Université de Caen Basse-Normandie
U.F.R. de Sciences
Département d'informatique

Bâtiment Sciences 3 - Campus Côte de Nacre
F-14032 Caen Cédex, FRANCE

Devoir 2015-2015

Niveau	**M1**
Parcours	**Informatique**
Unité d'enseignement	**UE3 - Programmation et Parallélisme**
Responsable	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr**

1 Systèmes experts

En Intelligence Artificielle Symbolique, un système expert désigne une application permettant de déduire des informations à partir de faits connus et de règles appelées règles d'inférence. Un fait est une propriété vraie ou supposée telle et l'ensemble des faits constitue ce que l'on nomme la base de faits.

En général, une règle est rédigée sous la forme d'une clause de HORN, c'est à dire qu'elle est composée soit :

— d'une conclusion ;
— de plusieurs hypothèses et d'une conclusion.

Les conclusions ne portent que sur un seul et unique fait. Lorsque toutes les hypothèses d'une règle sont vérifiées, celle-ci est dite déclenchable et le fait concerné par sa conclusion est ajouté à la base de faits. Les règles sans hypothèses sont quant à elles systématiquement déclenchables. L'ensemble des règles d'inférence est appelé base de règles. Celle-ci est consignée dans un fichier appelée base de connaissances selon une grammaire particulière.

Les systèmes experts sont classés par catégories. La catégorie 0 (zéro) est la plus simple de toutes : elle signifie que les règles d'inférence n'utilisent pas de variables et que ces dernières sont appliquées à la base de faits jusqu'à ne plus pouvoir rien déduire. Ainsi, dans ce type de système expert, la base de faits est de la forme `{ A }` où `A` désigne un fait, tandis que la base de règles est de la forme :

```
1) A
2) if B then D
3) if A then B
4) if A then C
5) if C and D then E
6) if Z then Y
```

L'algorithme de déduction des faits 1 (présenté plus bas) est appelé moteur d'inférence. Dans notre exemple, pour une base de faits initialement vide, cet algorithme produira la base de faits `{ A, B, C }` à l'issue de la première itération de la boucle `while` (ligne 2 de l'algorithme) par application des règles 1, 3 et 4. La seconde itération produira la base de faits `{ A, B, C, D, E }` par application des règles 2 et 5. Enfin, comme aucun fait nouveau n'aura pu être déduit lors de la troisième itération, l'algorithme s'arrêtera et proposera la base de faits `{ A, B, C, D, E }`. Nous constatons que la boucle `while` extérieure permet d'ignorer l'ordre d'énumération des règles.

Algorithm 1 Moteur d'inférence.

Input : base de faits, base de règles.
Output : base de faits.

```
1: continuer ← true
2: while continuer do
3:    continuer ← false
4:    for all règle in base de règles do
5:       if règle non déclenchée and règle déclenchable then
6:          ajouter la conclusion de règle à la base faits
7:          signaler règle comme déclenchée
8:          continuer ← true
9:       end if
10:   end for
11: end while
```

Le problème posé par la rédaction d'une base de connaissances importante est l'incohérence de ses règles, c'est à dire que plusieurs règles peuvent comporter des conclusions contradictoires. Considérons l'exemple simpliste ci-dessous :

```
1) médecin
2) if médecin then riche
3) if médecin then pauvre
```

Pour une base de faits initialement vide, l'unique itération du moteur d'inférence produira le contenu `{ médecin, riche, pauvre }` alors que les faits `riche` et `pauvre` sont mutuellement exclusifs.

La catégorie 0^+ désigne des systèmes experts manipulant des faits variables et non plus de simples symboles. Comme dans tout langage de programmation, une variable est définie par un type c'est à dire :

— des valeurs acceptables ;
— des opérations autorisées.

