5 Bases Relationnelles-Objet

Le relationnel-objet sur Oracle

On *simule* la définition de classes, donc l'imbrication de structures typique du modèle objet, dans un SGBD qui est relationnel.

Definition de Types (UDT)

Oracle permet de définir des types du style objet. Syntaxe :

```
CREATE TYPE t AS OBJECT (
    liste d'attributes et methodes
);
/
```

Le slash à la fin sert pour faire traiter par Oracle la définition de type.

Definition de Types, suite

Exemple. Définition d'un type pour représenter un point par ses coordonnées :

```
CREATE TYPE PointType AS OBJECT (
    x NUMBER,
    y NUMBER
);
/
```

NB :
Analogie avec la définition d'un type structuré ayant deux champs, qui ont, eux, un type atomique, en ODL.

Definition de Types, suite

Un type à objet peut être utilisé pour déclarer d'autres types à objet ou relations (au sense traditionnel, mais pas en FN1), par ex. on peut définir un type pour les lignes (géométriques) finies en indiquant les 2 extremités :

```
CREATE TYPE LineType AS OBJECT (
    end1 PointType,
    end2 PointType
);
/
```

NB :
Analogie avec la définition d'un type structuré ayant deux champs, ayant eux mêmes un type structuré (`PointType`), en ODL.

Definition de Types, suite

On peut alors crér une relation (presque traditionnelle) qui est un ensemble de lignes, où chaque ligne a un OID :

```
CREATE TABLE Lines (
    lineID INT,
    line   LineType
);
```

N.B.
Analogie avec la définition d'une classe en ODL, ayant deux attributes, `lineID` et `line`, où le second a un type structuré (donc la table "n'est pas en première forme normale").

Elimination de Types

Pour éliminer un type comme `LineType` :

```
DROP TYPE Linetype;
```

(“to drop” en anglais : faire tomber, jeter.)

N.B. On doit d’abord effacer toutes les tables et les autres types qui utilisent ce type (dans l’exemple : la table `Lines`).

Construction d'objets (valeurs). Constructeurs pre-définis pour crér des valeurs, dont les noms sont les noms des types.

Suite de l'exemple. Pour crér une valeur de type `PointType` : le mot `PointType` (nom du type) suivi par les valeurs en ().
Construction d'une ligne de la table `Lines` ayant idéntificateur 27, qui part du point (0,0) et arrive au point (3,4) :

```
INSERT INTO Lines
    VALUES(27, LineType(
                    PointType(0.0, 0.0),
                    PointType(3.0, 4.0)
                )
    );
```

N.B. On utilise `INSERT` comme pour l'insertion dans une table standard. On a créé aussi 2 valeurs de type `PointType` et 1 valeur de type `LineType`.

Déclarations de méthodes. Une déclaration de type peut inclure la déclaration de méthodes, à l'aide de `MEMBER FUNCTION` ou `MEMBER PROCEDURE`. Comme en ODL, il faut spécifier les types des entrées et sortie.

Suite de l'exemple. On ajoute une fonction `lenght` au type `LineType`, qui produit la longueur d'une ligne et la multiplie par un facteur.

```
CREATE TYPE LineType AS OBJECT (
    end1 PointType,
    end2 PointType,
    MEMBER FUNCTION length(scale IN NUMBER) RETURN NUMBER,

);
/
```

Définition du code d'une méthode

Le code de la méthode est donné à part, dans une instruction `CREATE TYPE BODY`.
Ici, on ne répéte pas les types des entrés (et des sorties des procédures). La variable spéciale `SELF` dans la déf. de la méthode indique la valeur courante.

Suite de l'exemple

```
CREATE TYPE BODY LineType AS
   MEMBER FUNCTION length(scale NUMBER) RETURN NUMBER IS
       BEGIN
         RETURN scale *
            SQRT((SELF.end1.x-SELF.end2.x)*(SELF.end1.x-SELF.
                 (SELF.end1.y-SELF.end2.y)*(SELF.end1.y-SELF.
            );
       END;
END;
/
```

Quelques exemples de requête relationnelle-objet

Calcul du double de la longueur des chaque ligne :

```
SELECT lineID, ll.line.length(2.0)
 FROM Lines ll;
```

La requête utilise la méthode `length`.

Il faut utiliser un alias (ou variable) (ici, `ll`) pour acceder à un champ avec la notation "`.`". En fait, `line` tout seul et `Lines.line` ne marchent pas.

