Organisation des cours et TD

- 3 parties assez distinctes qui présentent différents aspects relevant du Génie Logciel :

1. Explication et mise en pratique de la gestion de logiciels : installation et configuration.
2. Méthodes de modélisation des données.
3. Méthodes avancées de modélisation et de programmation.

Installation - configuration de logiciels

- Les paquetages, leurs formats.
- Configurer une installation.
- Installer un logiciel
- Délivrer un logiciel à un client.

Formats de Paquetages

Quelques rappels :

- Les binaires sont dépendants des systèmes d'exploitation.
- Les outils nécessaires à l'utilisation d'un logiciel : librairies graphiques, fontes textes, compilateurs ou machines virtuelles peuvent ou non être distribués avec le logiciel.
- Le choix des différentes solutions influe grandement sur la taille du paquetage à délivrer.

Formats de Paquetages

- Livraison d'exécutables : `.exe` sous Windows. Aucune reconfiguration possible par l'utilisateur. C'est l'utilisateur qui doit s'assurer que son environnement est capable de faire tourner le logiciel.

Formats de Paquetages

- Livraison d'exécutables : `.exe` sous Windows. Aucune reconfiguration possible par l'utilisateur. C'est l'utilisateur qui doit s'assurer que son environnement est capable de faire tourner le logiciel.
- Livraison de paquetages *customisables* - scripts d'installation plus ou moins modifiables par l'utilisateur.

Formats de Paquetages

- Livraison d'exécutables : `.exe` sous Windows. Aucune reconfiguration possible par l'utilisateur. C'est l'utilisateur qui doit s'assurer que son environnement est capable de faire tourner le logiciel.
- Livraison de paquetages *customisables* - scripts d'installation plus ou moins modifiables par l'utilisateur.
- Utilisation de formats de paquetages qui s'appuient sur un *Manager* de paquetage : `.rpm`, `.deb`, ou sur de simples formats d'archive `.tgz` ...

Formats de Paquetages

- **NB** :Il est aussi possible de télécharger une image ISO, c.-à-d. une copie brute octet par octet de données stockées sur un CD ou sur n'importe quel support.
- Le format le plus courant est le format ISO9660 (JOLIET et HFS) avec les attributs `Rock Ridge Interchange Protocol`.

Les managers de paquetages

Les plus répandus dans le monde Unix ont été mis en place par les principales distibutions Linux :

- Red Hat Package Manager : `rpm` utilisé aussi par Mandrake et SUSE.
- `dpkg` - distribution Debian - et son extension `apt-get`
- `pkg` pour NetBSD et `fink` pour le système Mac OS X.
- Il existe des commandes pour *“depackager”*.

  Exemple :

```
rpm2cpio apt-0.5.5cnc6-0.fdr.8.rh90.i386.rpm | cpio -i -m -d
dpgk -X apt-0.5.5cnc6-0.fdr.8.rh90.i386.rpm
```

Les paquetages RPM

- la commande `rpm -option nomdupackage.rpm` permet d'installer, mettre à jour, désinstaller, vérifier la présence d'un paquetage, mettre à jour la base ...

Les paquetages RPM

- la commande `rpm -option nomdupackage.rpm` permet d'installer, mettre à jour, désinstaller, vérifier la présence d'un paquetage, mettre à jour la base ...
- Le nom du package suit les règles suivantes : `soft.architecture(ou src).rpm`
- le fichier `/var/log/rpmpkgs` contient des informations sur les packages que vous avez installés.
- Le répertoire `/usr/lib/rpm` contient les scripts et les fichiers de configuration pour `rpm`.

Les paquetages RPM

- `rpmdb` permet d'agir sur la base de données liée au système de packages, `rpmbuild` de construire son propre `rpm`.
- Pour réaliser un paquetage `rpm` vous devez reproduire une organisation très précise de vos fichiers : BUILD/ RPMS/ SOURCES/ SPECS/ SRPMS/
- Un site avec toutes les explications pour construire un `rpm` :

http://qa.mandrakesoft.com/twiki/bin/view/Main/RpmHowTo

Les paquetages DEB

- `dpkg` est un script qui remplit le même rôle que `rpm` pour l'installation/desinstallation de package **debian**.
- `apt-get` est un script qui permet de rechercher des packages locaux ou distants - connexion à un serveur Web, par exemple. `deselect` utilise cet outil et propose une interface graphique de sélection/déselection des paquetages à installer.
- `fink` et `finkCommander` - application pour le système X de Mac - utilise apt-get et les packages Debian.

