EN"
        "http://www-imagis.imag.fr/Members/Xavier.Decoret/QT/XML/CODE/SOURCES/stone1.dtd"

<!--This file describe a jewel-->

<stone name="diamond">
    <appearance>
        <color>255 255 255 </color>
    </appearance>
    <geometry nbFacets="2">
        <triangle>
  0 0 0
  1 0 0
  1 1 0
        </triangle>
        <quad>
  0 0 1
  1 0 1
```

```
    1 1 1
    0 1 1
        </quad>
    </geometry>
</stone>
```

Quelques remarques sur ce fichier et sur XML :

- La première ligne spécifie la DTD, voir section *Spécification d'une DTD*;
- un document XML contient *forcément* un seul noeud racine;
- un noeud peut avoir des *attributs*;
- un noeud peut contenir du texte.

Voyons maintenant comment créer et lire un tel fichier.

Écriture

Le programme ci-dessous crée l'équivalent du fichier diamond.xml (le formatage est un peu différent mais ne change rien).

```
#include <qdom.h>
#include <iostream>

int
main(int, char **)
{
    QDomImplementation impl = QDomDocument().implementation();
    // document with document type
    QString name = "stone";
    QString publicId = "-//XADECK//DTD Stone 1.0 //EN" ;
    QString systemId = "http://www-imagis.imag.fr/DTD/stone1.dtd";
    QDomDocument doc(impl.createDocumentType(name, publicId, systemId));

    // add some XML comment at the beginning
    doc.appendChild(doc.createComment("This file describe a jewel" ));
    doc.appendChild(doc.createTextNode("\n")); // for nicer output

    // root node
    QDomElement stoneNode = doc.createElement("stone");
    stoneNode.setAttribute("name", "diamond");
    doc.appendChild(stoneNode);

    // appearance
    QDomElement appearanceNode = doc.createElement("appearance");
    QDomElement colorNode = doc.createElement("color");
    colorNode.appendChild(doc.createTextNode("255 255 255" ));
    appearanceNode.appendChild(colorNode);
    stoneNode.appendChild(appearanceNode);

    // geometry
    QDomElement geometryNode = doc.createElement("geometry");

    QDomElement triangleNode = doc.createElement("triangle");
    triangleNode.appendChild(doc.createTextNode("0 0 0 1 0 0 1 1 0" ));
    geometryNode.appendChild(triangleNode);

    QDomElement quadNode = doc.createElement("quad");
    quadNode.appendChild(doc.createTextNode("0 0 1 1 0 1 1 1 1 0 1 1" ));
    geometryNode.appendChild(quadNode);

    stoneNode.appendChild(geometryNode);

    cout<<doc.toString();
}
```

Cet exemple parle de lui-même et en copiant-collant à partir de lui, vous devriez pouvoir faire ce que vous voulez.

Lecture

Pour la lecture, c'est un peu plus compliqué. Tout d'abord, précisons que nous

dé crivons ici une utilisation basé e sur le modè le DOM. En gros, cela veut dire que le fichier XML est chargé *entiè rement* en mé moire sous forme d'un arbre et que vous devez ensuite parcourir les noeuds de cet arbre pour y ré cupé rer les informations qui vous inté ressent (et ignorer les autres). Il existe un autre modè le, appelé SAX dans lequel le fichier est parcouru au vol et des *callbacks* que vous spé cifiez sont appelé es pour chaque type de noeud. Ce modè le n'est pas dé taillé ici pour l'instant.

La premiè re chose à signaler, c'est que Qt ne vé rifie pas (pour l'instant) la conformité du fichier. Pour cela, vous pouvez utiliser des outils externes (voir la section Spé cification d'une DTD). On fait donc l'hypothè se suivante :

> Le fichier pris en entré e est bien formé . Il pourra donc ê tre correctement lu par Qt (pas de tag mal refermé par exemple). D'autre part, il contient les informations qu'on est en droit d'attendre.

On ne s'occupera donc pas de faire de la signalisation ni de la ré cupé ration d'erreur.

Le principe est simple. On parcourt chaque noeud et ses fils en connaissance de leur organisation hié rarchique. Il n'y a donc pas de ré cursivité car on sait exactement à quoi s'attendre pour chaque noeud. Par contre, il se peut que l'on rencontre des noeuds qu'on ne sait pas traiter (parce que la DTD a é té é tendue, voir la section Évolutivité) et on doit donc tester le nom des é lé ments.

L'exemple ci dessous montre comment ré cupé rer la couleur d'une pierre et le nombre de triangles et de quads dans ses facettes.

```
#include <qfile.h>
#include <qdom.h>
#include <iostream>

int
main(int , char **argv)
{
    unsigned int nbTriangles = 0;
    unsigned int nbQuads = 0;

    QDomDocument doc;
    //*********************************
    // Read the DOM tree form file
    //*********************************
    QFile f(argv[1]);
    f.open(IO_ReadOnly);
    doc.setContent(&f);
    f.close();
    //*********************************
    // Parse the DOM tree
    //*********************************
    // The root node is supposed to be a "stone" tag, we retrieve its name
    QDomElement root=doc.documentElement();
    cout<<"The stone name is " <<root.attribute("name", "NO NAME" )<<endl;
    // We traverse its children
    QDomElement child=root.firstChild().toElement();
    while(!child.isNull())
    {
        // We know how to treat appearance and geometry
        if (child.tagName() == "appearance")
        {
            // We traverse children
            QDomElement grandChild=child.firstChild().toElement();
            while(!grandChild.isNull())
            {
                // We know how to treat color
                if (grandChild.tagName() == "color")
                {
                    cout<<"The stone colour is " <<grandChild.text()<<endl;
                }
                grandChild = grandChild.nextSibling().toElement();
            }
        }
        else if (child.tagName() == "geometry")
```

