Informatique de Base 3

Projet ADPO

Sommaire

1 Étape 1

Dans cette première étape, nous avons mis en place l'analyse lexicale et l'analyse syntaxique d'une formule ainsi que la mise sous forme d'arbre abstrait et la négation de celle ci de celle ci.

1.1 AnalyseLexicale et AnalyseSyntaxique

Pour mettre ne place l'analyse, nous avons créer une classe Lexeme permettant d'identifier les différents symboles analysés par l'analyseur lexical et les utiliser dans l'analyseur syntaxique.

Nous avons créer une table d'analyse permettant de retourner un identifiant (entier) unique pour chaque lexeme identifié par l'analyseur syntaxique (et vice-versa).

Dans la classe Lexeme, nous avons implémentés différentes méthodes pour savoir si une chaine est une constante, une fonction ou une variable en nous appuyant sur des expressions régulières.

L'analyse lexicale est faite grâce à un automate à état fini.

L'analyse syntaxique est faite grâce à un analyseur syntaxique descendant récursif.

La chaine de caractère représentant la formule à analyser est stockée dans la classe d'analyse lexicale (pas encore de lecture à partir d'un fichier).

Table des symboles ayant servi dans l'implémentation de l'analyseur syntaxique:

	OU	*ET*	*NON*	*(*	*)*	$r \in \Re$	*VRAI*	*FAUX*	*$*
Formule	x	x	1	1	x	1	1	1	x
Disjonction	x	x	2	2	x	2	2	2	x
DisjonctionD	3	x	x	x	x	x	x	x	4
Conjonction	x	x	5	x	x	5	5	5	x
ConjonctionD	7	6	x	x	7	x	x	x	7
Facteur	x	x	8	9	x	9	9	9	x
FormuleQ	x	x	x	11	x	10	10	10	x
Atome	x	x	x	x	x	12	13	14	x
ListeTermes	x	x	x	x	x	x	x	x	x

1.2 L'arbre abstrait

Nous avons créer une classe pour chaque élément composant une formule (classe Ou, Et, Implique, ...).

L'arbre abstrait est généré durant l'analyse syntaxique, certaines méthodes ont du être modifiées (ajout d'un retour et de paramètres) pour le permettre.

L'appel à l'analyseur syntaxique nous retourne donc un objet représentant la formule sous forme d'arbre.

Les éléments ont été groupés par types (Binaire, Unaire, ...) en utilisant l'héritage, tous les éléments implémentent l'interface Formule.

On a aussi implémenté une première version de la classe Atome (ne contenant pas de liste de Termes pour l'instant), ils possèdent un attribut permettant d'indiquer si l'atome est positif ou négatif (nié).

La descente des négations a été mise en place en utilisant une méthode récursive qui retourne un nouvel objet correspondant à la formule niée.

Prototype de la méthode dans l'interface Formule :

```
public Formule nier(boolean val);
```

Exemple de méthode de négation (issu de la classe Et) :

```
public Formule nier(boolean val) {
        if (val)
                        return new Ou(fg.nier(true),fd.nier(true));
        else
                        return new Et(fg.nier(false),fd.nier(false));
}
```

1.3 Clause, FormeClausale et ClauseResolution

Nous avons mis en place la classe Clause contenant deux listes, une d'atomes positifs et l'autre d'atomes négatifs (dont la Clause est la disjonction).

Comme indiqué dans le sujet, on ne considère pour le moment que des clauses (pas d'implique, ...).

Nous avons implémenté une classe FormeClausale qui contient principalement une liste de Clauses (la FormeClausale représente donc la conjonction de celles-ci).

Pour le moment la création des Clause et de la FormeClausale est manuelle (pas encore de mise sous forme clausale «automatique» d'une formule) et leur principale utilité est le fonctionnement de la classe ClauseResolution.

La classe ClauseResolution permet de faire entrer une formule sous forme clausale en résolution en utilisant le système de résolution basé sur les clauses vu en cours :

$$F=(Cl(P),\varnothing,res),\ res=\frac{c1\vee r\vee c2\ ;\ c3\vee\neg r\vee c4}{c1\vee c2\vee c3\vee c4}$$