Structure d'un paquetage

- Entête ou Meta Données :
  - Signature ou numéro *"magique"*
  - Name
  - Version
  - Release
  - Architecture
  - On peut aussi trouver des informations sur les dépendances entre les fichiers, la liste des fichiers ...
- Archive : suivant le type du pacquetage on trouvera les fichiers tar, ziper, etc...

Archivage et Compression

- Création d'un fichier d'archive :
  - Respect de l'architecture et des chemins relatifs
  - La commande `tar` permet de désigner aussi bien un fichier qu'un périphérique comme destination:

```
tar -cvf /dev/st0 le-fichier-a-sauver
tar -xvf /dev/st0
```

Archivage et Compression

- Mise en librairie :
  - La commande `ar` vous permet de créer un fichier de *librairie* qui pourra être chargé dynamiquement (ou non) par les différents programmes C, C++, Fortran etc..
  - `ar -r` permet de créer ce type d'archive ou d'ajouter un fichier à une librairie existante :

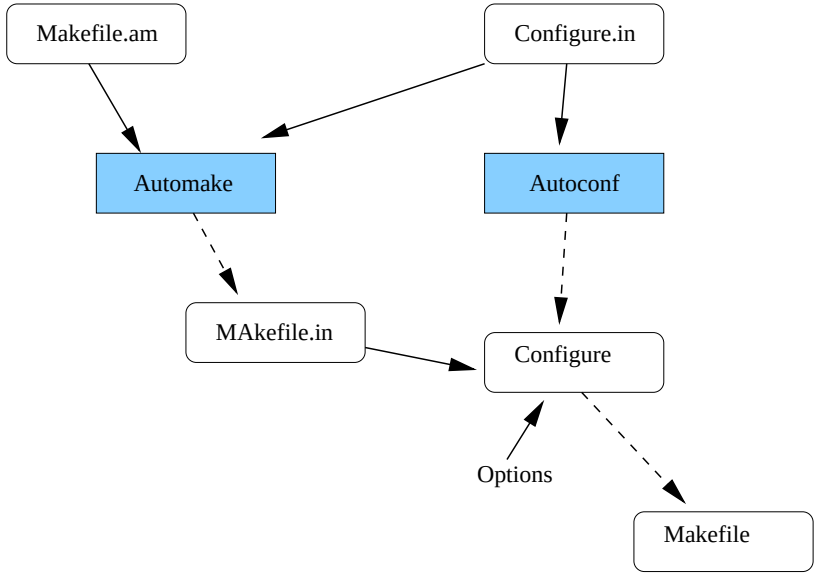
    ```
    ar -r libarbre.a arbre.o toto.o
    ```
  - Vous pouvez lister (-t) le contenu d'une librairie.
  - Les librairies créées par `ar` sont statiques, il existe des librairies dynamiques qui sont créées avec `gcc -shared`

Archivage et Compression

- La commande `gzip` permet de compresser le fichier `.tar` obtenu.
- La commande `bzip2` plus récente est plus efficace mais plus lente.
- les options `z` ou `j` de `tar` permettent d'automatiser le lien archivage/compression. **Avantage** : vous gardez votre fichier `.tgz`

Les outils de configuration

- Objectifs :
  - Déterminer les contraintes d'architecture, les dépendances logiciels et de librairies : compilateur, librairies graphiques, outils de documentation . . .
- Appliquer des scripts pour générer les fichiers `Makefile`.



Configure

- Le fichier `configure.in` contient des macros `autoconf`.
- Le script `configure` est indépendant de `autoconf`. L'utilisateur n'en aura pas besoin.
- Le fichier `configure.status` contient le résultat de la dernièere exécution de `configure`.

Configure

- Les lignes commencent par `dnl` sont des commentaires.
- Le fichier doit toujours commencer par `AC_INIT(monfichier.c)`. Cette ligne sert seulement de test pour vérifier que le script est au bon endroit.
- Il se termine toujours par `AC_OUTPUT(Makefile)`. Il est possible de spécifier plusieurs fichiers `Makefile`, autant que de `Makefile.in`
- `AC_PROG_CC` cherche le compilateur C de la machine.
- `AC_CHECK_LIB(X11, XDisableAccessControl)` teste la présence des librairies citées.