Ainsi, le fait `riche`, de type booléen, ne peut prendre comme valeur que vrai ou faux et les opérations permises sur ce fait sont les opérations logiques telles que `and` ou `not`. De même, le fait `profession`, de type symbolique, prend comme valeur une chaîne de caractères et les seules opérations autorisées sur lui sont la comparaison (égalité ou différence). Dans cette catégorie, les bases de connaissances peuvent ressembler à ce qui suit :

```
1) profession = médecin
2) if profession = médecin then riche
3) if profession /= médecin then not riche
4) if riche then salaire = 50000
5) if not riche then salaire = 2000
6) if fortune > 1000000 then millionnaire
```

2 Sujet

En C++, les arguments sont transmis à une fonction ou une méthode uniquement par alias. De même, une fonction ou une méthode ne retourne un résultat par copie que si elle ne peut le faire par alias. Enfin, les méthodes d'instances qui modifient l'état de l'objet qui les invoque doivent être différenciées de celles qui ne le modifient pas. Vous veillerez donc, sous peine de sanctions, à respecter la philosophie du langage. En outre, vous apporterez un soin particulier à l'écriture des classes pour qui la recopie de leurs instances n'a aucune pertinence sémantique. Pour ôter le risque d'une recopie accidentelle, vous pourrez soit implémenter l'idiome de la « non-copyable class », soit exploiter le mot-clé `delete` *introduit par la norme 2011. Enfin, toutes les opérations d'allocation dynamique seront effectuées via des pointeurs intelligents de la bibliothèque standard (pointeurs dotés d'une mécanisme intégré de garbage collector).*

En l'occurrence, la classe générique ***shared_ptr*** *(pointeurs exploitant un comptage des références à un même objet cible) est la plus appropriée.*

Nous nous intéressons à l'écriture du modèle d'un système expert de catégorie 0^+ dont les caractéristiques sont les suivantes :

— trois types de faits sont manipulés : booléens, symboliques et entiers ;
— la grammaire dans laquelle est rédigée la base de connaissance peut varier.

L'ensemble des composants logiciels de cette application sera affecté au paquetage (espace de noms) maître `sysexp`.

2.1 Construction de la base de règles

La construction de la base de règles se fait à partir du fichier contenant la base de connaissances. Comme la grammaire de ce dernier peut varier, il est indispensable de dissocier ce processus de construction du modèle de l'application. Il existe plusieurs solutions permettant de mettre en œuvre cette séparation : dans notre cas, nous choisissons le creational design pattern `Builder` [1] dont la figure 7 présente le diagramme de classes.

2.1.1 Les grammaires

Les fichiers contenant les bases de connaissances sont des fichiers textes rédigés en format libre (ils peuvent contenir des lignes vide, aucune longueur maximum de ligne n'est imposée, etc.). Le contenu de ces fichiers est régi par une grammaire. Quels que soient les mots-clés employés, ces grammaires ont en commun de commencer par la déclaration du type de tous les faits manipulés suivie d'une énumération de règles. La figure 1 présente cette structure commune.

```
Base_De_Connaissances  ::=  Déclarations Règles
```

FIGURE 1 – Structure commune à toutes les grammaires.

La figure 2 présente les règles régissant les déclarations de faits dans une grammaire que nous appellerons LORRAINE (comme il en existera plusieurs, nous leur attribuons des noms de régions pour les distinguer).

Ces règles montrent que :

— les faits booléens sont déclarés en premier puis viennent les faits symboliques et enfin les faits entiers ;
— la déclaration d'une famille de faits commencent par un mot-clé (par exemple `fait_entiers`) suivi du signe égal (`'='`), d'une liste de faits et enfin d'un point-virgule (`';'`) ;
— une liste de faits peut être vide (ce qu'exprime le symbole spécial ϵ) ou (ce qu'exprime la barre verticale) composée d'un fait suivi de zéro, une ou plusieurs fois une séquence constituée d'une virgule (`','`) et d'un fait (ce qu'expriment les accolades). Un fait ne peut être déclaré qu'une seule fois (règle sémantique qui ne peut être exprimée par une grammaire) ;
— un fait, qu'il soit booléen, symbolique ou entier, n'est in fine qu'un simple identificateur ;
— un identificateur commencent toujours par une lettre minuscule ou majuscule suivie de zéro, un ou plusieurs caractères parmi :
  — une lettre ;
  — un chiffre ;
  — le caractère souligne (`'_'`). Si ce dernier est présent alors il doit obligatoirement être suivi d'une lettre ou d'un chiffre.

