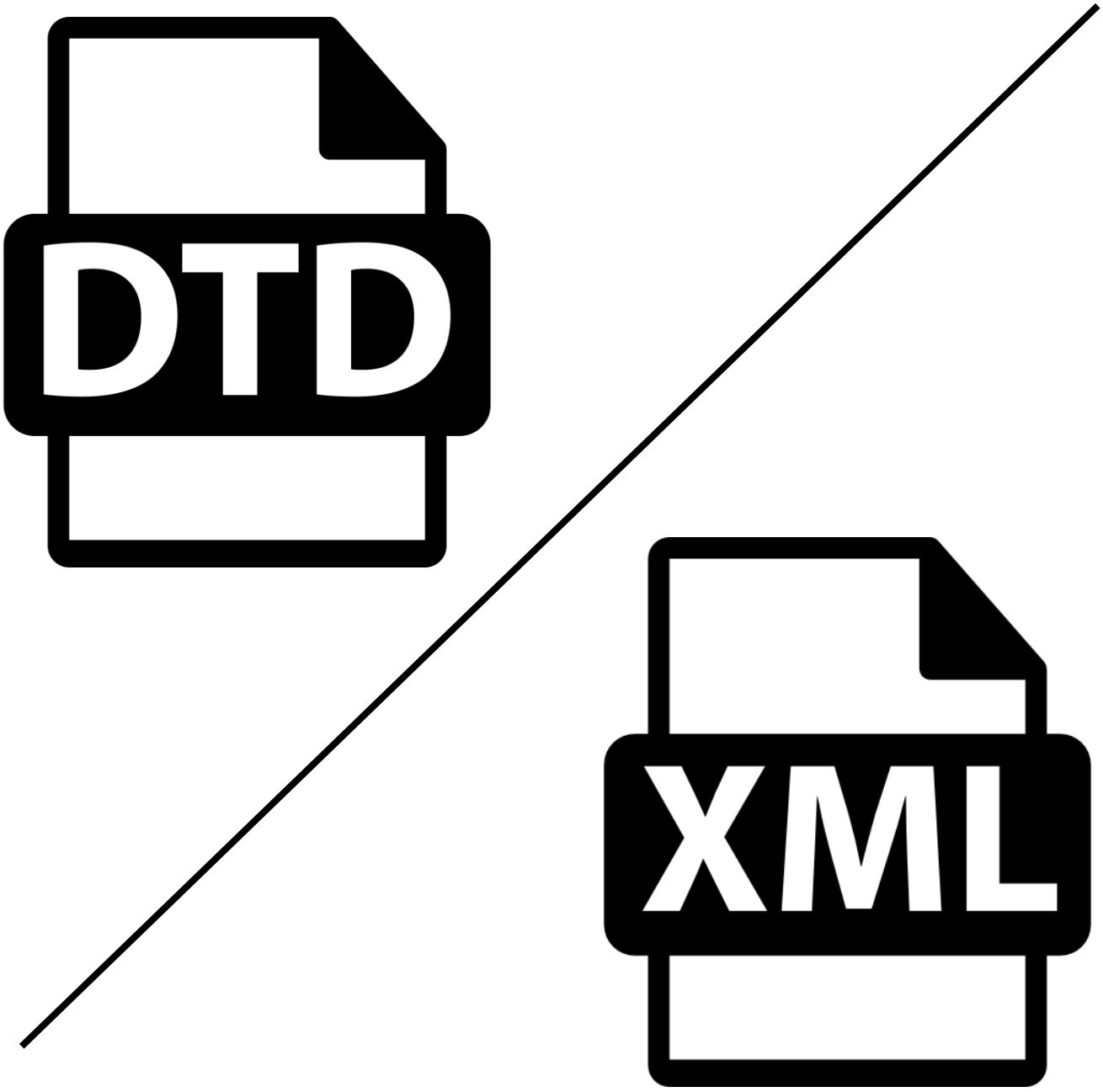
I’m in love with the XML

MENG Henri , GRUMELLON Dylan , ABIB Paul



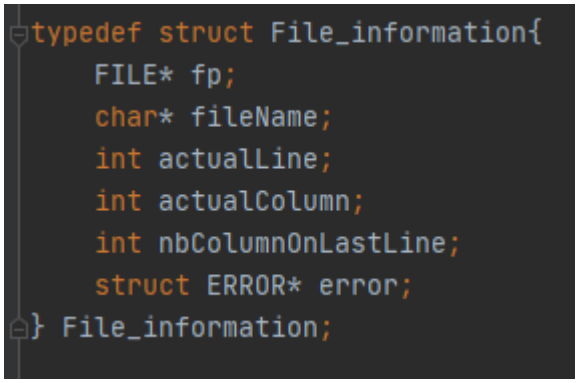
- **Introduction**

L'objectif de ce projet est de construire une application permettant de valider des fichiers XML à l’aide d’une DTD