```
            {
                // We traverse children
                QDomElement    grandChild=child.firstChild().toElement();
                while(!grandChild.isNull())
                {
                    // We know how to treat triangle and quad
                    if    (grandChild.tagName()    ==    "triangle")    ++nbTriangles;
                    else    if    (grandChild.tagName()    ==    "quad")    ++nbQuads;
                    grandChild    =    grandChild.nextSibling().toElement();
                }
            }
            child    =    child.nextSibling().toElement();
        }

        cout<<"There are "      <<nbTriangles<<" triangle(s)"    <<endl;
        cout<<"There are "      <<nbQuads<<" quad(s)"    <<endl;

        return    0;
}
```

Remarquez que l'on ne vérifie pas que le noeud racine est bien de type "stone", mais l'on pourrait par sécurité. Remarquez aussi aue XML est *case sensitive* et qu'on a donc pas de souci à se faire lors de la comparaison de chaîne de caractères et de noms de *tag*.

Vous pouvez surtout voir que la façon de traverser l'arbre DOM permet d'ignorer les noeuds qui pourrait être présents dans une DTD étendue. Par exemple un sous-noeud d'*appearance* décrivant le coefficient de diffraction et tout une partie *physic* décrivant les paramètres physiques. Le fichier diamond-extended.xml ci-dessous est donc bien parsé.

```
<!DOCTYPE   stone
          PUBLIC   "-//XADECK//DTD Stone 2.0 //EN"
          "http://www-imagis.imag.fr/Members/Xavier.Decoret/QT/XML/CODE/SOURCES/stone1.dtd"

<!--This file describe a jewel-->

<stone    name="diamond">
      <comment>
    The diamond is a beautiful stone!
      </comment>
      <appearance>
            <color>255 255 255      </color>
            <diffraction>0.46</diffraction>
      </appearance>
      <geometry    nbFacets="2">
            <triangle>
    0 0 0
    1 0 0
    1 1 0
            </triangle>
            <quad>
    0 0 1
    1 0 1
    1 1 1
    0 1 1
            </quad>
      </geometry>
</stone>
```

Édition externe

Dans cette section, on montrera l'exemple d'une transformation XSLT pour faire une bel page web représentant diamond.xml.

Remarques avancées

- Utilisation du *Document Type*
- Style des fichiers

- Spé cification d'une DTD

Utilisation du *Document Type*

Vous pouvez utiliser le *document type (DOCTYPE)* d'un document pour dé terminer le numé ro de version d'un fichier XML. L'exemple ci-dessous vé rifie qu'un fichier est bien du type "stone" et affiche le numé ro de version.

```
#include    <qfile.h>
#include    <qdom.h>
#include    <qregexp.h>
#include    <iostream>

int
main(int ,          char    **argv)
{
        QDomDocument    doc;
        //*********************************
        // Read the DOM tree form file
        //*********************************
        QFile    f(argv[1]);
        f.open(IO_ReadOnly);
        doc.setContent(&f);
        f.close();
        //*********************************
        // Check the doctype
        //*********************************
        QDomDocumentType    type    =    doc.doctype();
        if    (type.name()    !=    "stone")
        {
                cerr<<"ERROR: this XML file is not a stone one"                    <<endl;
        }
        else
        {
                QRegExp    regexp("-//XADECK//DTD Stone ([^ ]+) //(.+)"          );
                if    (regexp.search(type.publicId())    !=    -1)
                {
                        cout<<"Version number = "          <<regexp.cap(1)<<endl;
                }
                else
                {
                        cerr<<"ERROR: malformed publicId"        <<endl;
                }
        }
        return    0;
}
```

Style des fichiers

- attribut ou texte?

Spé cification d'une DTD

Le problè me de la documentation est un impondé rable de l'informatique. Combien de fois avez vous pesté contre un programme mal documenté , ou contre un fichier de configuration ou un format de donné à la syntaxe obscure? XML contient en son sein le mé canisme de documentation : il s'agit de la DTD. En gros, ce document stipule la syntaxe d'un document d'un type donné (la partie DOCTYPE de nos fichiers XML).

> La DTD est donc un moyen standardisé de documenter le format de vos fichiers et de permettre aux autres de les é diter. Il permet aussi à des outils syntaxiques de vé rifier la validité d'un fichier et de localiser d'é ventuels erreurs.

Écrivez des DTD!

Comme on l'a déjà dit, Qt ne vérifie pas encore les DTD et vous pouvez donc omettre d'en écrire une. Pourtant, je vous encourage vivement à le faire car en plus, c'est facile! Cependant, on ne rentrera pas ici pas dans le détail technique de l'écriture d'une DTD. Consultez un site approprié pour cela. Voici simplement la DTD des document de types "stone" qu'on a utilisé ici.

```
<!ELEMENT stone (appearance?|geometry)>
<!ELEMENT appearance color>
<!ELEMENT color (#PCDATA)>
<!ELEMENT geometry (triangle+|quad+)>
<!ATTLIST geometry nbFacet (#PCDATA) #IMPLIED>
<!ELEMENT triangle (#PCDATA)>
<!ELEMENT quad (#PCDATA)>
```

Vos documents XML commenceront alors par une commande DOCTYPE qui stipule :

- le nom du type
- un identifiant public qui donne les détails (version, langue, ...)
- une URL pour accéder à la DTD

Page maintained by Xavier Décoret