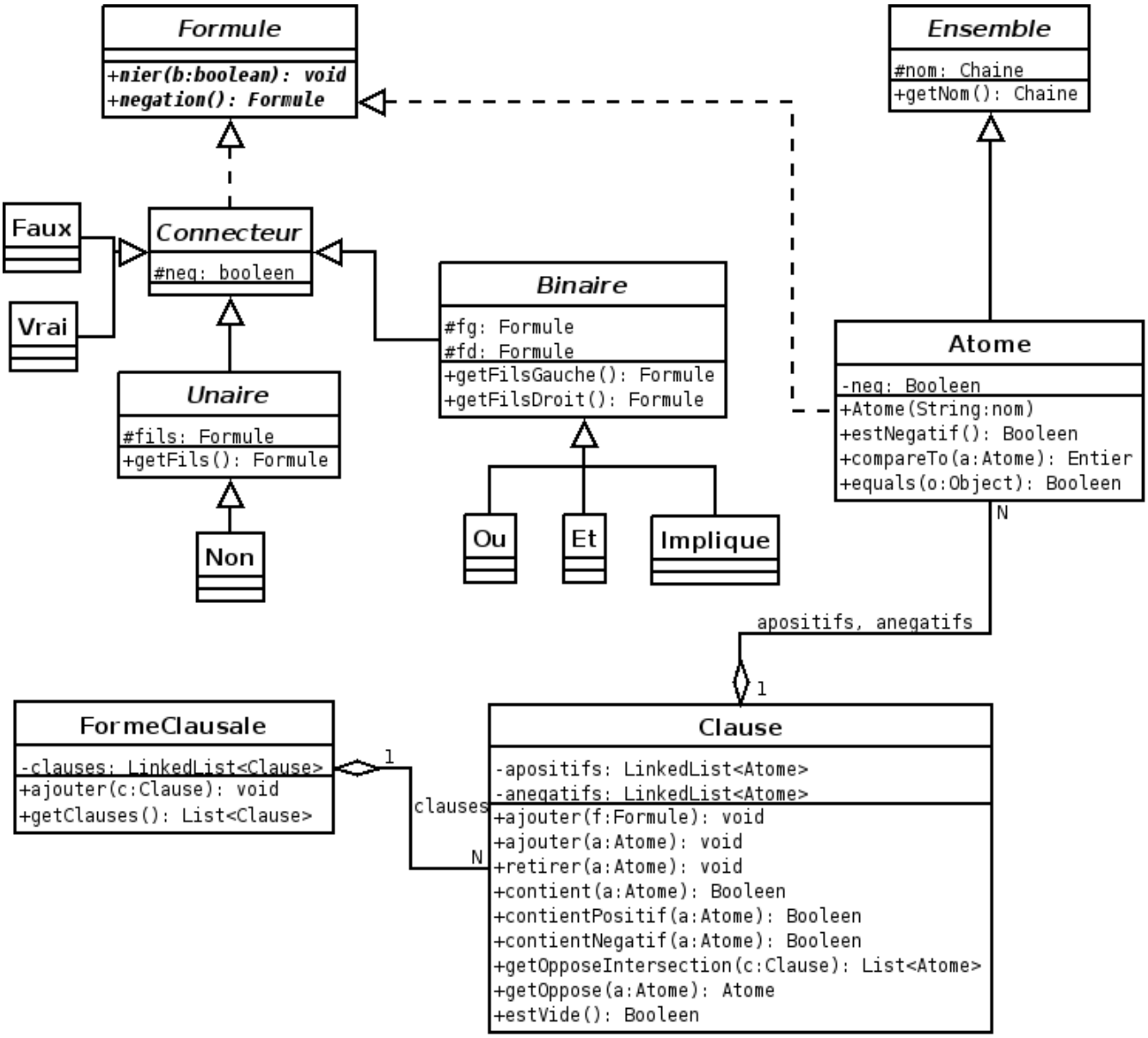
Nous avons aussi créer une classe abstraite Résolution comprenant des méthodes qui pourront nous être utiles lors de l'implémentation du système de résolution de Robinson (lire un choix au clavier, afficher les clauses, ...).

Le système est fonctionnel et utilisable à travers la classe de test.