Exemple de macros

```
dnl INSTALLEDJPEG avoids HAVE_LIBJPEG
dnl to appear twice in config.h.in
AC_CHECK_LIB(jpeg, jpeg_set_defaults, INSTALL
if test "$INSTALLEDJPEG" = 1
then
AC_DEFINE(HAVE_LIBJPEG)
LIBS="-ljpeg $LIBS"
READJPEGS=f_readjpg.c
WRJPEGS=f_wrjpg.c
else
        ..............
fi
```

Exemple de macros

```
else
        AC_MSG_CHECKING("for ../jpeg")
        if test -d ../jpeg
        then
            AC_MSG_RESULT("yes")
            AC_DEFINE(HAVE_LIBJPEG)
            LDFLAGS="$LDFLAGS -L../jpeg"
            CPPFLAGS="$CPPFLAGS -I../jpeg"
            LIBS="-ljpeg $LIBS"
            READJPEGS=f_readjpg.c
            WRJPEGS=f_wrjpg.c
        else
            AC_MSG_RESULT("no")
            READJPEGS=
            WRJPEGS=
        fi
fi
```

Make

- C’est une commande qui prend en argument par défaut un fichier dont le nom est `makefile` ou `Makefile`.
- l’option -f permet de spécifier un autre nom de fichier.
- Un `Makefile` est un ensemble de règles correspondant à des tâches à réaliser.
- Chaque tâche a une cible, le plus souvent un fichier à construire.
- L’utilisation la plus courante est la mise en oeuvre de directives de compilation : C, C++, LaTeX etc...

Make

- Forme d’une règle :

```
cible : dependances
[tabulations]commandes
```

Make

- Par défaut la première cible sera exécutée.
- Si cette première cible requiert l'exécution d'autres commandes, celles-ci seront également exécutées.
- Sinon l'exécution de `make` se terminera.

Make

- Exemple :

```
all: dependA dependB
        echo ‘‘ok pour all’’

dependA:
        echo ‘‘ok pour A’’

dependB:
        echo ‘‘ok pour B’’
```

Make

- Les dépendances peuvent être liées à d'autres fichiers:

```
fic.all: fic.c fic.h
	cat fic.h fic.c > fic.all
```

- Une double vérification est effectuée : existance du fichier et date.

Make

- Comme n'importe quel script un `makefile` peut contenir la définition de variables :

```
dataFile=*.dat

data.txt: $(dataFile)
        cat item1.dat item2.dat > data.txt
```

- Attention : **tous les fichiers .dat** du répertoire courant seront pris en compte.
- `$(commande)` est ce qu'on appelle une *wildcard*. c.-à-d. qu'elle force l'interprétation de `*.dat`.

Plus subtile !

- Une version *“simplifiée”* :

```
dataFile=$(wildcard *.dat)

data.txt: $(dataFile)
        cat $? > $@

info:
        echo $(dataFile)
```

- La commande exécutée s’affiche à l’écran.

Make in nutshell

- Des variables particulières :

$@	la cible
$<	la première dépendance
$?	toutes les dépendances plus récentes que la cible
$^	le nom de toutes les dépendances séparées par un espace

Make in nutshell

- Automatisation des règles :

```
fic = intro.pdf cours.pdf

all: $(fic)

$(fic) : %.pdf : %.dvi
        dvipdf $<

%.dvi : %.tex
        latex $<
```

Make in nutshell

- Exécution conditionnelle: structure `if`

```
OS=Linux
JC=javac
JLIB=
all: clean
ifeq ($(OS),Linux)
        $(JC) -classpath linux.jar *.java
else
        $(JC) *.java
endif
        echo $(OS)
```

- Il est possible de changer la valeur de la variable `OS`:

```
make  OS='uname'
```

Make in nutshell

- Les différents tests :

```
ifeq (arg1,arg)2
ifneq (arg1,arg)2
ifdef variable
ifndef variable
```

Makefiles multiples

- Appel et exécution d'une liste de makefiles :

```
all: courant
        (cd a;$(MAKE) all)
        (cd b;$(MAKE) all)
        (cd c;$(MAKE) all)

courant:
        echo "Traiter courant"
```

Gestion de Version

- Besoins : Travail en équipe sur un même projet et historique du développement.
- Partage de fichiers, accès aux modifications
- Possibilités de réaliser ses propres *expériences* sans compromettre la compilation dse autrwes autres participants.
- Pouvoir défaire des modifications ou générer une nouvelle *branche* de développement.