Coordonnées de la prèmiere extremité de chaque ligne :

```
SELECT ll.line.end1.x, ll.line.end1.y
     FROM Lines ll;
```

Relations imbriquées, déclaration du schéma

La valeur d'un attribut d'une table peut être une relation.

Définition du schéma d'une table dont les élément sont des polygones, tout polygone étant une table (de points) lui-même.

```
CREATE TYPE PolygonType AS TABLE OF PointType;
/

CREATE TABLE Polygons (
    name   VARCHAR2(20),
    points PolygonType)
    NESTED TABLE points STORE AS PointsTable;
```

Dernière ligne : les sous-tables réprésentant les polygones ne sont pas stockées directement comme valeurs de l'attribute `points`, dans une table Polygons qui ne serait pas en FN1. C'est juste une *façon de voir*. En realité, Oracle crée une table ordinaire, dont le nom est `PointsTable`, où on stocke des n-uplets ordinaires.

Relations imbriquées, remplissage des tables

La valeur de chaque relation de niveau inférieur est représentée par une liste de valeurs du type approprié (`PolygoneType` dans notre exemple).

Exemple d'insertion d'un polygone qui est un carré :

```
INSERT INTO Polygons VALUES(
    'square', PolygonType(PointType(0.0, 0.0), PointType(0.0,
                          PointType(1.0, 0.0), PointType(1.0,
                         )
);
```

Relations imbriquées, exemples de requêtes

Les points du carré :

```
SELECT points
FROM Polygons
WHERE name = 'square';
```

Le point du carré qui sont sur la diagonale (x=y) :

```
SELECT ss.x
FROM THE(SELECT points
         FROM Polygons
         WHERE name = 'square'
        ) ss
WHERE ss.x = ss.y;
```

(sous-requête, relation du niveau inférieur dans le `FROM` avec le mot-clé `THE` et utilisation de la variable `ss` pour la nommer.)

6 XML et les Données Semi-structurées

– L’apparition de XML (*eXtensible Markup Language*) (plus “évolué” que HTML) a mené au nouveau concept de *données semi-structurées*.
– XML : standard W3C d’échange de données sur le Web. Permet un échange sur un format standard, independemment des formats de stockage de ces données.
– Grande flexibilité.
– Multitude de standards associés :
  - Formats de Schémas : DTD et XML-schéma
  - Langages de Requête : XPATH, XQUERY (extension de XPATH),...
  - XSLT : notation pour transformer un document XML d’un format à un autre.
  etc.
– Lien avec le cours *Documents Structurés* ?

Parmi les applications BD :

– Intégration de Données (“mediateur”).
– Bases de Données en Biologie.

6.1 HTML

HTML (*Hyper Text Markup Language*) : un standard d'écriture de documents pour le Web.

HTML est un langage à **balises** ("étiquettes"). Ces balises sont **fixes**, à fonctions prédéfinies.

Les balises de HTML permettent de :

– **Mettre en forme un texte**
  Ex. `<B> </B>, <I> </I>, <CENTER> </CENTER>`,....
– **Créer des liens** (balises "amarres").
  Ex :
  `<A HREF="http://www.univ-evry.fr/">Université d'Evry Val d'Essonne </A>`

Le rôle des balises autres que les “amarres” est celui de *présenter visuellement* du *texte* en un certain format.

Par exemple :

– `<B> bla </B>` sert à écrire **bla** à la place de bla
– `<I> bla </I>` sert à écrire *bla* à la place de bla
– `<CENTER> bla </CENTER>` sert à écrire

  bla

  à la place de :

  bla
– `<H1> bla </H1>`, `<H2> blabla </H2>` `<H3> blablabla </H3>` servent à introduire des titres (des “sections”), par ordre d’importance décroissant.

6.2 Limites de HTML

HTML n’est pas adapté à l’interrogation des données.

Il permet de mettre en FORME un texte.

Il ne permet pas de STRUCTURER “logiquement” un contenu.

Exemple

Une organisation publies des données stockées dans une BD relationnelle. Des pages web sont crées.

Une autre organisation veut une analyse de ces données ; son logiciel a acces seulement aux pages HTML.

– Une petite modification du format d'un élément d'une page web peut casser ce logiciel !
– Même si on a besoin seulement de la valeur moyenne d'une colonne d'une table, on peut avoir besoin de charger une base entière via plusieurs requêtes de pages HTML.

7 XML

XML : nouveau standard adopté par le World Wide Web Consortium (W3C) comme complément de HTML permettant un échange aisé de données de sur le web.