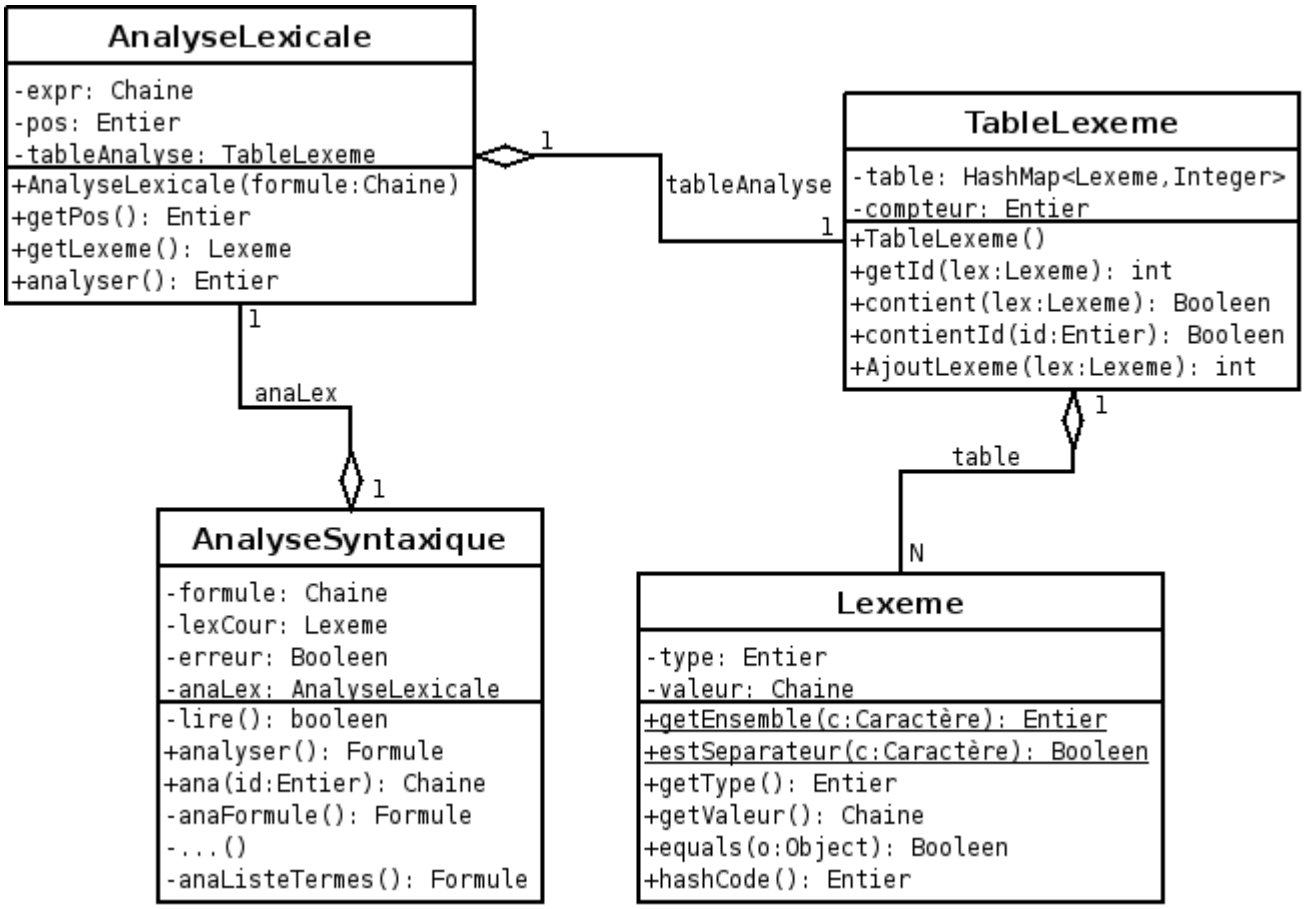
1.4 Résumé

Schéma UML des classes :

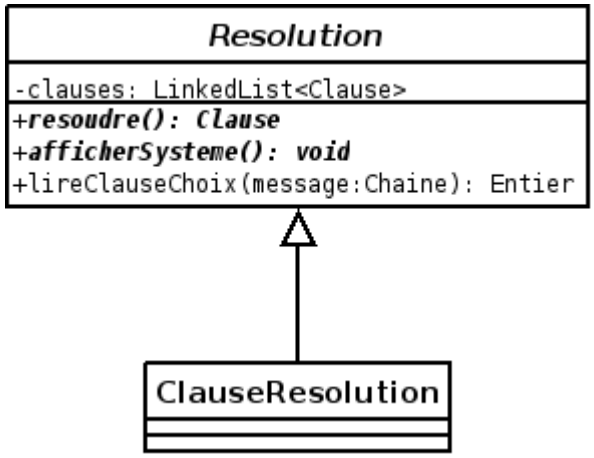
- adpo.formule :



- adpo.analyse :



- adpo.resolution :



Classes importantes:

- Formule
- Clause
- ClauseResolution

2 Étape 2

2.1 Mise sous forme clausale

Pour la mise sous forme clausale, nous nous sommes appuyés sur la classe FormeClausale déjà implémentée.

Nous avons ajouté une interface *OperateurClausal* pour permettre la mise sous forme clausale d'une formule. Cette interface contient une méthode qui se charge de faire remonter récursivement la forme clausale d'une formule en établissant des compositions entre les différentes clauses.

Prototype de la méthode :

```
public FormeClausale formeClausale();
```

L'interface est implémentée par la classe *Et* et par la classe Ou.

Pour mettre une formule sous forme clausale, on peut utiliser le constructeur de *FormeClausale* prenant en paramètre une *Formule* qui fait lui même appel à la méthode *FormeClausale.ajouter (Formule)*. La fonction *ajouter* se charge d'appeller la méthode *formeClausale()* des éléments de la formule.

Remarque: La mise sous forme clausale doit impérativement être faite après la simplification et la négation de la formule.

2.2 Simplification d'une formule

Le but de la simplification est d'alléger les expressions composées des éléments Vrai et Faux.

Elle est faite à travers une méthode récursive ajoutée à l'interface Formule :

```
public Formule simplification();
```

Chaque élément d'une formule retourne une éventuelle mise à jour en fonction de la valeur de son, ses fil(s).

Exemple d'implémentation dans la classe Implique:

```
public Formule simplification() {
	Formule fgs = fg.simplification(), fds = fd.simplification();

	if ((fgs instanceof Faux) || (fds instanceof Vrai))
			return new Vrai();
	if (fgs instanceof Vrai)
			return fds;
	if (fds instanceof Faux)
			return fgs;

	return new Implique(fgs,fds);
}
```

Remarque: la simplification doit impérativement être effectuée avant la négation de la formule.

2.3 Négation d'une formule

La négation d'une formule est faite en deux étapes : la descente des négation puis la création de la formule niée.

2.3.1 Descente des négations

La descente des négations est faite par une méthode récursive, la méthode *nier* de l'interface *Formule* :

```
	public void nier(boolean b);
```

Exemple d'implémentation dans la classe Implique :

```
public void nier(boolean b) {
	this.neg = b;
	if (b) {
			fg.nier(false);
			fd.nier(true);
	} else {
			fg.nier(true);
			fd.nier(false);
	}
}
```

2.3.2 Création de la Formule niée

La création de la formule niée est faite grâce à la méthode récursive *negation* de l'interface *Formule*.