La figure 3 présente la structure d'une énumération de règles dans la grammaire LORRAINE. Ses caractéristiques sont :

— une énumération est constituée de zéro, une ou plusieurs règles, chaque règle étant terminée par un point-virgule (par conséquent, une énumération peut être vide) ;
— les règles se subdivisent en deux catégories : celles qui possèdent une condition (règles avec prémisses) et celles qui n'en possèdent pas (règles sans prémisse) ;

```
Déclarations              ::=  Déclarations_Booléennes
                               Déclarations_Symboliques
                               Déclarations_Entières
Déclarations_Booléennes   ::=  'faits_booleens' '=' Faits_Booléens ';'
Déclarations_Symboliques  ::=  'faits_symboliques' '=' Faits_Symboliques ';'
Déclarations_Entiers      ::=  'faits_entiers' '=' Faits_Entiers ';'
Faits_Booléens            ::=  ϵ | Fait_Booléen { ',' Fait_Booléen }
Faits_Symboliques         ::=  ϵ | Fait_Symbolique { ',' Fait_Symbolique }
Faits_Entiers             ::=  ϵ | Fait_Entier { ',' Fait_Entier }
Fait_Booléen              ::=  Identificateur
Fait_Symbolique           ::=  Identificateur
Fait_Entier               ::=  Identificateur
Identificateur            ::=  Lettre { [ '_' ] Alphanumérique }
Lettre                    ::=  'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' |
                               'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
                               'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
                               'y' | 'z'
                               'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
                               'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
                               'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
                               'Y' | 'Z'
Alphanumérique            ::=  Lettre | Chiffre
Chiffre                   ::=  '0' | Chiffre_Positif
Chiffre_Positif           ::=  '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
                               '9'
```

FIGURE 2 – Grammaire LORRAINE : règles régissant les déclarations de faits.

— les règles qui ne possèdent pas de condition sont uniquement constituées d'une conclusion ;
— les règles qui possèdent une condition commencent par le mot-clé `si` suivi de la condition, du mot-clé `alors` et d'une conclusion.

```
Règles                ::=  { Regle ';'}
Règle                 ::=  Règle_Sans_Premisse | Règle_Avec_Premisses
Règle_Sans_Premisse   ::=  Conclusion
Règle_Avec_Premisses  ::=  'si' Condition 'alors' Conclusion
```

FIGURE 3 – Grammaire LORRAINE : structure d'une énumération de règles.

La figure 4 présente les règles régissant les conclusions dans la grammaire LORRAINE :
— une conclusion porte toujours sur un seul et unique fait : selon le type de ce fait, la conclusion est qualifiée de booléenne, symbolique ou entière ;
— une conclusion booléenne est soit une affirmation (le fait sur lequel elle porte prendra la valeur vrai) soit une négation (le fait sur lequel elle porte prendra la valeur faux). Les négations sont signalées par le mot-clé `non` qui précède le fait sur lequel porte la conclusion. Une conclusion booléenne n'est sémantiquement correcte que si le fait sur lequel elle porte est booléen ;
— les conclusions symboliques imposent le signe égal derrière le nom du fait sur lesquelles elles portent, ce signe désignant ici l'affectation. La partie droite de cette affectation peut être une constante symbolique (un simple identificateur différent d'un nom de fait symbolique) ou bien le nom d'un fait symbolique. Une conclusion symbolique est sémantiquement correcte si les faits et constantes qu'elle fait intervenir sont tous symboliques. D'autre part, une constante symbolique ne peut être le nom d'un fait booléen ou entier ;
— comme les conclusions symboliques, les conclusions entières imposent le signe égal derrière le fait sur lesquelles elles portent, signe suivi d'une expression entière infixée. Le premier terme de cette expression peut être précédé du signe plus (`'+'`) ou (`'-'`), ce signe étant optionnel (ce qu'expriment les crochets). Une expression entière peut faire intervenir des constantes entières ou bien des noms de faits entiers (l'expression est sémantiquement incorrecte si elle fait intervenir des faits d'un autre type). Une conclusion entière n'est sémantiquement correcte que si l'expression en partie droite de l'affectation l'est et si le fait sur lequel elle porte est bien de type entier.

```
Conclusion             ::=  Conclusion_Booléenne |
                            Conclusion_Symbolique |
                            Conclusion_Entière
Conclusion_Booléenne   ::=  Fait_Booléen | 'non' Fait_Booléen
Conclusion_Symbolique  ::=  Fait_Symbolique '=' Constante_Symbolique |
                            Fait_Symbolique '=' Fait_Symbolique
Constante_Symbolique   ::=  Identificateur
Conclusion_Entière     ::=  Fait_Entier '=' Expression_Entière
Expression_Entière     ::=  [Additif] Terme { Additif Terme }
Terme                  ::=  Facteur { Multiplicatif Facteur }
Facteur                ::=  Constante_Entière |
                            Fait_Entier |
                             '(' Expression_Entière ')'
Additif                ::=  '+' | '-'
Multiplicatif          ::=  '*' | '/'
Constante_Entière      ::=  '0' | Chiffre_Positif { Chiffre }
```