CVS

- CVS : Concurrent version System permet autant de gérer les versions de l'ensemble du projet que de chaque fichier.
- Fonctionne en Client/Serveur : stockage centralisé et accès concurrents aux fichiers.
- On ne stocke pas l'intégarlité des fichiers révisés mais les modifications qui ont eu lieu.
- Possibilité de gérer des autorisations d'accès.

Architecture CVS

- **Repository** désigne le répertoire qui contient l'ensemble du projet.
- **Module** désigne un répertoire du nom de ce module.
- L'ensemble des fichiers et répertoire sont organisés à partir d'un répertoire *racine* qui sera connu grâce à la variable `CVSROOT`.

CVS Serveur

- Créer un répertoire `cvs`
- Initialiser CVS en exécutant la commande `cvs init` dans ce répertoire. Ceci réalise 3 opérations :
  - création du répertoire racine du référentiel
  - initialisation de l'environnement
  - initialisation du dépôt proprement dit.

CVS Serveur

- La commande `init` crée
  - un répertoire du nom de `CVSROOT` dans le répertoire `cvs`
  - une collection de fichiers nécessaires au bon fonctionnement de CVS
- **Remarque** : CVS fonctionne toujours avec 2 versions de chaque fichier : la version courante et la version contenant les information de modifications `toto,v`

CVS Serveur

- Mettre des sources dans CVS :
  - se déplacer dans le répertoires contenant les sources du projet,
  - exécuter la commande :
    ```
    cvs import -m "Initialisation du projet" MonProjet Toto initial
    ```
- Ceci crée le répertoire `MonProjet` dans le répertoire `cvs` au même niveau que le répertoire `CVSROOT`
- Ce répertoire contient maintenant vos fichiers dont les noms sont suffixés par `,v`

CVS - Client

- Pour récupérer des fichiers :
  - Définir la variable `CVSROOT`. Cette variable peut contenir aussi bien un chemin local qu'un chemin distant `machine:lechemin`.
  - Exécuter `cvs checkout MonProjet`
- Ceci aura pour effet de créer un répertoire `MonProjet` contenant les fichiers sources et un répertoire `CVS`.

CVS - Client

- Le répertoire `CVS` du client contient 3 fichiers :
  - `Entries` : liste des fichiers avec leur date de `checkout`
  - `Repository` : nom du module
  - `Root` : valeur de la variable `CVSROOT`

CVS - Client

- Mise à jour du serveur avec des modifications locales :
  - `cvs commit <fichier_modif>`
- Si aucun nom de fichier n'est spécifié, c'est l'ensemble des fichiers qui est mis à jour.
- Ajout d'un commentaire associé aux modifications :
  - `cvs commit -m ''correction de l'erreur de date'' <fichier_modif>`

CVS - Client

- Gestion des conflits de mise à jour :
  - Si plusieurs utilisateurs ont édités en même temps le fichier, cvs enverra un message rapportant les conflits.
  - Un solution simple avec
    `cvs update fichier_modifi`
    cvs réalise alors un merge des modifications
  - En cas de conflits insolubles de façon automatique, l'option `commit` forcera la mise à jour.

CVS - Client

- Ajout de fichiers : `cvs add -m "message" <nom_de_fichier>`
- Suppression de fichiers : `cvs remove <nom_de_fichier>`
- Ces 2 commandes doivent être suivies d'un `commit` pour être prises en compte.
- Toutes les commandes CVS sont accessibles depuis `emacs`

Parenthèses

- Définition d'une `macro(commande)`
  - Procédé d'écriture permettant de définir des opérations composées à partir des instructions du répertoire de base d'une machine donnée, ou à partir du répertoire des commandes d'un programme.
  - Commande formée par une succession d'autres commandes répétitives, et pas forcément très structurée : macros latex, macro emacs macros C

Parenthèses

- SSH et SCP
  - Connexion distante : protocole ssh 1 ou 2

    ```
    ssh -X -l login IPmachine
    ```
  - Copie distante de fichiers : utilise également le protocole ssh 1 ou 2

    ```
    scp  exemple.txt login@IPmachine:chemin
    scp -r login@IPmachine:chemin .
    ```
- La modification des variables de `configure` ou de `make` doit être exprimée en chemin “absolu” - pas de répertoire `.`

Méthodes de modélisation des connaissances

- Structuration des données en fonction des supports :
  - Base de données : SGBD
  - Fichiers : XML
- Langages de description de modèles : Merise, UML.