Elle permet de transformer les opérateurs qui ont été niés dans la formule.

Prototype de la méthode :

```
public Formule negation();
```

Exemple d'implémentation dans la classe Et :

```
public Formule negation() {
	if (neg)
			return new Ou(fg.negation(),fd.negation());
	else
			return new Et(fg.negation(),fd.negation());
}
```

2.4 Tests

Nous avons implémenté plusieurs classes de tests pour les différentes parties du programme sur des formules.

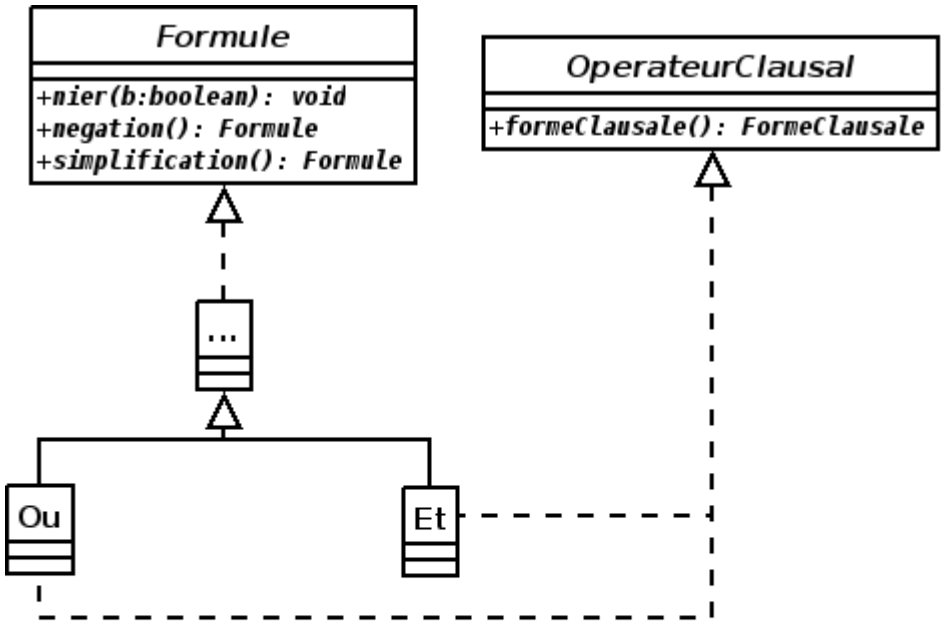
Pour l'instant nous disposons de classes permettant de tester l'analyse, la mise sous forme clausale et la résolution. Les classes de tests utilisent une méthode statique commune (Tests.anaArgs) pour analyser les arguments ([-f <fichier>][-e <formule>]).

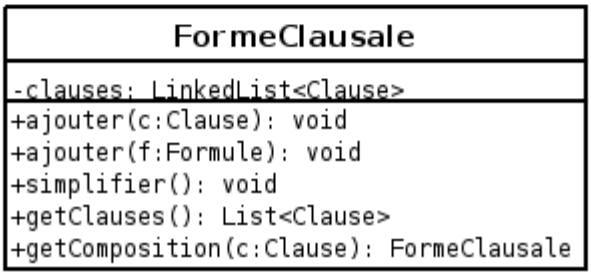
Ces classes sont regroupées dans le package *adpo.test* .

