Compilateur: LitUml

Rapport de projet

Sol Rosca & Nathan Latino (INF3b)

Janvier 2020

Table des matières

1. Introduction

Le but de ce projet est de créer un language qui permet de décrire un diagramme de classe et d'en générer en SVG.

LitUml est utilisé pour dessiner des diagrammes de classe inspiré par UML en les décrivant textuellement à l'aide d'un langage qui se veut simple à lire et permissif. Cela veut dire qu'il est tout à fait possible de dessiner des diagrammes incohérents avec par exemple deux classes qui héritent respectivement l'une de l'autre. LitUml est plus un outil de dessin qu'un outil de modélisation.

2. Objectifs et fonctionnalités

- Possibilité de déclarer des classes
- Une classe peut posséder des membres répartis en plusieurs catégories:
  - Attributs
  - Méthodes
  - Autres
- Possiblité de styler un élément textuel d'une classe
- Possiblité de faire des relations entre les classes
- Sortie sous forme de SVG

3. Description générale

Le langage de LitUml est interprété de façon recursive. Il lit l'arbre node par node et crée des balises svg en continue. Il continue de s'executé jusqu'à avoir une erreur ou à ce que l'arbre soit entièrement consumé. Cela permet d'avoir une meilleure vision des erreurs qu'il peut y avoir dans le code sans devoir interpréter le fichier dans sa totalité.

- Langage source: LitUml
- Langage destination: SVG

4. Input

```
{
    MaClasse("<<interface>>"),
    [
        "+ attrA: int",
        i"+ attrB: int",
        b"+ cette entré n'a pas de sens mais elle peut être là",
        ib"+ attrC"
    ],
    [
        "+ uneMethode(a: int, b: float): float",
        i"+ uneMethodeAbstraite(a: int, b: float): void",
        ib"+ uneMethodeAbstraiteGrasse(a: int, b: float): void",
    ]
}

{ Batiment(i"<<Abstract>>"), ["private location: String"]}

{ Maison(i"") }

---

Batiment <-- Maison,
Maison <-- MaClasse,
```

Cet exemple est le résultat initialement souhaité. Il n'est malheureusement pas complètement adapté à notre interpréteur. En effet,la version finale ne gère pas les préfixes devant les strings qui servent à moduler le style de la ligne.

5. Description du language

Le fichier doit être séparé en deux parties distinctes avec trois tirets `---` :

5.1. Partie définition

La première partie contient la définition des classes.

- Une classe est délimitée par des acolades et composée de deux types d'éléments séparés par des virgules:
  - L'identificateur, c'est à dire le nom de la classe ainsi qu'un éventuel stéréoptype entre parenthèses.
  - Des blocs délimités par des crochets contenant les membres séparés par des virgules. Ces derniers sont écrits entre guillemets et peuvent être aussi spécifiques que l'utlisateur le souhaite.

Exemple:

```
{
MyClass("Stéréotype"),
    ["attrA","attrB: int",],
    ["aLazyMethod", "+ aBetherOne() : float"]
}
```

5.2. Partie relations

La seconde partie utilise les noms définis dans la première partie pour créer des relations entre classes.

Exemple

```
MyClass <-- MySecondClass
```

6. Logique et arbre

6.1. Analyse lexicale

L'analyseur lexicale permet de convertir des chaînes de caractères en symbole. Le nom des classes sont identifiés et donc pourront être utilisées comme des variables plus tard. Certains caractères sont réservés pour le style, les associations et la sépartion du fichier.

Des lignes de commentaires peuvent être ajoutés en les préfixant `//` sans que celle-ci soit utilisé par l'analyseur syntaxique.

6.2. Analyse Syntaxique

Pour la partie syntaxique, les règles sont données pour toujours avoir une partie **relation** et une partie **classe**.

Les règles pour les classes suivent une logique de possession. Le noeud de la classe définit la variable utilisable par la suite. Les noeuds enfants contiennent soit le Stéréotype, soit ses blocs constituant (attributs, méthodes, ou autre si l'utilisateur le souhaite).

Les relations sont créés pour faire en sorte d'avoir un noeud qui contient le type relation (héritage, association, agrégation, etc.) et ses deux enfants sont les "opérandes" de la relation.