Méthodes de modélisation des connaissances

- Utilisation de types de données complexes.
- Nécessité de communiquer la composition de ces types :
  - Par le code : dépendance au langage de programmation, communication du source du programme, accès aux interfaces.
  - Par un schéma externe : problème de la maintenance et de la vérification du schéma.

Les Markup Languages

- Les Markup Languages (ML) sont des outils qui permettent d'exprimer des modèles de structuration des données.
- Un ensemble d'outils permet de lier plus ou moins strictement la description et le fichier de données.
- **Remarque** : ceci ne dispense nullement de réaliser le modèle de conception (avec UML ou un autre langage).

Bibliographie

- Comprendre XSLT par Bernd Amann et Philippe Rigaux - O'Reilly
- Schémas XML de Jean-Jacques Thomasson - Eyrolles
- Le site officiel : http://www.w3.org/XML
- Le site français : http://xmlfr.org
- Un tutoriel pour les débutants chez developpez.com : http://xml.developpez.com/cours/
- Tutoriel XPath : http://jerome.developpez.com/xmlxsl/xpath/

Le Langage XML

- XML pour *eXtensible Markup Language*.
- Ancêtre : GML (Generalised Mark-up Language) de Charles Goldfarb, lui- même inspiré par William Tunicliffe (1967) qui décrit le premier la séparation du contenu d'un document de sa présentation.
- En 1986 le standard SGML est établi, 3 ans avant la création du HTML et du Web.
- XML est développé en 1996 par Jon Bosak.
- Historiquement, l'apparition de XML dans la suite de SGML et HTML fait que les présentations de XML et HTML sont souvent liées mais dans les faits, il n'existe pas d'obligation de considérer XML dans le contexte du Web.

Différences entre XML et HTML

- Le XML est un langage descriptif.
- Originaire du SGML, il s'illustre lui aussi à l'aide de balise.
- La différence avec le HTML se situe au niveau de sa capacité à s'auto-structurer dans sa façon de décrire l'information. Alors que ce dernier se contente de formater une information pêle mêle.
- La balise XML décrit l'information qu'elle jalonne alors que le HTML détermine la façon de présenter l'information qu'il balise.

Format XML

- Un fichier XML est balisé par des TAGS ou mots clés qui fonctionnent comme un système de parenthésage mathématique.
- Ces mots clés peuvent être vus comme les éléments lexicaux et l'agencement ou composition que vous réalisez avec ces mots clés définit une sorte d'arbre syntaxique qui permet de reconstruire les instances des objets de votre programme.
- La notion d'arbre est ici essentielle, un document XML possède une racine, des branches de décomposition et des feuilles avec les valeurs des variables.

Les TAGS

- Un TAG est un mot - ou mot clé - encadré par les signes $<$ $>$. L'ensemble des TAGS *"valides"* définit le langage de description.
- Chaque fichier XML doit commencer par un TAG entête :

```
<?xml version="1.0"
      encoding="ISO-8859-1"?>
```

- TAG commentaire :

```
<!--Description du reseau metabolique''-->
```

Les TAGS

- TAG décrivant un noeud de l'arbre :

```
<section> Debut du noeud
</section> Fin du noeud
```

- TAG avec attribut :

```
<Chapitre numero="1">
```

NB : le TAG se referme sans citer l'attribut.

Les TAGS

- Si XML définit lui-même un ensemble de TAGS, il doit surtout être considéré comme un **meta langage**, c'est à dire un langage qui permet d'en définir d'autres.
- Les **attributs** :
  - Chaque TAG peut supporter des spécifications qui sont alors indiqués comme les valeurs des attributs de ces TAGS.
  - Les attributs autorisés pour chaque TAG doivent être précisés lors de la définition du langage.
- Les TAGS et les attributs ne sont pas prédéfinis, ce qui signifie que l'utilisateur est libre de créer les TAGS et les attributs qui lui sont nécessaires.