2.5 Résumé

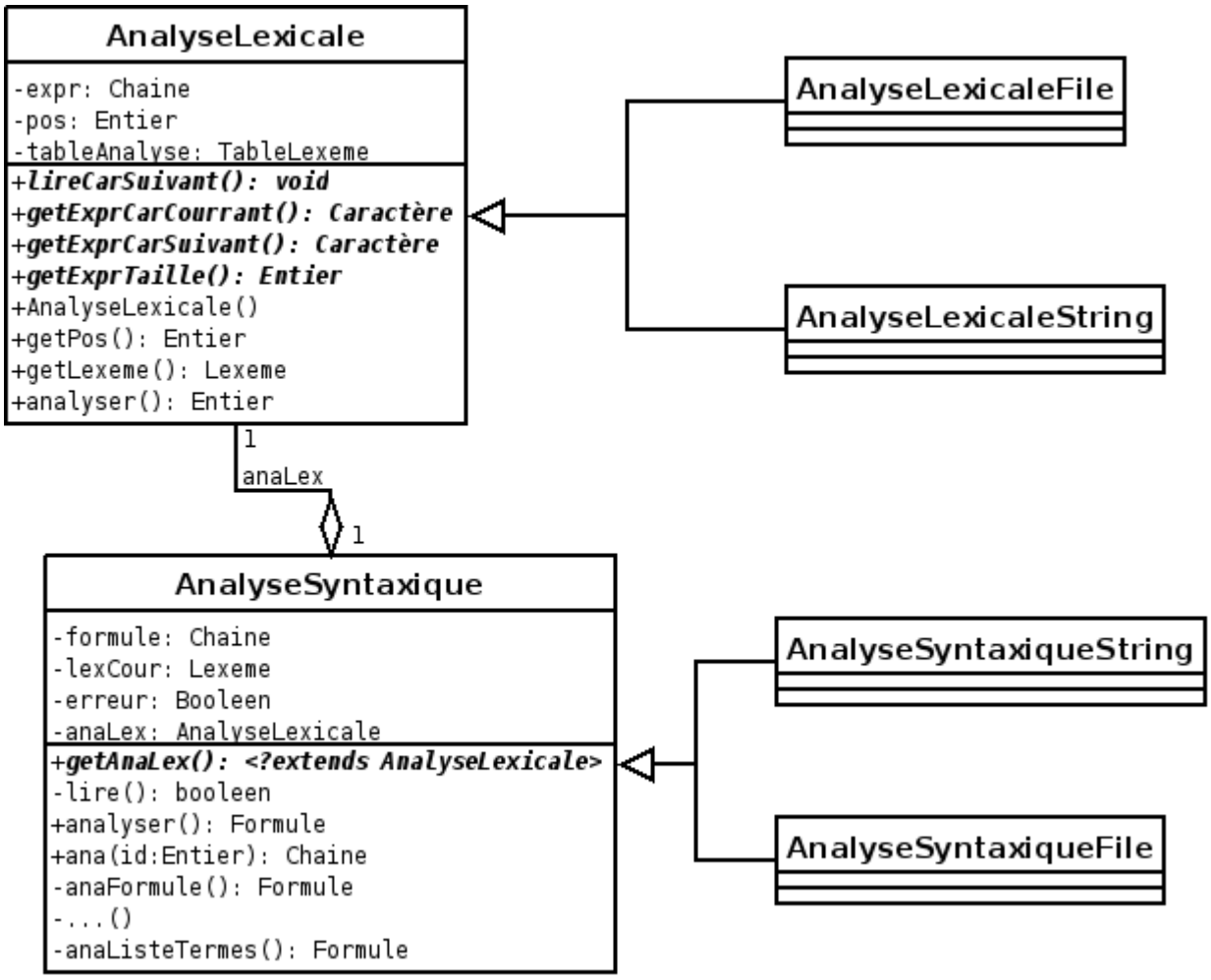
Schéma UML des classes:

- adpo.formule :

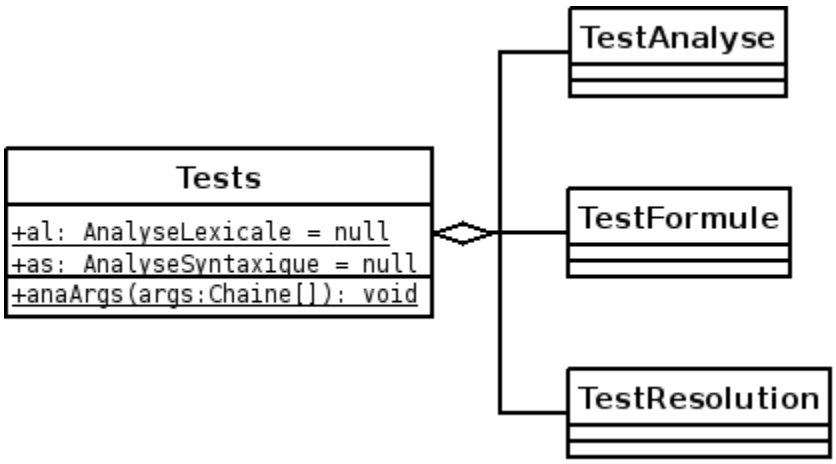




- adpo.analyse :



- adpo.test :



Classes importantes:

- adpo.formule.Clause
- adpo.formule.FormeClausale
- adpo.resolution.ClauseResolution

3 Étape 3

3.1 Liste de Termes

Nous avons tout d'abord créer les différentes classes *Fonction*, *Constante* et *Variable*. Ces trois classes implémentent l'interface *Terme* et la classe abstraite *Ensemble*.

Ensuite, nous avons mis en place une classe abstraite pour gérer les listes de termes (dans *Atome* et *Fonction*), *EnsembleTerme*. Cette classe permet d'effectuer tout type de traitement sur un ensemble de terme, comme l'ajout d'un terme ou son retrait.

3.2 Substitution

La substitution d'une variable au sein d'un atome est faite grâce à la classe *Substitution*. Dans une substitution, on stocke des couples <*Variable,Terme*> correspondant à une variable et au Terme qui lui est substitué. On peut obtenir la substitution correspondant à un nom de variable. Il est aussi possible d'appliquer une substitution à un *Atome* ou plus généralement à un *EnsembleTermes*.

3.3 Unification

Dans une unification on considère une liste de couples de deux termes et d'une substitution (deux termes permettant de stocker temporairement les termes à substituer avant de les identifier clairement).

L'algorithme d'unification utilisé est celui qui nous a été donné. On effectue une simplification puis une decomposition puis une inversion enfin une élimination sur les deux atomes à unifier. Ceci est fait grâce aux méthodes *unifier*, *simplifie*r, *decomposer*, *inverser*, *eliminer* de la classe *Unification*.

