On veut représenter dans un langage objet des programmes écrits dans un langage imaginaire travaillant sur les entiers uniquement. Un **programme** est une séquence de **déclarations** de variables (chacune avec une expression d'initialisation), suivie d'une séquence d'instructions comprise entre `begin` et `end`. Dans un premier temps, la seule **instruction** est l'affectation, qui affecte à une variable la valeur de l'évaluation d'une **expression**. Les expressions peuvent comporter:

- des variables
- des constants littérales entières
- les quatre opérateurs arithmétiques binaires usuels
- une expression conditionnelle *si_alors_sinon* qui prend 3 sous-expressions. À l'exécution, si la valeur de la première sous-expression est différente de `0`, on exécute la partie *alors* et on renvoie son résultat, sinon on renvoie le résultat de la partie *sinon*.

On doit pouvoir ultérieurement ajouter de nouvelles sortes d'instructions ou d'expressions.

On veut représenter un programme par une instance d'une classe d'une hiérarchie à_ définir.
Exemple: Le programme ci-dessous comporte deux déclarations et deux affectations.

```
X := 12; y:= 0; begin y := si 1 alors 3 sinon x; x :=y; end
```

et pourrait être représenté symboliquement par l'ensemble d'instances suivantes :

```
Var d1 = new Var("x", new Cste(12));
Var d2 = new Var("y", new Cste(0));
Exp e3 = new ITE(new Cste(1), new Cste(3), new Acces(d1));
Inst i1 = new Affectation(d2, e3);
Inst i2 = new Affectation(d1, new Acces(d2));
Programme prog = …                    // à définir selon votre hiérarchie
```

où `Inst`, `Exp`, `ITE`, etc,sont des classes à définir. La classe `Acces` représente l'accès à une variable, sous la forme d'une référence à la déclaration qui a introduit cette variable.

1. Définir une hiérarchie de classes UML[1] avec l'ensemble des classes nécessaires, dont une classe `Programme`. Précisez les cardinalités des associations, les attributs avec leur type, les classes et méthodes abstraites, les constructeurs. On doit pouvoir représenter des expressions et instructions quelconques du langage.

On suppose que le programme source ne comporte pas d'erreur et qu'un analyseur permet de passer du texte du programme à une instance de la classe `Programme`.

2. Définir une méthode `int eval()` d'évaluation d'une expression et une méthode `void exec()`sur les instructions. Donnez le code de **toutes** les méthodes concernées.

3. On considère deux nouvelles instructions : l'instruction *`skip`*, qui ne fait rien, et la boucle *`while (expr) instruction`*, dont le corps contient une unique instruction. Adapter votre hiérarchie et définir la méthode `exec`.

4. Une expression est dite « statique » si sa valeur est connue à la compilation. Par exemple une expression arithmétique est statique si chacun de ses deux opérandes est soit une constante, soit une expression elle-même statique. Définir une méthode `boolean estStatique()`sur les expressions qui retourne `true` si et seulement si une expression est statique.

5. On veut définir une méthode `Exp simplifie()`qui renvoie, si possible, une version simplifiée de l'expression à laquelle elle est appliquée, et l'expression elle-même sinon. La version simplifiée d'une expression statique est sa valeur ; pour la multiplication, on pourrait définir par exemple que `1*e` se simplifie en `e` et `0*e` se simplifie en `0` pour toute expression `e`. Pour la conditionnelle, la simplification de l'expression `e3` de l'exemple serait la représentation de la constante `3`.

Enrichir votre hiérarchie de classes pour associer à chaque sorte d'expression une **liste ordonnée de méthodes de simplifications** et écrire une méthode `simplifie` qui simplifie au maximum une expression selon ces méthodes. Illustrez votre solution en écrivant le code d'un simplificateur.

1 En UML, `{ordered}` permet d'indiquer qu'une association à valeurs dans une collection est ordonnée.