Définir une arborescence

- La structure d'un document se décompose en un préambule (entête) et un corps.
- Ce corps commence par le TAG racine de l'arbre et se terminera par le même TAG dans sa position *fermée*.
- Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?
<journal nom="Linux  Gogo">
  <!--Description du numero''-->
  <Chapitre numero="1">
    ..................
  </Chapitre>
</journal>
```

Visualiser un fichier XML

- La plupart des navigateurs Web peuvent afficher correctement les données d'un fichier XML sous forme d'arbre.
- **Attention** : à l'heure actuelle le traitement des fichiers XML par les navigateurs n'est pas normalisé. Il est extrêmement fréquent d'obtenir des résultats différents suivant le navigateur utilisé.
- Exemple :

Les Feuilles de Style

- **XSL** (*eXtensible Style sheet Language*) : feuille de style pour XML.
- Objectifs :
  - factoriser les attributs d'affichage,
  - améliorer la lisibilité d'un document,
  - séparer l'aspect graphique du fichier qui contient les données.
  - possibilité d'adapter l'affichage aux périphériques : ecran, imprimante etc
- Dans le principe très proche des CSS de HTML.

XSL

- Standard du W3C - Version 1.0 - basé sur le standard DSSSL (Document Style Semantics and Specification Language), norme ISO 1996.
- XSL est un langage de transformation de document : trier un document, extraire uniquement certaines informations.
- 2 étapes : transformation par XSLT du document XML en un autre document XML, puis mise en forme.
- C’est dans ce 2ième temps que l’on peut choisir le mode de présentation, par exemple HTML.
- La transformation est opérée par un processeur **XSLT**

XSLT - XPath

- Version 1.0 - novembre 1999.
- Les navigateurs implémentent plus ou moins XSl/XSLT.
- Il est nécessaire d'implémenter XPath pour utiliser XSLT.
- XPath est un langage de *pattern matching* qui travaille à partir de l'arbre syntaxique obtenu par la transformation XSLT du fichier..

Syntaxe XSLT

- Balise de déclaration :

```
<?xml version=’’1.0’’encoding=’’ISO-8859-1
 <xsl:stylesheet xmlns:xsl=’’http://www.w3
                              1999/XSL/transform
 <xsl: template match=’’/liste_clubs’’>

</xsl:template>
</xsl:stylesheet>
```

Déclaration de la structure du document

- Le fait que la définition de la structure du document et les données soient placées ensemble dans le fichier XML est un des reproches majeurs qui est fait à XML.
- DTD et Schémas
  - Il est possible de séparer ces deux informations et de spécifier la structure dans un *Document Type Definition* **DTD** ou bien dans un *Schéma* **xsd**.

Caractéristiques des DTD

- Un DTD décrit la structure logique du document.
- C'est un ensemble de règles ou contraintes que tout document qui déclare ce DTD doit respecter afin d'être considéré comme bien formé.
- **Remarque** : un document qui ne déclare pas de DTD est considéré bien formé par défaut.
- La présence d'un DTD permet d'exporter la structure d'un document à l'extérieur de celui-ci.
- Lorsque deux documents déclarent le même DTD il est possible de garantir qu'ils respectent la même syntaxe.

Caractéristiques des DTD

- Déclaration d'un fichier DTD :

```
<!DOCTYPE mesDTD ‘‘mesDTD.dtd’’>
```

- La vérification des règles du DTD se fait au moyen d'un *parser* XML.
- Un parser a un rôle assez semblable à celui d'un compilateur :
  - Analyse syntaxique du programme et signalement des erreurs.
  - Traduction du programme en langage machine - remplacement des entités par leur valeur.
  - Construction de l'arbre syntaxique - mais pas de traitement dans le cas du parser.

Caractéristiques des DTD

- Les DTD ont leur propre langage :
  - Déclarer un élément :

    ```
    <!ELEMENT nom du champs (type du champ
    ```
  - Exemple:

    ```
    <!ELEMENT journal (#PCDATA)>
    ```
  - Fichier XML de données correspondant :

    ```
    <journal> Linux a Gogo </journal>
    ```

Caractéristiques des DTD

- Il est possible de déclarer une liste d'éléments composites comme suit :

```
<!ELEMENT journal (nom, adresse, numero)>
```

- Liste d'attributs :

```
<!ATTLIST champs type_attribut valeur>
<!ATTLIST journal type_journal #PCDATA>
```

- Fichier XML correspondant :

```
<journal type_journal=''mensuel''>
    <nom> Linux  Gogo </nom>
        <adresse> Paris </adresse>
        <numero> 25 </numero>
</journal>
```

Caractéristiques des DTD

- Déclaration des entités : une entité est un type de données qui permet de faire référence à un autre élément du DTD en tant que type de l'élément du TAG.