– Le but principal de XML n'est pas de décrire un format de texte, mais de **structurer** logiquement un contenu.
– Les balises ont le rôle de classer des données selon une hiérarchie définie par l'auteur du document XML.

– Avec XML, la mise en forme textuelle est effectué dans une *feuille de style*, un document separé qui associe des formes de présentations (texte en gras, en italique, centré, etc.) aux balises. Des feuilles différentes permettent des formattages différents du même document.
– Des outils permettent de convertir un document XML en HTML, afin de pouvoir afficher une page web.

Example 1 Un petit document XML

```
<?xml version=’’1.0 encoding=’’iso-8859-1’’?>

<communication prior=’’important’’>
<pour> Virginie </pour>
<sujet> Rappel </sujet>
<message> N’oublie pas de lire l’article
<lire> Lutz et al. 2002. </lire>
Il faut bien comprendre
<reflechir> la preuve de terminaison.  </reflechir>
Rendez-vous <date> mercredi </date> <lieu> dans mon bureau </lieu>
</message>
<signature> Serena </signature>
</communication>
```

Suite de l’exemple

Résultat de la mise en forme grâce à une feuille de style :

Priorité : important
Pour : Virginie
Sujet : Rappel

N’oublie pas de lire l’article *Lutz et al. 2002.* Il
faut bien comprendre **la preuve de terminaison.** Rendez-vous **mercredi** *dans mon bureau*

Serena

Suite de l’exemple

Résultat de la mise en forme avec une **autre** feuille de style :

Priorité : IMPORTANT
Pour : VIRGINIE

Sujet : RAPPEL

N’oublie pas de lire l’article *Lutz et al. 2002.*

Il faut bien comprendre *la preuve de terminaison.*

Rendez-vous **mercredi dans mon bureau**

Serena

XML modèlise des informations :

– En organisant les données en un *graphe* d'*objets complexes*
– En les structurant de façon plus *flexible* par rapport au au modèle relationnel ou objet : les données sont dites *semistructurées*

graphe, *objet complexe*, *flexible*, *semistructuré* = ? ? ? ?

$\Rightarrow$ Voir la suite...

7.1 Syntaxe de base de XML

La composante essentielle est l'*élément*, un morceau de document delimité par une balise d'ouverture (ex. `<toto>`) et une de fermeture (ex. `</toto>`.

Un élement peut contenir du texte, des autres élements (→ "objet complexe"), ou un mélange des deux.

– Les balises (leur noms) **sont définies par les utilisateurs**.
– Elles n'ont pas de signification prédéfinie : elles indiquent seulement comment structurer le document sous forme de arbre (ou, plus généralement, de graphe).

Example 2

```
<personne>
<nom> Alan  </nom>
<age> 42 </age>
<email> agb@abc.com </email>
</personne>
```

Ici on a :

– *Un élément complexe de “sorte” (“type”)* `personne`*, qui consiste d’un triplet d’éléments ayant les “sortes”* `nom`,`age`,`email`.
– *Un élément* `Alan` *de “sorte”* `nom`
– *Un élément* `42` *de “sorte”* `age`
– *Un élément* `agb@abc.com` *de “sorte”* `email`

Suite de l’exemple

Le contenu de ce document peut être représenté :

– Soit par un arbre où les *noeuds internes* sont etiquéttés par les baliseses.

– Soit par un arbre où les *arcs* sont etiquéttés par les balises.

FIGURES AU TABLEAU

Example 3

```
<gens>
<personne>
<nom> Alan  </nom>
<age> 42 </age>
<email> agb@abc.com </email>
</personne>
<personne>
<nom> Patricia  </nom>
<age> 36 </age>
<email> ptn@abc.com </email>
</personne>
</gens>
```

Remarque : on peut utiliser plusieurs éléments ayant la même balise pour représenter une collection.

Dans l’exemple 3, une entité de “sorte” `gens` est une collection de personnes...

A nouveau, on peut représenter ces informations sous forme d’un arbre, avec 2 possibilités.