• Analyse rapide de l'application

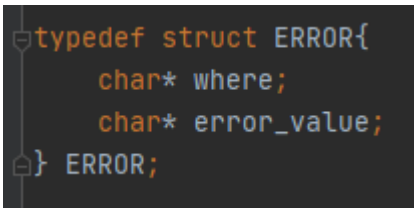
FILE_INFORMATION

- File_information
  - FILE*
  - ERROR*

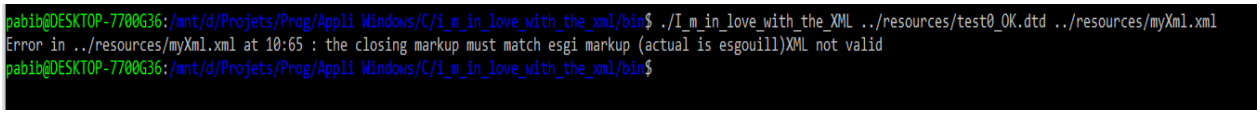
	Structure permettant d'ouvrir et de se deplacer dans un fichier (avec gestion d'erreur). • fileName : nom du fichier • actualLine : line actuellement lu • actualColumn : colone actuellement lu • nbColumnOnLastLine : nombre de colone sur la ligne précedente • ERROR : system d'erreur

ERROR_GESTION

- ERROR

	Structure permettant de créer une erreur.

Nous avons essayé de gérer le plus d'erreur possible, notamment au niveau de la XML. Pour cela, nous avons créer un système permettant de retourner la **ligne**, la **colonne**, et le **problème** d'une erreur et de l'afficher dans la console.



Comment ça marche ? À chaque fois qu'on utilise un fichier nous allons passer par notre structure **File_Information** qui va sauvegarder l'emplacement actuel. Et lorsqu'on détecter une erreur, nous allons appeler notre **createError()** permettant de récupérer l'emplacement, et le message d'erreur à afficher.

READ_XML

Dans cette partie, nous allons lire et examiner les balises XML.

- XML_basic
  - XML_basic**
  - Attribute**

<table>
<tr>
<td>

```
typedef struct XML_basic{
    char* elementName ;
    char* value;
    int valueSize;
    int valueCapacity;
    struct XML_basic** markupList;
    int markupSize;
    int markupCapacity;
    struct Attribute** attributeList;
    int attributeSize;
    int attributeCapacity;
    char* comment;
    int commentSize;
    int commentCapacity;
} XML_basic;
```

</td>
<td>

Structure permettant de stocker une balise, ses balises enfants et ses attributs.

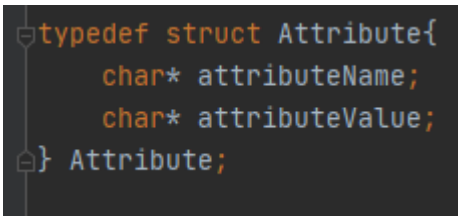
- elementName : nom de la balise
- value : pcdata de la balise
- markupList : liste dans enfants de la balise
- attributeList : liste des attributs de la balise
- comment : commentaire contenu dans la balise

</td>
</tr>
</table>

ReadInsideXml permet de lire a l'interrieur d'une balise xml. Cette fonction est recursive car elle va s'appeller quand elle repere une autre balise. Elle permet de repérer les balise, les commentaires et la pcdata de la balise courante.

