Hatim M'RABET EL KHOMSSI
Tarek HOUAMED

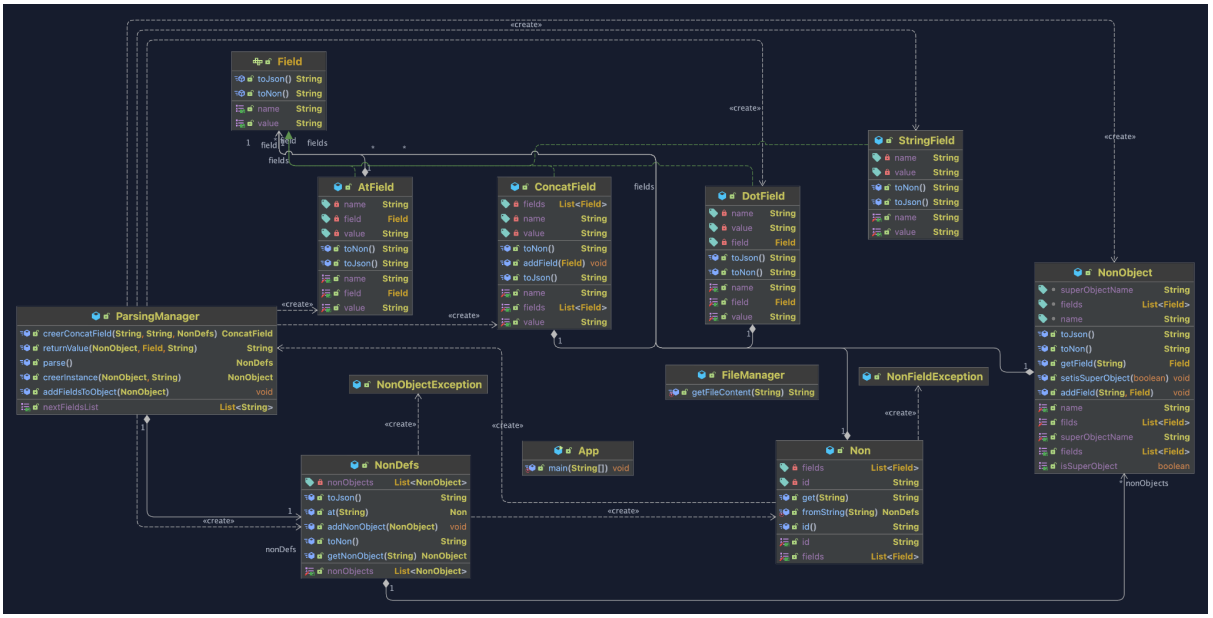
Sujet : Non

1- A partir d'un fichier "config.non" où on a mis l'exemple fourni en TP, on arrive à afficher ,à la fin de l'exécution, alice et bob avec les bonne valeurs pour leurs champs :

```
alice
mail: alice.etu@exemple.tld
name: alice
login: alice
bob
mail: bob.etu@exemple.tld
login: bob
name: robert
```

On a ajouté aussi des tests unitaires pour au lieu d'utiliser le fichier .non, on peut aussi utiliser juste une chaîne de caractères de format Non. Dans ces tests, on a testé tout les cas importants comme si on essayait d'avoir un objet qu'on a pas, il faut déclencher une exception etc…

2- UML



3- En utilisant le format Non qu'on a, on va créer un objet NonDefs, qui va contenir tous les objets avec leurs valeurs.

Pour faire cela, on va d'abord parser le contenu, en utilisant des règles de parsing pour chaque type d'objet, on les a divisés en 2 parties: les objets (les objet de base comme "univ") et les instances (comme "bob" et "alice"), après on va aussi parser les champs qui le suit, pour les objet de base, cela nous permet de savoir quels sont les champs d'un objet, et pour les instance, cela nous permet de changer la valeur d'un champ s'il est modifié après la création de l'instance.

Pour créer un objet, si le parsing de la ligne réussit, on va récupérer le l'id de l'objet et créer un NonObject avec une valeur true pour son attribut isSuperObject qui va nous permettre par la suite de faire la différence entre un objet parent et une instance. Après, on parcourt les champs qui le suivent, chaque ligne est parsé pour savoir la nature du champ, et on ajoute le champ avec le type convenu à la liste des champs de l'objet.
Pour créer une instance, on parse d'abord la ligne et on vérifie que c'est le format d'une instance, après, on récupère l'objet parent, puis on parcourt les champs de l'objet, et en se basant sur le type du champ, on va savoir quelle valeur utiliser ou s'il faut seulement récupérer la valeur qu'on a. Par exemple, si c'est un DotField, on va utiliser la valeur de l'id de l'instance qu'on est entrain de traiter, si c'est un StringField, on va directement prendre son contenu.
Lors de la création des champs, certains champs comme DotField et ConcatField vont avoir un attribut field ou fields où on va mettre le champ de base où il y a l'information non traitée.