Example 4

```
<biblio>
<livre>
<auteurs>
<nom> Abiteboul </nom>
<nom> Bunemann </nom>
<nom> Suciu </nom>
</auteurs>
<titre> Data on the Web </titre>
<edition>
<nom-ed> Morgan Kaufman </nom-ed>
<adresse-edition>
<rue-ed> 340, Pine Street </rue-ed>
<ville-ed> San Francisco </ville-ed>
<etat-ed> California </etat-ed>
<pays-ed> USA </pays-ed>
</adresse-edition>
</edition>
</livre>
<livre>
.
.
.
</livre>
</biblio>
```

7.2 Pourquoi “objet complex” ?

Comparer avec les BD relationnelles (en première forme normale), où le domaine de tout attribut contient seulement des valeurs atomiques, et toute “entité est plate” :

nom	titre	nom-ed	rue-ed	ville-ed	etat-ed	pays-ed
Abit	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
Bune	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
Suciu	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
⋮						

Dans le document XML de l’exemple 4, un livre est un objet **complex**, composé d’une séquence d’auteurs, d’un titre et d’une adresse (comme dans le BD à objet). La première et la troisième composantes sont elles mêmes des objets complexes.

7.3 Pourquoi “semistructuré” ?

Un premier élément de réponse :
un objet complexe peut avoir des composantes optionnelles, le “schéma” de la base n’est pas rigide.

Différence par rapport au modèle relationnel, où le nombre d’attributs du schéma d’une relation est fixé en avance.

Example 5 *Un livre peut* **éventuellement** *être stocké avec son prix ; le champ* `pays-ed` *est aussi optionnel :*

```
<biblio>
<livre>
<auteurs>
<nom> Abiteboul </nom>
<nom> Bunemann </nom>
<nom> Suciu </nom>
</auteurs>
<titre> Data on the Web </titre>
<edition>
<nom-ed> Morgan Kaufman </nom-ed>
<adresse-edition>
<rue-ed> 340, Pine Street </rue-ed>
<ville-ed> San Francisco </ville-ed>
<etat-ed> California </etat-ed>
<pays-ed> USA </pays-ed>
</adresse-edition>
</edition>
<prix-en-euros> 44 <prix-en-euros>
</livre>
<livre>
<auteurs>
<nom> Gardarin </nom>
</auteurs>
<titre> Internet/Intranet et Bases de Donn\’ees </titre>
<edition>
<nom-ed> Eyrolles </nom-ed>
<adresse-edition>
<rue-ed> 61, Bld Saint Germain </rue-ed>
<ville-ed> Paris </ville-ed>
```

```
<etat-ed> France </etat-ed>
</adresse-edition>
</edition>
</livre>
</biblio>
```

Dans les exemples vus jusqu'à ici, les données sont organisées en arbres. Les réferences produisent des **graphes**.

On peut faire référence à un sommet déjà existant dans le graphe car on peut associer à un *idéntificateur* à chaque élément.

Pour "pointer" vers un élément ayant identificateur, disons `cle` (nom choisi par l'auteur), on exploite l'existence des *attributs* XML.

En général, un attribut XML sert à dèfinir une **proprieté des données** ; sa valeur est une chaîne de caractéres. (Dans l'exemple 1, `prior` était un attribut).

La syntaxe générale de la déclaration d'attributs est :

```
<balise attribut1=valeur1 ... attributN = valeurN> ...
</balise>
```

Par ex. :

```
<nom langue=français> Abiteboul </nom>
```

```
<nom langue=anglais> Bunemann </nom>
```

Pour identifier un élément il suffit d'utiliser un attribut dont le type déclaré (où ?) est ID :

`<balise attribut=valeur> </balise>` marche, à condition que `attribut` soit de type ID.
La syntaxe abrégée `<balise attribut=valeur/>`
est aussi possible.

Par ex. :

```
<Livre ISBN="isbn-95456255"/>
```

C'est dans le schéma du document XML (voir après) que l'on on déclare l'attribut ISBN comme ayant le type ID.

Pour faire **référence** à un élément on utilise un attribut de type IDREF :

```
<balise attributRef= identificateur> </balise>
```

La syntaxe abrégée

```
<balise attributRef= identificateur /balise>
```

est aussi possible.

Par ex :

```
<Edition Editeur="LinuxFrench Edition1" EditeurRef= "LFNET"
/>
```

Ici, on déclarera l'attribut `EditeurRef` comme ayant le type IDREF.

Un élément de la forme :

```
<balise attributRef= identificateur /balise>
```

où `attributRef` est de type IDREF n'a pas de contenu. Il est dit dit *élément vide*.