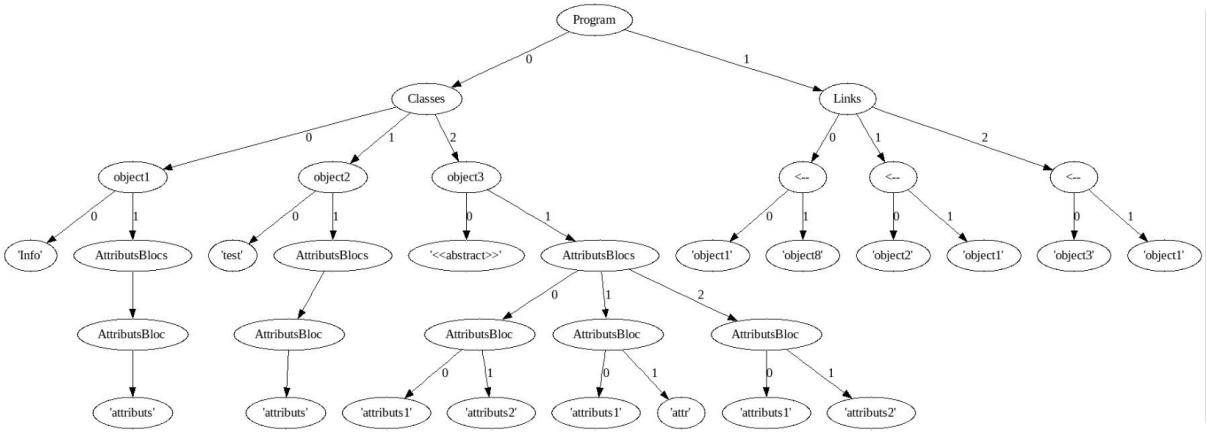

Figure 1 Arbre Syntaxique

6.3. Analyse Sémantique

Pour ce travail, il n'y pas de couture. Il pourrait être intéressant à l'avenir d'avoir un analyseur syntaxique pour empêcher les incohérances vis à vis des "rêgles" d'UML.

6.4. Partie Arrière

L'interpreteur génère un fichier svg pendant la lecture de l'arbre. Une classe SVG permet la création à partir des noms et attributs des classes. Les lignes sont créées pour séparer chaque bloc. Les relations ont une flèche qui détermine le sens de cette dernière et fait en sorte de tracer le lien par dessous les rectangles qui représentent les classes.

7. Utilisation

Les fichiers de démonstration sont contenu dans le dossier /input. Les lignes qui suivent décrivent comment créer un nouveau diagramme:

Déclarons une classe (Il est nécessaire de la flanquer de `("")`):

```
{
    House("")
}
```

Nous pouvons caractériser la classe avec un stéréotype (ou peu importe quoi d'autre entre guillemets):

```
{
    House("<<abstract>>")
}
```

Ajoutons lui un attribut (le format entre guillemets n'a pas d'importance):

```
{
    House("<<abstract>>"),
    [
        "- name : string",
    ]
}
```

Ajoutons quelques methodes (le format entre guillements n'a toujours pas d'importance):

```
{
    House("<<abstract>>"),
    [
        "- name : string",
    ],
    [
        "+ sell(): float",
        "+ clean()"
    ]
}
```

Nous pouvons ajouter autant de blocs qu'l on souhaite:

```
{
    House("<<abstract>>"),
    [
        "- name : string",
    ],
    [
        "+ sell(): float",
        "+ clean()"
    ],
    [ "weird stuff", "some more" ]
}
```

Ok retirons ce truc et ajoutons une nouvelle classe (il n'est pas nécessaire de séparer les classes par une virgule).

```
{
    House("<<abstract>>"),
    [
        "- name : string",
    ],
    [
        "+ sell(): float",
        "+ clean()"
    ]
}
{
    Room(""),
    [
        "- size : double",
    ]
}
```

Pour séparer la déclaration et les relations, ajoutons trois tirets `---`:

```
{
    House("<<abstract>>"),
    [
        "- name : string",
    ],
    [
        "+ sell(): float",
        "+ clean()"
    ]
}
{
    Room(""),
    [
        "- size : double",
    ]
}

---
```

Maintenant nous pouvons décrire les liens entre les classes:

```
{
    House("<<abstract>>"),
    [
        "- name : string",
    ],
    [
        "+ sell(): float",
        "+ clean()"
    ]
}
{
    Room(""),
    [
        "- size : double",
    ]
}

---

House <-- Room
```

- Il ne peut y avoir qu'un délimiteur et il doit être composé de 3 tirets.
- L'ajout d'un commentaire se fait deux slash comme en C.

8. Conclusion

La création d'UML est nécessaire dans tous les projets pour les phases de conception. Le besoin d'avoir un outil qui sépare la présentation de la logique est essentiel pour gagner du temps. Ceci était notre tentative.

Comme annoncé par le responsable de filière en début d'année, la troisième possède un cours de management du temps caché. Il était nécessaire de faire des choix et tous les cours ne pouvent pas être suivi avec l'attention qu'ils méritent. Nous sommes conscients que nous avons fait le minimum syndical mais nous l'avons fait au mieux.

Le projet a les fonctionnalités essentiels implémentées et fonctionne. Il existe beaucoup d'améliorations possibles:

On peut ajouter des rêgles strictes à respecter sous peine de lever une exception ou une erreur, la possiblité d'avoir des relations différentes, un layout svg différent ou encore ajouté des possiblités de personnalisation pour l'utilisateur. Et bien d'autres choses encore...