Pour la sérialisation de nos objets vers une représentation Json ou Non, l'objet NonDefs contient des fonction toJson() et toNon(), qui vont faire appelle aux mêmes fonctions des NonObject qui vont faire pareils pour leurs champs, et la représentation change pour chaque type de champ.

4- Après avoir un objet NonDefs avec la liste des objets et des instances, on peut maintenant le manipuler comme on veut.
Par exemple on peut récupérer un objet Non si on utilise la fonction ***get***, ou un champ avec la fonction ***at***:

```java
@Test
public void checkValidCase() throws NonFieldException, NonObjectException
{
    NonDefs nonDefs = Non.fromString(content: "univ:\n.name 'Universite Exemple'\n.domain 'exemple.tld'");
    assertTrue( nonDefs.getNonObjects().size() == 1 );
    assertEquals("exemple.tld", nonDefs.at(id: "univ").get(champ: "domain"));
}
```

et qui va lancer une exception s'il ne trouve pas l'objet ou le champs cherché:

```java
@Test(expected = NonObjectException.class)
public void shouldThrowNonObjectException() throws NonObjectException
{
    NonDefs nonDefs = Non.fromString(content: "univ:\n.name 'Universite Exemple'\n.domain 'exemple.tld'");
    nonDefs.at(id: "test3");
}
```

```java
@Test(expected = NonFieldException.class)
public void shouldThrowNonFieldException() throws NonFieldException, NonObjectException
{
    NonDefs nonDefs = Non.fromString(content: "univ:\n.name 'Universite Exemple'\n.domain 'exemple.tld'");
    nonDefs.at(id: "univ").get(champ: "test3");
}
```

On peut aussi sérialiser l'objet NonDefs vers une représentation Json ou Non:

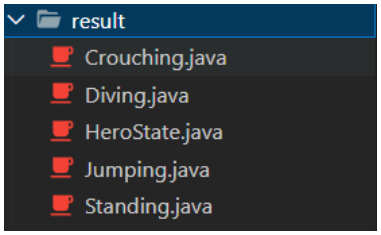
```
{
"univ":{"name":"Universite Exemple","domain":"exemple.tld"},
"student":{"mail":".login '.etu@' univ.domain","name":".login","login":"@"},
"alice":{"mail":"alice.etu@exemple.tld","name":"alice","login":"alice"},
"bob":{"mail":"bob.etu@exemple.tld","login":"bob","name":"robert"},
}
```

```
univ:
.name 'Universite Exemple'
.domain 'exemple.tld'
student:
.mail .login '.etu@' univ.domain
.name .login
.login @
alice: student
.mail 'alice.etu@exemple.tld'
.name 'alice'
.login 'alice'
bob: student
.mail 'bob.etu@exemple.tld'
.login 'bob'
.name 'robert'
```

Hatim M’RABET EL KHOMSSI
Tarek HOUAMED

Sujet : Graphe

1- En utilisant l’exemple que vous nous avez donné, on le parse et à partir du résultat on va créer des classes Java et l’interface HeroState, voici le résultat qu’on a eu aprés l’execution:




```java
package univ.lmd.result;

public interface HeroState {

    public HeroState up();

    public HeroState down();

    public HeroState release();

    public HeroState next();
}
```