Example 6

```
<geographie-USA>
</etats>
<etat>
<etat cle = ’’e1’’>
<code-etat> IDA </code-etat>
<nom-etat> Idaho </nom-etat>
<capitale ref-cap = ’’v1’ />
<villes-dans  ref-a-villes = ’’v1’’ />
<villes-dans  ref-a-villes = ’’v3’’ />
...
</etat>
<etat>
...
</etat>
</etats>
<villes>
<ville>
<ville iden = ’’v1’’>
<code-ville> BOI </code-ville>
<nom-ville> Boise </nom-ville>
<etat-de-la-ville ref-a-etat = ’’e1’’ />
</ville>
<ville>
<ville iden = ’’v2’’>
<code-ville> CCN </code-ville>
<nom-ville> Carson City </nom-ville>
<etat-de-la-ville ref-a-etat = ’’e2’’ />
</ville>
<ville>
<ville iden = ’’v3’’>
<code-ville> MO </code-ville>
```

```
<nom-ville> Moscow </nom-ville>
<etat-de-la-ville ref = ’’e1’’ />
</ville>
...
</villes>
</geographie-USA>
```

La possibilité de faire des références fait passer de la structure de *arbre* à celle plus générale de *graphe* orienté avec une racine.

Figure au tableau.

XML permet de mélanger des données textuelles et des sous-éléments au sein d’un élément :

Example 7
```
<personne>
Voici mon meilleur ami
<nom> Alan  </nom>
<age> 42 </age>
Je ne suis pas sure de l’adresse e-mail suivante˜:
<email> agb@abc.com </email>
</personne>
```

Pas naturel du point de vue BD, mais du à l’origine de XML comme langage de documents hyper-texte.

7.4 Schémas pour des documents XML

Deux formats :

1. Un *DTD* (**D***ocument* **T***ype* **D***efinition*)
2. Un *XML-schema*, qui a une structure de typage plus riche.

7.4.1 Les DTD

Un DTD peut être vu comme une sorte de schéma pour les données XML. Il est **optionnel** $\Rightarrow$ données **semistructurées**.

Une document XML qui, en outre d'être syntaxiquement correct, a un DTD, et le respecte, et dit *valide*.

Syntaxe des DTD

Un document XML est composé de deux lignes initiales optionnelles suivie par une suite des éléments. La première des deux lignes initiales indique la version de XML utilisée, la deuxième contient le DTD.

```
<? xml version=’’1.0’’?>
<?DOCTYPE nom [Declarations-de-Type]>
<nom> ... </nom>
```

La balise `nom` est la balise racine. `Declarations-de-Type` est une suite $D\acute{e}claration_1$,...,$D\acute{e}claration_n$
où toute déclaration introduit le nom d’un élément et sa “sorte”, c.à.d une déscription de la “forme de son contenu”.

Syntaxe d’une Déclaration du DTD

Pour le moment, ignorons les déclarations des types des attributs.

Chaque déclaration du DTD est constituée du symbole <, puis de la chaîne de caractères `!ELEMENT`, puis d’une balise, puis d’un modèle de contenu, et, enfin, le délimiteur de fin > :

< `!ELEMENT` balise modèle_contenu>

Il y a cinq sortes différents de modèles de contenu.

Modèles de contenu dans une déclaration d'un DTD

1. Contenu vide : `<!ELEMENT balise EMPTY >`
2. Pas de contraintes sur le contenu : `<!ELEMENT` balise `ANY>`.
   (NB : données "semistructurées" !)
3. Elément ne contenant que des données textuelles :
   `<!ELEMENT balise #PCDATA>`
4. Elément ne contenant que d'autre éléments : `<!ELEMENT` balise motif `>`
   où `motif` est une *expression XML-regulière* sur l'alphabet des noms des balises.
5. Eléments de contenu "mixte" : mélange à la fois d'éléments et de données textuelles.

Les opérateurs des expression XML-regulières

– Le symbole , indique la concaténation.
Exemple : `chat,chien` signifie qu'un chien doit suivre un chat. L'ordre compte dans les documents XML (arbres ordonnés). (**pourquoi ? ?**)
– Le symbole | est le XOR logique.
Exemple : `chat` | `tortue` | `chien` signifie que soit un chat soit une tortue soit un chien est acceptable (mais un seul de ces animaux).
– Le symbole ? rend l'expression immédiatement précédente optionnelle.
Exemple : (`chat,chien`) ? signifie que une suite d'un chat puis d'un chien peut être placée à cet endroit, ou omise.