READ_ATTRIBUTE

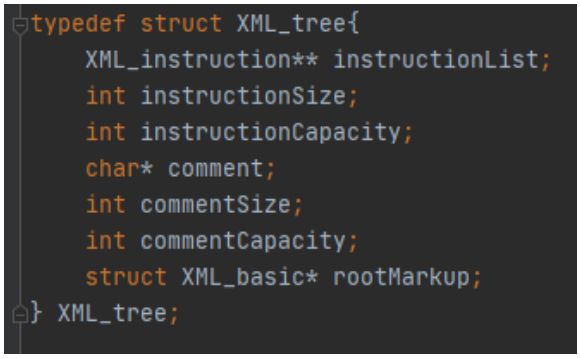
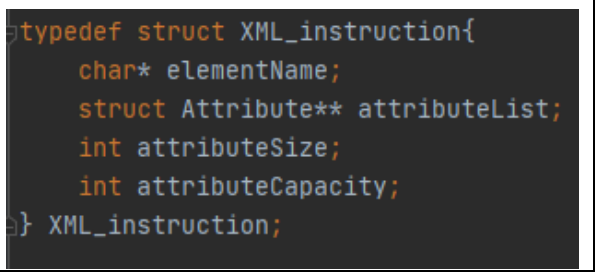
- Attribute

	Structure regroupant le nom et la valeur d'un attribut

XMLFILE_READ

Dans cette partie, nous allons lire et examiner une XML.

- XML_tree
  - XML_instruction**
  - XML_basic*

	Structure permettant de refleter un fichier xml • instructionList : liste des instructions xml (< ?xmlinstruction?>) • comment : commentaire a l'exterieur de la balise root • rootMarkup : la balise root
Structure représentant une instruction xml • elementName : nom de l'instruction • attributeList : liste des attributs	

ReadXml permet de commencer a lire un fichier xml. Elle enregistrer les commentaire, les balise d'instruction puis, quand elle trouve la balise root, commence a lire dedans.

CreateXmlTree renvoi l'alocation d'un XML_tree
ReadInstruction permet de stocker une balise d'instruction.
FreeXml_tree free un xml_tree.
FreeXml_instruction free un xml_instruction
ShowXmlFile permet d'afficher le fichier xml enregister dans le terminal;

READ_DTD

Dans cette partie, nous allons lire et examiner une DTD. Pour cela, on retrouvera 5 structures imbriquées de cette façon :

- markupContainer
  - markup
    - parameter
      - element
      - Attribute

<table>
<tr>
<td>

```
typedef struct markupContainer {
    markup* markupArray;
    unsigned int size;
    unsigned int capacity;
} markupContainer;
```

</td>
<td>

- C'est dans cette structure qu'on va garder toutes les informations par rapport à notre DTD dans la globalité.
- Il possède un **tableau de markups**, suivi de sa **taille** et de sa **capacité**

</td>
</tr>
<tr>
<td>

- Un markup est la représentation d'un élément et/ou d'un attribut
- On y retrouve le **type**, **nom**, **paramètres** de ce dernier

type (ELEMENT, ATTLIST, DOCTYPE)

</td>
<td>

```
typedef struct markup
{
    char* markup_type;  // !DOCTY
    char* markup_name;
    parameter markup_parameters;
} markup;
```

</td>
</tr>
<tr>
<td>

```
typedef struct parameter
{
    char* parameter_type;

    char* category;  // c
    element element;  // 
    attribute attribute;
    char* entity;
} parameter;
```

</td>
<td>

- Les paramètres d'un markup, eux aussi possède un **type** spécifique à celui du markup
- Ils peuvent être une **categorie,** un **ou des élément(s)**, un **attribut,** ou bien une **entité**

category (#PCDATA, #CDATA)

</td>
</tr>
<tr>
<td>

- On est parti du principe qu'un attribut possède un **nom**, un **type**, et une **valeur**

</td>
<td>

```
typedef struct attribute
{
    char* attribute_name;
    char* attribute_type;
    char* attribute_value;
} attribute;
```

</td>
</tr>
</table>

`` typedef struct element { char** elements; unsigned int elements_size; unsigned int elements_capacity; } element; ```	• Pour **element**, on donne la possibilité d'avoir plusieurs enfants, d'où un **tableau** ! • Et bien évidemment, la **taille** et la **capacité** de ce tableau

Maintenant qu'on a toutes les structures, on peut s'attaquer à la logique en elle même.

La méthode principale est **find_dtd_content()**, c'est à partir d'elle qu'on va enclencher notre algorithme. Le but serait de retourner une variable de type **markupContainer** contenant le contenu de notre DTD !