FIGURE 4 – Grammaire LORRAINE : règles régissant les conclusions.

La figure 5 présente les règles régissant les conditions dans la grammaire LORRAINE :

— une condition est constituée d'au moins une prémisse suivie de zéro, une ou plusieurs fois une séquence constituée du mot-clé `et` et d'une nouvelle prémisse ;
— comme une conclusion, une prémisse porte toujours sur un seul et unique fait et, en conséquence, adopte la même classification ;
— une prémisse booléenne est soit une affirmation (le fait sur lequel elle porte vaut vrai) soit une négation (le fait sur lequel elle porte vaut faux). Sa correction sémantique et la même que celle d'une conclusion booléenne ;
— une prémisse symbolique impose un comparateur derrière le fait symbolique sur lequel elle porte. Ce comparateur est soit le signe égal, soit le signe différent (`'/='`). La partie droite de cette prémisse est identique à celle des conclusions symboliques. La correction sémantique d'une prémisse symbolique est identique à celle d'une conclusion symbolique ;
— comme une prémisse symbolique, une prémisse entière impose un comparateur sur les entiers derrière le fait sur lequel elle porte. Sa partie droite est identique à celle des conclusions entières. Par conséquent, le correction sémantique d'une prémisse entière est la même que celle d'une conclusion entière.

```
Condition            ::=  Prémisse { et Prémisse }
Prémisse             ::=  Prémisse_Booléenne |
                          Prémisse_Symbolique |
                          Prémisse_Entière
Prémisse_Booléenne   ::=  Fait_Booléen | 'non' Fait_Booléen
Prémisse_Symbolique  ::=  Fait_Symbolique '=' Constante_Symbolique |
                          Fait_Symbolique '/=' Fait_Symbolique
Conclusion_Entière   ::=  Fait_Entier Comparateur Expression_Entière
Comparateur          ::=  '<' | '>' | '<=' | '>=' | '=' | '/='
```

FIGURE 5 – Grammaire LORRAINE : règles régissant les conditions.

Nous remarquons que les règles régissant les prémisses d'une condition sont très proche de celles régissant les conclusions : cette proximité sera exploitée lors de la modélisation des règles dans la section suivante de ce document.

Finalement, la figure 6 présente un exemple de base de connaissances rédigée dans la grammaire LORRAINE. Dans cet exemple, toutes les familles de faits en comporte au moins un (mais ce n'est pas toujours le cas) et chaque règle occupe une seule ligne (là non plus ce n'est pas toujours le cas).

2.1.2 Pattern `Builder`

La motivation de ce pattern est d'encapsuler le processus de construction pas à pas d'un objet complexe. Les classes et interfaces impliquées sont :

`Builder` : une classe abstraite (ou interface si le langage d'implémentation propose cette caractéristique) proposant une méthode permettant de fabriquer un produit pas à pas. Si le produit est un assemblage de n sous-systèmes alors la première invocation représente l'ajout du premier sous-système au produit et ainsi de suite. Cette classe abstraite est suffisamment générale pour permettre de construire n'importe quel produit ;

`ConcreteBuilder` : une classe qui réalise l'interface `Builder` tout en implémentant un modèle de construction pas à pas (par exemple de type automate). Cette classe propose une méthode permettant à un client de récupérer le produit fini tout en maintenant un lien vers le dernier produit fabriqué : ce lien permet au client d'obtenir plusieurs exemplaires dudit produit sans avoir à recommencer l'intégralité du processus de construction ;