Les opérateurs des expression XML-regulières, suite

– Le symbole $+$ signifie qu'une suite non vide d'éléments conformes à l'expression immédiatement précédente est réquise.
Exemple : (`chat` | `chien`)$^+$ signifie qu'il doit y avoir un nombre non nul de chats et de chiens.
– Le symbole $*$ signifie qu'une suite éventuellement vide d'éléments conformes à l'expression immédiatement précédente est réquise.
Exemple : (`chat`,`chien`)$*$ signifie que, à cet endroit, ou bien il n'y a rien du tout, ou alors il y a une suite de chats et chiens telle que tout chat est immédiatement suivi par un chien et tout chien est immédiatement précédé par un chat.

Example 8

```
<!ELEMENT article
(titre, sous-titre?, auteur*, (paragraphe|table|figures)+, biblio
```

Cette déclaration décrit le contenu d’un article comme étant composé d’un titre suivi éventuellement d’un sous-titre, puis de 0 ou plusieurs auteurs, puis d’une combinaison (non-vide) de paragraphes, tables et figures, puis, éventuellement, d’une bibliographie.

Example 9 *Un exemple simple de DDT.*

```
<!DOCTYPE gens [
<!ELEMENT gens (personne)*>
<!ELEMENT personne (nom, age, e-mail)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
]>
```

Le document de l'exemple 3 est valide par rapport à ce DTD.

En résumant, il y a plusieurs raisons pour lesquelles on peut qualifier des données représentées dans un document XML comme étant *semi-structurées* :

1. Le document n'a pas de schéma (DTD ou XML-schéma). Dans ce cas, on a juste du texte, que l'on ne sait pas comme interroger ! (Sauf par recherche de mot clé, comme pour les documents HTML)
2. Le document a un schéma. Mais :
   (a) Une déclaration de la forme `<!ELEMENT` balise `ANY>` ne donne aucune information sur la structure.
   (b) Une déclaration comportant ? prévoit l'optionalité d'une balise B dans le motif associé à une balise A. (Mais : penser aux valeurs nulles dans les SGBD relationnels...)

A la place d’inclure le DTD dans le document, on peut aussi le sauver dans un fichier separé, qui peut être placé à une URL différente. Ceci permet à différents sites web de partager un unique schéma.

Déclaration d'attributs dans un DTD

Un DTD permet aussi de déclarer le type des attributs. Par exemple `ID` est le type des attributs permettant de donner des idéntificateurs aux éléments (voir l'exemple 6 : `cle` est de type `ID`). Si un attribut A est déclaré comme ayant le type IDREF, ceci indique que la valeur de A est un idéntificateur d'un élément (l'élément pointé").

Le mot-clé ATTLIST est utilisé pour déclarer le type d'une liste d'attributs.

En outre de déclarer le type des attributs, on décrit leur comportement ; le mots-clés `#REQUIRED, #IMPLIED` indiquent, respectivement, si un attribut est obligatoire ou optionnel.

Syntaxe d'une déclaration de types pour une liste d'attributs :

`<!ATTLIST` $nom_{att1}\ type_{att1}\ descr_{att1}, ..., nom_{attN}\ type_{attN}\ descr_{attN}$`>`

Example 10 *Un DDT pour les données de l'exemple 6 :*

```
<!DOCTYPE  geographie-USA [
<!ELEMENT geographie-USA (etats|villes)*>
<!ELEMENT etats (etat)*>
<!ELEMENT etat (code-etat,nom-etat,capitale,villes-dans*)>
<!ATTLIST etat cle ID #REQUIRED>
<!ELEMENT code-etat (#PCDATA)>
<!ELEMENT nom-etat (#PCDATA)>
<!ELEMENT capitale EMPTY>
<!ATTLIST capitale ref-cap IDREF #REQUIRED>
<!ELEMENT villes-dans EMPTY>
<!ATTLIST villes-dans ref-a-villes IDREFS #REQUIRED>
<!ELEMENT villes (ville)*>
<!ELEMENT ville (code-ville,nom-ville,etat-de-la-ville)>
<!ATTLIST ville iden ID #REQUIRED>
<!ELEMENT code-ville (#PCDATA)>
<!ELEMENT nom-ville (#PCDATA)>
<!ELEMENT etat-de-la-ville EMPTY>
<!ATTLIST etat-de-la-ville ref-a-etat IDREF #REQUIRED>
]>
```

Un défaut des DTD : on ne peut pas déclarer que les sortes des valeurs de l'attribut `ref-cap`*, par exemple, sont des villes. Les DTD n'offrent pas un typage adéquat des données semistructurées.*