3.4 RobinsonResolution

La classe RobinsonResolution (qui étend la classe abstraite Resolution) permet de déterminer si une formule est un théorème de la logique du premier ordre en utilisant le système de résolution de Robinson :

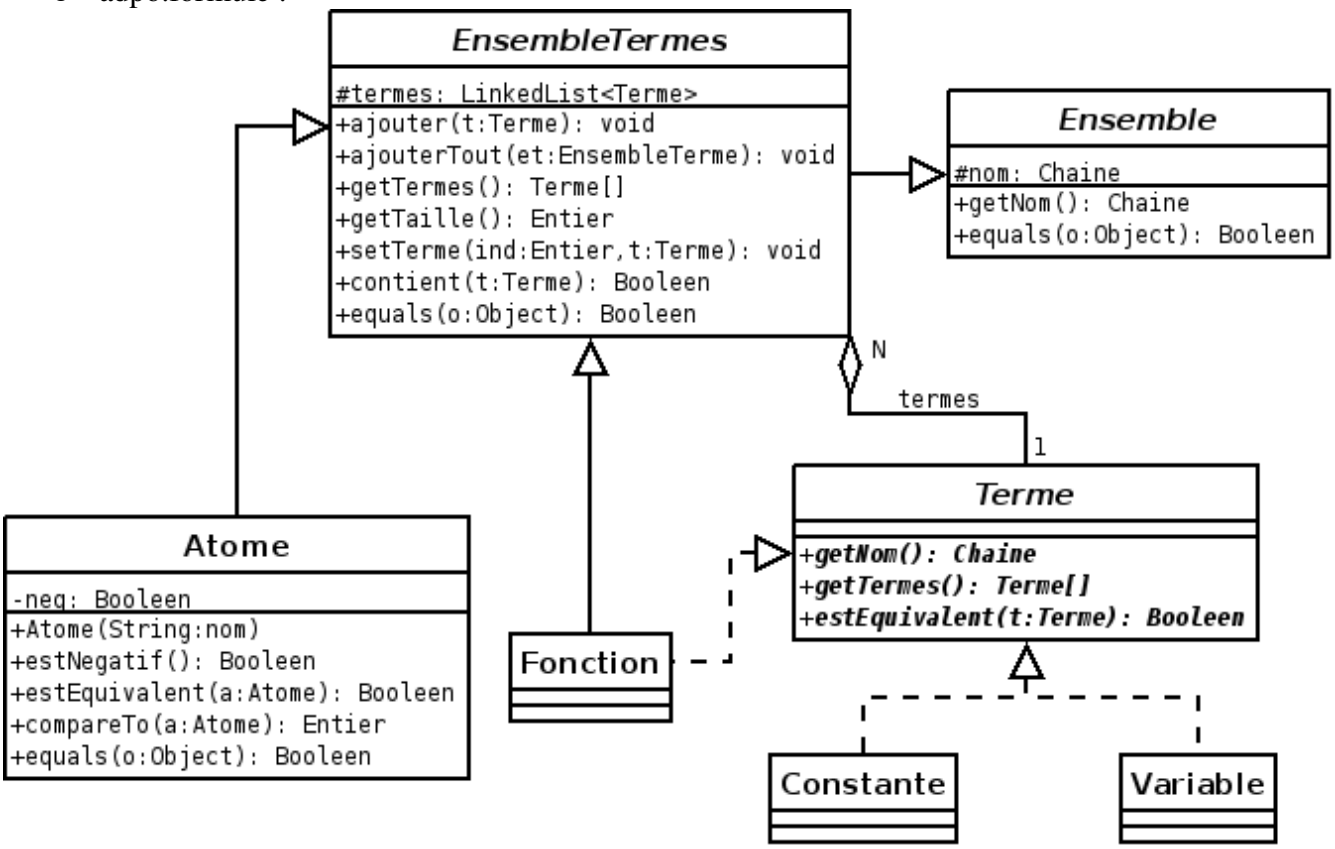
$$F=(Cl(L),\emptyset,(\exists\sigma)(\sigma(a1)=\sigma(a2))$$
$$\{\frac{a1\vee A;\neg a2\vee B,}{\sigma(A)\vee\sigma(B)}(res),\frac{A\vee a1\vee a2}{\sigma(A)\vee\sigma(a12)}(\text{fac+}),\frac{A\vee\neg a1\vee\neg a2}{\sigma(A)\vee\sigma(\neg a1)}(\text{fac-})\})$$

Cette classe s'appuie sur la classe Unification que nous venons d'implémenter en proposant à l'utilisateur d'appliquer une règle sur les atomes qui sont unifiables en proposant pour chacun une substitution adéquate.