```
<!ENTITY nom valeur>
<!ENTITY MENS ‘‘Mensuel’’>
<!ATTLIST journal type_journal ENTITY #REQ
```

- Fichier XML correspondant :

```
<journal type_journal=’’MENS’’>
     <nom> Linux a Gogo </nom>
</journal>
```

- Il est possible de déclarer des entités externes c.-à-d. décrites dans un autre fichier.

```
<!ENTITY MENS SYSTEM ‘‘entites.xml’’>
```

Limites et Inconvénients des DTD

- Les DTD sont écrits dans une syntaxe particulière différente de la syntaxe XML.
- Un seul type primitif PCDATA défini.
- Pas de possibilités de poser des contraintes sur les données ; nombre précis d'occurences (seule la forme * ou 0..n existe), format (date, longueur de chaine.
- Depuis mai 2001 le W3C préconise donc de remplacer les DTD par des **schémas** XML.

Les Schémas

- Contrairement aux DTD, les schémas permettent de décrire l'imbrication et l'ordre d'apparition des éléments et de leurs attributs soit une **grammaire** d'un langage particulier.
- Un **schéma** XML est un fichier écrit en XML.
- Si un **schéma** existe le fichier de données devra contenir la déclaration de ce *schéma*.

Exemple :

```
<journal
 xmlns:xsi=
   ’’http://www.w3.org/2001/XMLSchema-instanc
 xsi:noNamespaceSchemaLocation=
                          ’’journal.xsd’’>
 <nom> Linux  Gogo </nom>
</journal>
```

Description d'un Schéma

- Les descriptions utilisent un nouveau TAG `xsd`.
- L'élément racine du *schéma* est de type `xsd:schema`.
- Pour décrire un élément on donne son *nom*, son *type*, et si nécessaire les liens ou contraintes qui s'appliquent.
- Pour décrire un élément on donne son *nom*, son *type*, et si nécessaire les liens ou contraintes qui s'appliquent.

Exemple

```
<xsd:element name=''journal''>
 <xsd:complexType>
   <xsd:all>
     <xsd:element name=''nom''
                  type=''xsd:string''/>
     <xsd:element name=''adresse''
                  type=''xsd:string''/>
     <xsd:element name=''numro''
                  type=''xsd:positiveInteger'
     <xsd:element name=''type_journal''
                  type=''type_j''/>
   </xsd:all>
 </xsd:complexType>
</xsd:element>
```

Exemple

```
<xsd:simpleType name=’’type_j’’/>
  <xsd:enumeration value=’’mensuel’’>
  <xsd:enumeration value=’’hebdomadaire’’/>
  <xsd:enumeration value=’’journalier’’/>
</xsd:simpleType>
```

Utiliser des fichiers XML

- Les fichiers XML ne sont pas destinés à être analysés *“manuellement”*, entre autre à cause de leur verbosité.
- Pour relire un fichier XML, le programme utilisé doit connaitre la syntaxe et les règles du langage XML et s’il existe, le DTD ou le schéma spécifique défini pour l’application.
- Un `parser` est un programme qui sachant un grammaire interprète les différents éléments contenu dans un fichier de données.

XPath

- Un langage différent de XML pour adresser des parties de document.
- Une syntaxe compacte à utiliser dans le XML.
- XPATH opère sur la structure logique du document pas sur la syntaxe.
- La documentation se trouve à `http://xmlfr.org/w3c/TR/xpath/`

XPath - les types d'objets

- Lorsqu'elle est évaluée toute expression XPath prend un des 4 types :
  1. Collections de noeuds : `node-set` - ensemble non-ordonné de noeuds sans doublons
  2. Booléens : `true/false`
  3. Réels : `number`
  4. Chaines de caractères : `string`

XPath - les types d'objets

- Exemple d'utilisation de XPath dans XSLT :
  - `child::*` résultat de type node-set
  - `child::nom=''` résultat de type booléen
  - `number(5 div2)` résultat de type number
  - `string(child::nom)` résultat de type string - valeur du premier noeud fils nom du noeud courant.

Sélection

- Sélection à partir du nom d'un type d'élément :