Et sous forme d'une chaîne les autres méthodes découpant notre algorithme vont s'appeler les une après les autres :

```
markupContainer* find_dtd_content(File_information*);
void find_dtd_markups(File_information*, markupContainer*);
void find_dtd_markup(File_information*, markupContainer*);
void retrieve_dtd_info(char*, markupContainer*);
void get_dtd_type(char* str, int* i, markupContainer*);
void get_dtd_name(char* str, int* i, markupContainer*);
void find_typeOf_param(char* str, int* i, markupContainer*);
void get_dtd_param_category(char* str, int* pos, markupContainer* markupArray);
void get_dtd_param_element(char* str, int* pos, markupContainer* markupArray);
void get_dtd_param_attribute(char* str, int* pos, markupContainer* markupArray);
void get_dtd_param_entity(char* str, int* pos, markupContainer* container);
```

1. Chaque méthode a sa propre importance dans la récupération des informations.
2. L'algorithme va lire le fichier contenant notre DTD, ensuite il va détecter notre DTD dans ce fichier grâce au markup **!DOCTYPE.**
3. Après ça, il va pouvoir la manipuler en la parcourant complètement, lors de son parcours il va détecter chaque markup.
4. Chaque markup est directement conservé dans une variable en tant que chaîne de caractères pour y être découpé.
5. La découpe se passe en plusieurs étapes :
    a. Détection du type
    b. Du nom
    c. Et on garde le reste symbolisant les paramètres pour la découper à son tour

    | *Ces informations sont gardées dans notre structure* ***markup***

6. Même procédé pour notre variable contenant le reste.
    a. Détermination du type du contenu (category? element? attribut? entity?)
    b. Selon le cas :
        i. Récupération de la valeur ou des différentes valeurs

ii. Détection et stockage du nom, type, valeur de ce dernier de façon **récursive** s'il y a plusieurs attribut dans un même markup

| *Nous sommes actuellement dans notre structure **parameter***

7. Ainsi de suite jusqu'à que l'algorithme détection la fin de la DTD

GTK_FUNCTION

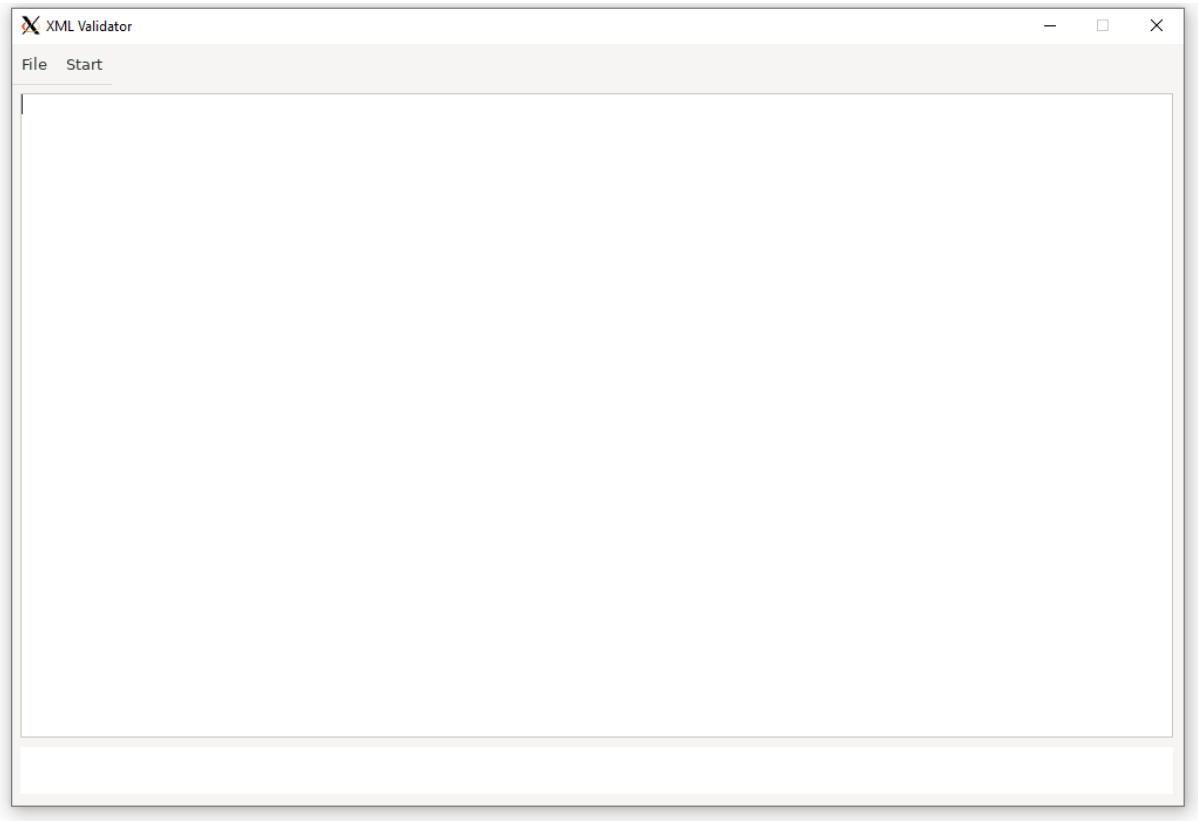
`` typedef struct App{ int exist; GtkWidget *mainWindow; GtkBuilder *builder ; GtkWidget *openDTD; GtkWidget *openXML; GtkWidget *verifXML; GtkWidget *message; GtkWidget *verifXMLwithDTD; struct markupContainer * dtd; struct XML_tree *xml; } App; ```	Structure de l'application graphique.