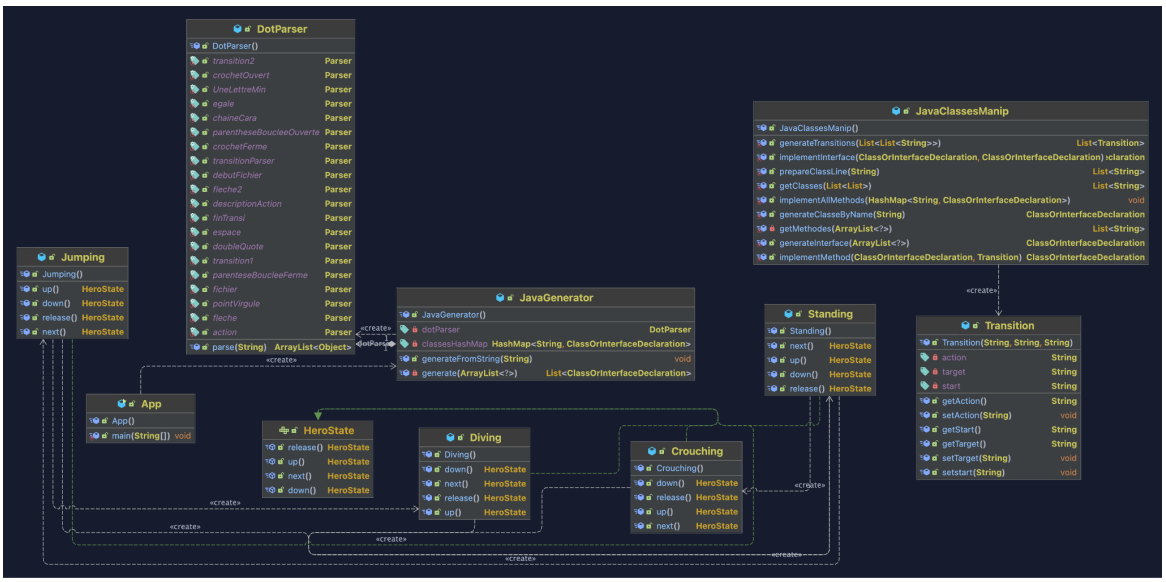
```java
package univ.lmd.result;

public class Diving implements HeroState {

    public HeroState up() {
        return this;
    }

    public HeroState down() {
        return this;
    }

    public HeroState release() {
        return this;
    }

    public HeroState next() {
        return new Standing();
    }
}
```

```java
package univ.lmd.result;

public class Jumping implements HeroState {

    public HeroState up() {
        return this;
    }

    public HeroState down() {
        return new Diving();
    }

    public HeroState release() {
        return this;
    }

    public HeroState next() {
        return new Standing();
    }
}
```

```java
package univ.lmd.result;

public class Standing implements HeroState {

    public HeroState up() {
        return new Jumping();
    }

    public HeroState down() {
        return new Crouching();
    }

    public HeroState release() {
        return this;
    }

    public HeroState next() {
        return this;
    }
}
```

```java
package univ.lmd.result;

public class Crouching implements HeroState {

    public HeroState up() {
        return this;
    }

    public HeroState down() {
        return this;
    }

    public HeroState release() {
        return new Standing();
    }

    public HeroState next() {
        return this;
    }
}
```

2- UML



3- Notre code va d'abord, en utilisant la chaîne de caractère fourni dans le TP, parser la chaise pour pouvoir extraire le nom de l'interface, puis il va récupérer pour chaque état initial, son état final et la transition qui permet d'arriver à cet état.

```
public static void main( String[] args ) throws Exception
{
    String text = "digraph HeroState {\n" +
    "Standing -> Jumping [ label = \"up\" ];\n" +
    "Jumping -> Diving [ label = \"down\" ];\n" +
    "{Jumping, Diving} -> Standing;\n" +
    "Standing -> Crouching [ label = \"down\" ];\n" +
    "Crouching -> Standing [ label = \"release\" ];\n" +
            "}";
    JavaGenerator generator = new JavaGenerator();
    generator.generateFromString(text);
```

A partir du résultat du parsing, on aura une liste d'objets, qu'on va utiliser par la suite, pour générer l'interface de base, les transitions et les classes qu'on va utiliser.
Par la suite, on va implémenter ces classes, en se basant sur le résultat des transitions pour savoir le contenu de chaque méthode, et avec quoi on doit l'implémenter. à la fin de notre code, et lors de la première version version, on a remarqué que certains méthodes comme up() et down() de crouching n'avaient pas de corps, du coup, on a ajouté une autre fonction qui va implémenter toutes les méthodes vides, par un ***return this.***
La fonction ***generate*** nous permet d'avoir les classes et l'interface à partir de la liste d'objets (résultat du parsing), qu'on va par la suite écrire chacun dans un fichier à part, dans le package "***result***".

```
public void generateFromString(String text) throws Exception {
    ArrayList<Object> list = dotParser.parse(text);
    String packageLine = "package univ.lmd.result;\n\n";
    List<ClassOrInterfaceDeclaration> classesAndInterface = generate(list);
    for(ClassOrInterfaceDeclaration c : classesAndInterface){
        File tmp = new File(parent: "./src/main/java/univ/lmd/result/", c.getName() + ".java");
        tmp.createNewFile();
        FileWriter myWriter = new FileWriter(tmp.getAbsolutePath());
        myWriter.write(packageLine + c.toString());
        myWriter.close();
    }
}
```

ce qui nous permet d'avoir le résultat que vous avez vu en haut.

4- Lors de la génération de notre code, nous avons la possibilité de l'utiliser sans avoir à modifier ce code, l'une des limitations constatée est le fait que ce fameux code peut être difficile à maintenir dès lors que le développeur choisit de modifier une partie générée, ceci est dû au fait qu'à chaque régénération de code, sa partie sera effacé et il devra donc recommencer à chaque fois.
Un autre problème possible est qu'un code généré soit fonctionnel sur une machine ne le sera pas forcément sur une autre.
L'une des alternatives possible serait de limiter la régénération de code uniquement sur les fragments de code non modifiés par un humain à l'aide de commentaires qui bloqueraient la

partie encadré.Il faudrait aussi générer des tests pour le code généré afin de vérifier que celui-ci est toujours fonctionnel malgré les modifications.