```
<xsl:value-of select="nom_element"/>
```

- L'utilisation du caractère `/` permet de définir le chemin d'accès
- Utilisation du caractère `*` : exemple `*/paragraph` sélectionne tous les noeuds paragraphes quelque soit leur ascendant.
- L'expression `section//paragraph` permet la recherche de tous les noeuds `paragraph` descendants du noeud `selection` directement descendant du noeud courant.
- Les caractères `.` et `..` sont utilisés comme d'habitude.

Expressions

- Sélection d'attribut :
  - `section[@titre]` sélectionne les éléments `section` qui ont un attribut `titre`.
  - `section[@titre="Introduction"]` sélectionne les éléments section dont l'attribut `titre` a pour valeur `Introduction`.
- Opérateurs arithmétiques : $+ \ - \ * \ Div \ mod$
- Opérateurs logiques :
  - Comparaison : $= <= >= > <$
  - Les opérateurs `and`, `or` et `not` peuvent être utilisés : Exemple de sélection des noeuds `section` n'ayant pas d'attribut `titre`

    `section[not(@titre)]`

Expressions

- Chemin de localisation des noeuds : définition de l'axe, du noeud et de 0 à n prédicats pour spécifier la recherche :
- Quelques **axes** :
  - `child` : contient les enfants directs du noeud contextuel.
  - `descendant` : contient les descendants du noeud contextuel. Un descendant peut être un enfant, un petit-enfant...
  - `parent` : contient le parent du noeud contextuel, s'il y en a un.

Expressions

- Quelques **axes** (suite):
  - `ancestor` : contient les ancêtres du noeud contextuel. Cela comprend son père, le père de son père... Cet axe contient toujours le noeud racine, excepté dans le cas où le noeud contextuel serait lui-même le noeud racine.
  - `attribute` : contient les attributs du noeud contextuel ; l'axe est vide quand le noeud n'est pas un élément.
  - `self` : contient seulement le noeud contextuel.

Expressions

- Quelques **axes** (suite):
  - `preceding` (`following`) : tous les noeuds précédants (suivants) le noeud courant dans l'ordre de déclaration, quelque soit le degré d'éloignement de la racine.
  - `preceding-sibling` (`following-sibling`) : les noeuds frères du noeud courant le précédant (le suivant) dans l'ordre de déclaration.

Expressions

- Quelques exemples :
  - `child::*` sélectionne tous les éléments enfants du noeud contextuel.
  - `child::nom=''` sélectionne tous les éléments enfants dont le nom est vide.
  - `child::text()` : sélectionne tous les noeuds de type texte du noeud contextuel.
  - `attribute::name` : sélectionne tous les attributs `name` du noeud contextuel.

Fonctions usuelles

- `number()` : évalue une expression arithmétique et la convertit en `number`
- `round()` : arrondit un `number`
- `sum()` : retourne un `number` résultat de la somme de toutes les valeurs des noeuds d'un `node-set`.
- `count()` : retourne un `number` dont la valeur est le nombre de noeuds d'un `node-set`
- `position()` : retourne un `number` représentant la position du noeud courant dans le `node-set` courant.
- `last()` : retourne un `number` représentant la position du dernier noeud du `node-set` courant.

Fonctions usuelles

- Traitement des chaines de caractères :
  - `string()` : évalue un objet XPath de n'importe quel type et le convertit en `string`.
  - `contains(arg1,arg2)` : retourne `true` s'il existe une occurence du deuxiemme argument dans le premier, `false` sinon.

Fonctions usuelles

- Sélection des noeuds :
  - `comment ()` : sélectionne tous les noeuds commentaires fils du noeud courant.
  - `text()` : sélectionne tous les noeuds fils du noeud courant, ne contenant que du texte.
  - `node()` : sélectionne tous les noeuds fils du noeud courant.
  - `id(‘‘identifiant’’)` : sélectionne l’élément normalement unique, qui a un attribut de type `ID` valant `"identifiant"`.