La partie sur GTK est incomplète cependant elle possède tout de même quelques fonctionnées intéressantes. Il est possible à travers notre fenêtre d'ouvrir un XML et une DTD, et de retourner si notre XML est valide ! On y retrouve une barre de navigation permettant de faire ceci. De plus, 1 champs d'entrée incomplet, et 1 champs de sortie pour les résultats.

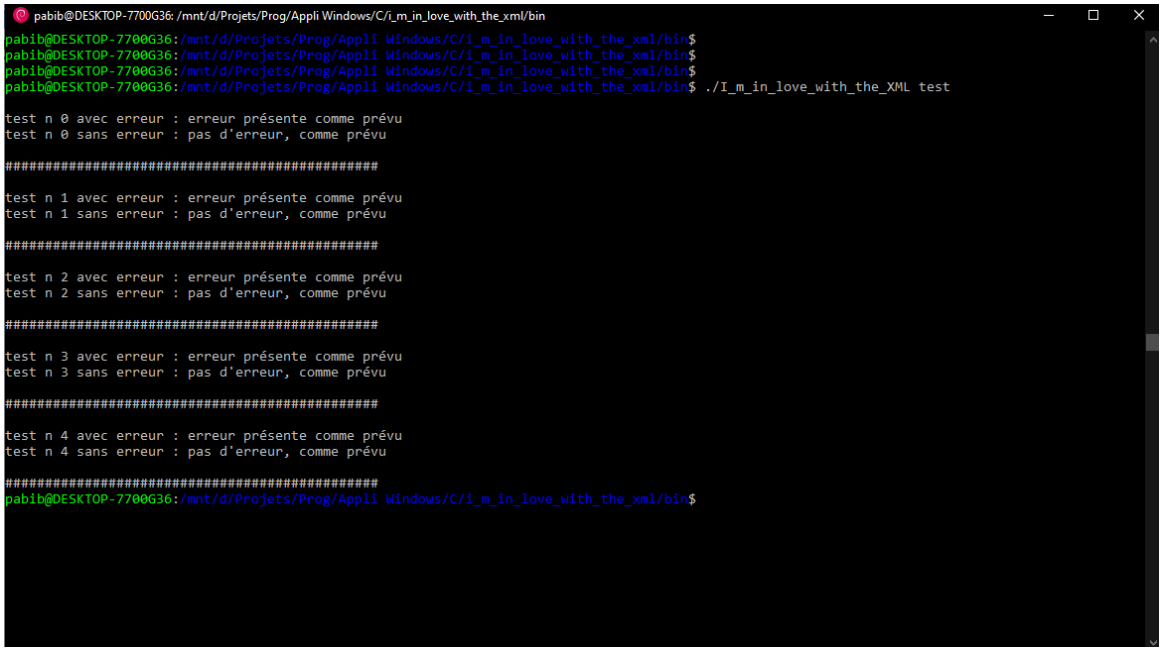
- # Dossier d’utilisation

Le fichier peut se lancer avec des arguments :