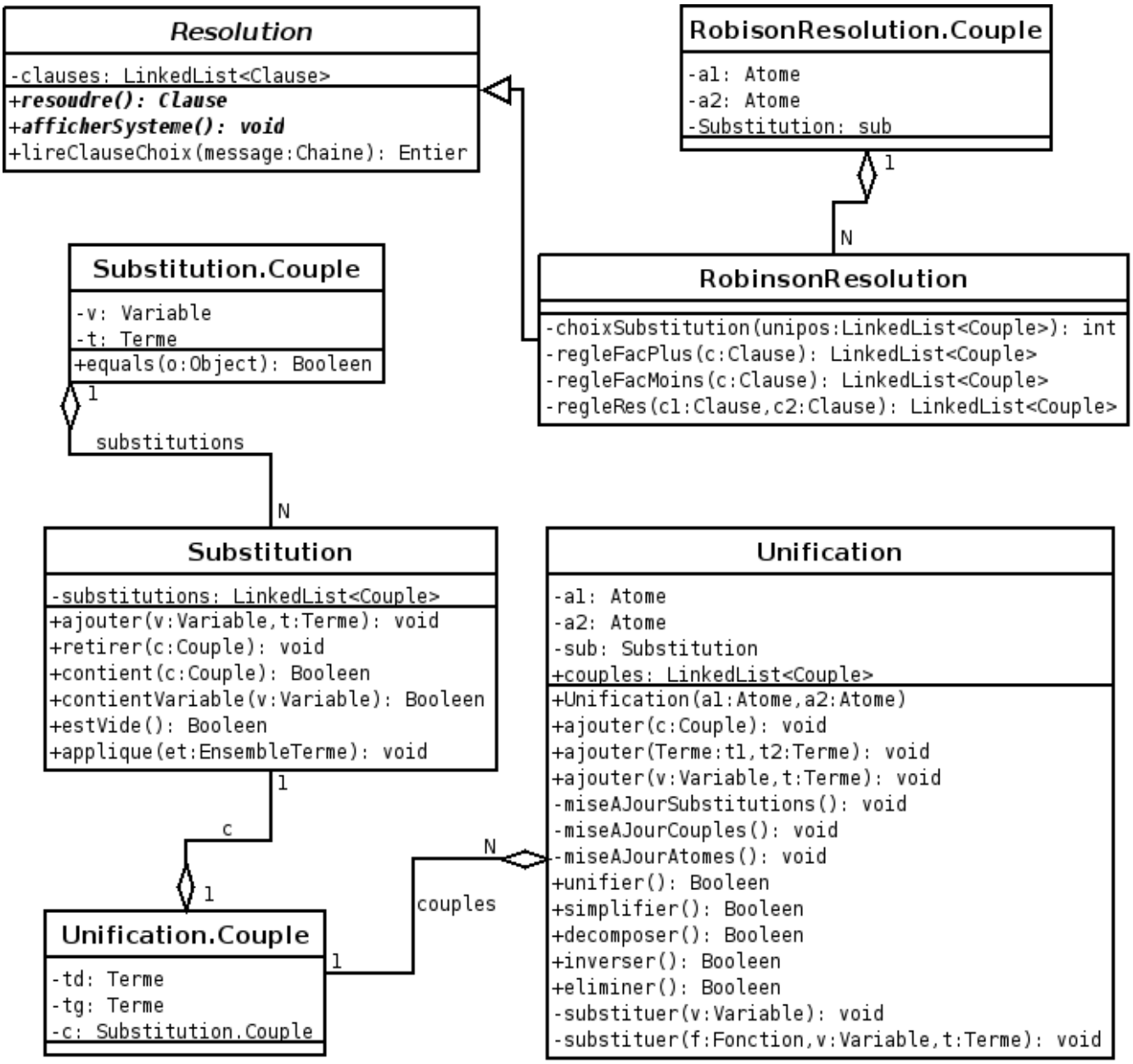
3.5 Résumé

Schéma UML des classes:

- adpo.formule :



- adpo.resolution :



Classes importantes:

- adpo.formule.EnsembleTerme
- adpo.formule.Variable, adpo.formule.Constante, adpo.formule.Fonction
- adpo.resolution.Substitution
- adpo.resolution.Unification
- adpo.resolution.RobinsonResolution

4 Étape 4

4.1 Quantifieurs

Les classes IlExiste et PourTout nous ont permis d'implémenter les quantifieurs. Ces deux classes implémentent l'interface *Formule*, plus précisément la classe abstraite *Binaire* pour laquelle nous utilisons maintenant la généricité. Ainsi, IlExiste et PourTout étendent *Binaire<Variable,Formule>*, nous en avons profité pour transformer *Unaire* de la même façon.

4.2 Cloture universelle

La cloture universelle est implémentée grâce à l'interface *Closible* (que l'interface *Formule* étend). Elle est faite de manière récursive à travers la formule.

Prototype de la méthode :

```
public void clotureUniverselle(Collection<Variable> quantifie,
        Collection<Variable> tout) ;
```

Quand lors de la descente récursive, on tombe sur un quantifieur, on ajoute la variable liée à ce quantifieur dans la liste *quantifie*, quand on rencontre une *Variable*, on l'ajoute dans la liste *tout*.

Ainsi, après l'application de la méthode, il ne nous reste plus qu'à calculer quelles variables sont dans la liste *tout* et pas dans la liste *quantifie* pour avoir la liste des variables libres.

Une fois cette liste obtenue, on clos la formule en ajoutant le quantifieur universel sur les variables libres en début de formule.

4.3 Skolémisation

La skolemisation est faite de manière récursive à l'aide de la méthode *skolemiser* de l'interface *Formule*.

Prototype de la méthode :

```
public Formule skolemiser(Collection<Variable> quantifie, Substitution sub);
```

Lors de la descente récursive, dès que l'on traite un élément de type *PourTout*, on ajoute la *Variable* qui lui est liée à la liste quantifie.

Ainsi lorsque l'on rencontre un élément de type *IlExiste*, il nous est possible d'effectuer une substitution sur l'élément qui lui est associé par une fonction des variables quantifiées précédentes. La substitution à effectuer est alors ajoutée à la *Substitution* passée en paramètre.

Il ne nous reste donc plus qu'à appliquer cette substitution aux *Atomes* qui sont rencontrés dans la suite du traitement.

On a aussi ajouté une classe *Skolemisation* pour nous permettre d'obetnir un identifiant unique pour noms de fonctions qui sont substituées aux variables.

Exemple d'implémentation dans *IlExiste* :

```
public Formule skolemiser(Collection<Variable> quantifie, Substitution sub) {
	Formule ret;
	Substitution.Couple couple;
	Fonction fonct;

	fonct = new Fonction(Skolemisation.getFonctionId());
	for (Variable v : quantifie)
		fonct.ajouter(v);

	couple = new Substitution.Couple(fg,fonct);

	sub.ajouter(couple);
	ret = fd.skolemiser(quantifie,sub);
	sub.retirer(couple);
	return ret;
}
```

Pour la plupart des classes, on se contente de retourner une copie de l'élément courant.

Exemple d'implémentation dans *Ou* :

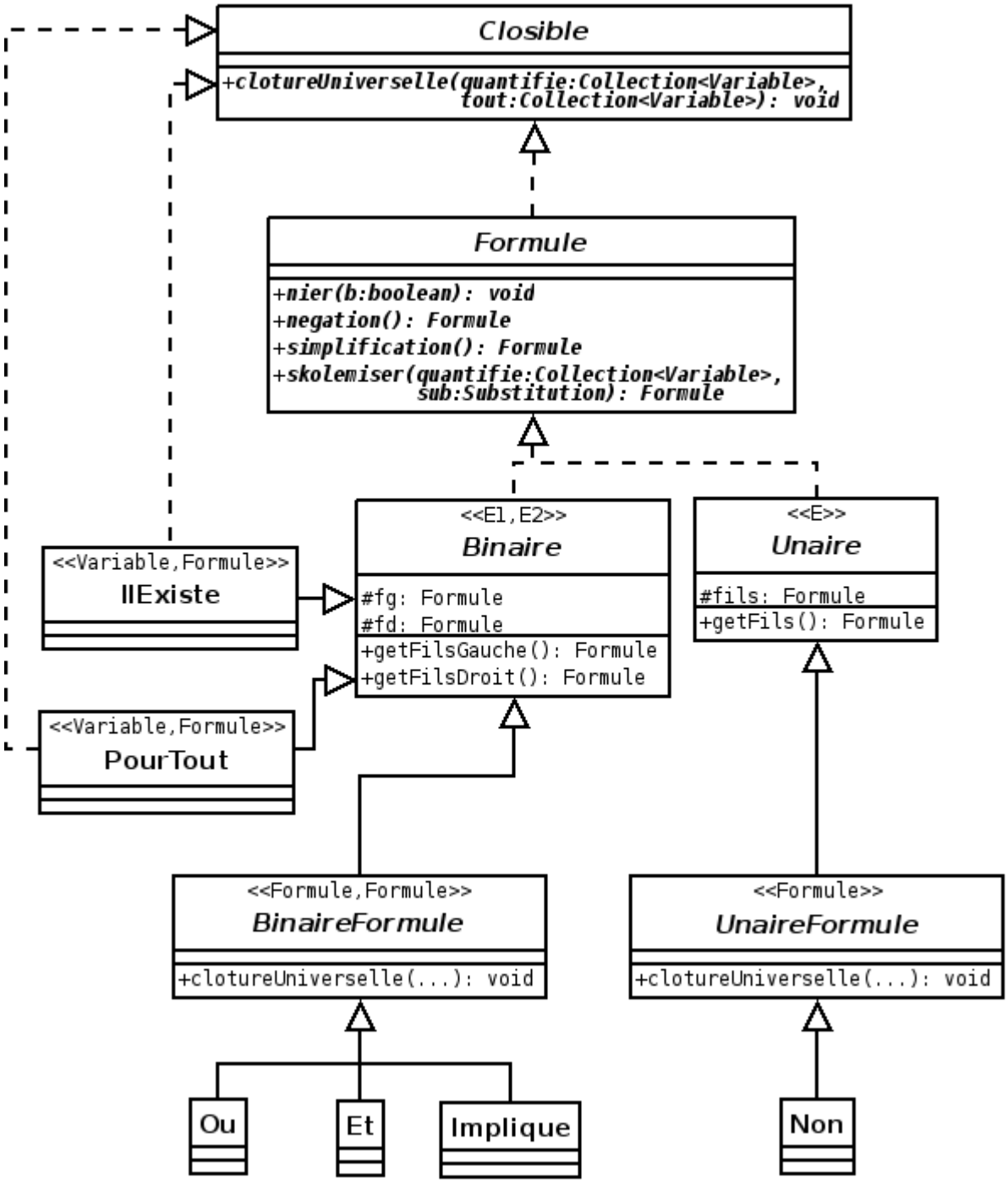
```
public Formule skolemiser(Collection<Variable> quantifie, Substitution sub) {
	return new

	Ou(fg.skolemiser(quantifie,sub),fd.skolemiser(quantifie,sub));
}
```

4.4 Résumé

Schéma UML des classes:

- adpo.formule :



Classes importantes:

- adpo.formule.IlExite, adpo.formule.PourTout
- adpo.resolution.FormeClausale
- adpo.formule.Closible