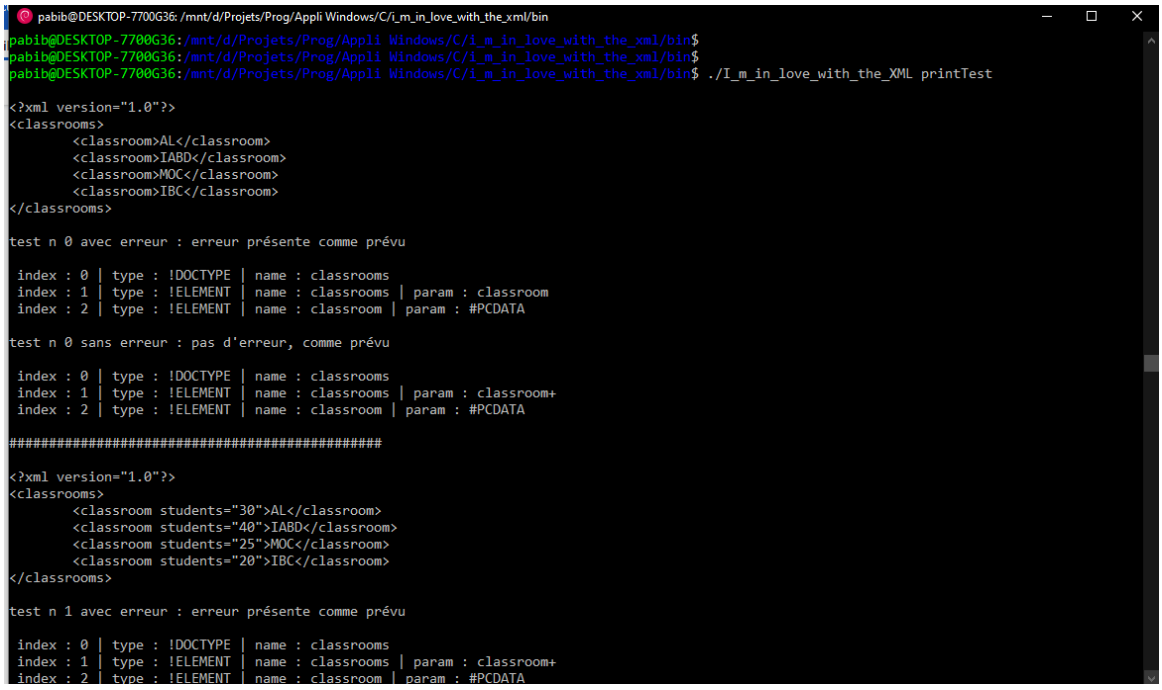
1. Sans argument : lance l’application graphique



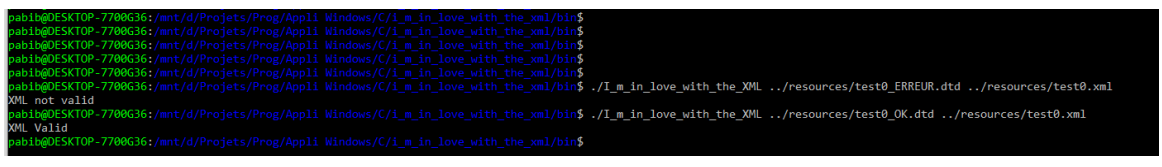
2. Avec l’argument “test” : permet de lancer les tests de base



3. Avec l'argument "printTest" : permet de lancer les tests avec la représentation des fichiers XML et DTD concernés



4. Avec deux arguments : permet de lancer la vérification d'un fichier xml en fonction d'une dtd passé en paramètre



Bilan

Tout au long de ce projet, nous avons fait face à des problèmes, tant au niveau humain que technique, sans oublie d'organisation et matériel.

Cependant, nous sommes tout de même satisfaits du résultat. Notre projet est encore largement perfectible, voici une petite liste non-exhaustif de ce qui aurait pu être amélioré :

- Une gestion d'erreur pour la DTD du même niveau que celle pour la XML
- Un code de meilleure qualité sur certains points
- GTK