`Director :` une classe permettant de lancer la construction d'un produit en exploitant le builder qui lui est transmis par le client via son constructeur logique. Ce builder peut être modifié par le biais d'un mutateur ;

`Product :` le produit à fabriquer.

```
faits_booleens = avoir_fait_prison, habite_chateau, intelligent, fort,
riche, grand, lourd, malhonnete, parents_riches, pauvre, travailleur,
chercheur, thesard, leger ;


faits_symboliques=profession ;
faits_entiers =fortune,fortune_parents,poids,taille,travail_par_jour,
combien;

chercheur ;
non thesard ;
profession = medecin;
poids = (-46 + 95);
fortune_parents =
  (1000/10 * (25 + 80) / 3 + 50 * (12 - 5) + 14000 / 2) - 1000 ;

si malhonnete et fort alors riche ;
si parents_riches et intelligent alors riche ;
si travailleur et intelligent alors riche ;
si fortune > 10000 alors riche ;
si profession = medecin alors riche ;
si profession = informaticien alors riche ;
si profession = fonctionnaire alors pauvre ;
si non habite_chateau alors pauvre ;
si riche alors non pauvre ;
si pauvre alors non riche ;
si avoir_fait_prison alors malhonnete ;
si grand et lourd alors fort ;
si taille > 185 alors grand ;
si poids > 95 alors lourd ;
si poids < 50 alors leger;
si poids < 50 alors fortune = (2 * fortune_parents) - 1;
si fortune_parents > 10000 alors parents_riches ;
si travail_par_jour > 8 alors travailleur ;
```

FIGURE 6 – Exemple de base de connaissances rédigée dans la grammaire LORRAINE.

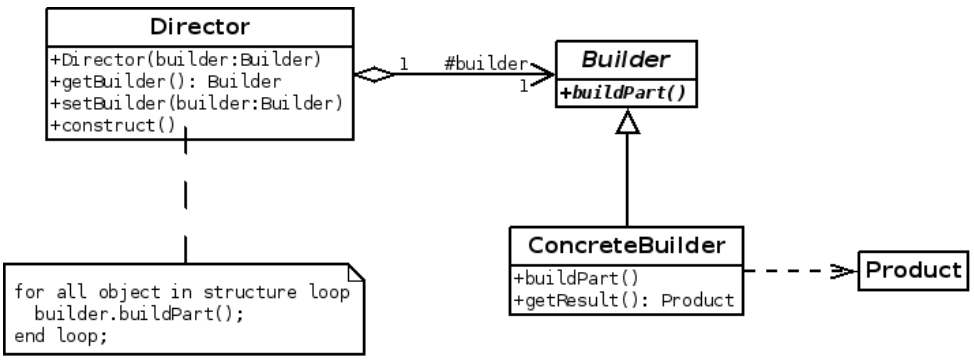

FIGURE 7 – Diagramme de classes du creational design pattern `Builder`.

Le diagramme de séquence de ce pattern est le suivant :

1. le client instancie la classe `ConcreteBuilder` (appelons cette instance `aConcreteBuilder`) ;
2. le client instancie la classe `Director` en fournissant `aConcreteBuilder` à son constructeur logique (appelons cette instance `aDirector`) ;
3. le client invoque la méthode `construct` de `aDirector`, qui, à son tour, invoque autant de fois que nécessaire la méthode `buildPart` de son attribut `builder` ;
4. le client récupère le produit fini en invoquant la méthode `getResult` de `aConcreteBuilder`.

Dans notre cas, le produit à fabriquer (en un seul pas) est une base de règles. Par conséquent, nous allons définir l'équivalent de la classe abstraite `Builder` puis autant de classes dérivées que de grammaires. Le rôle de notre équivalent de la classe `ConcreteBuilder` est triple :
— s'assurer de la correction syntaxique de la base de connaissances ;
— s'assurer de sa correction sémantique (voire règles énoncées ci-dessus) ;
— produire la base de règles correspondante.
De fait, pour chaque grammaire, le programmeur doit définir une hiérarchie de classes permettant de représenter :
— les jetons de la grammaire ;
— une analyseur lexical de la grammaire ;
— un builder concret de base de règles assurant conjointement l'analyse syntaxique, sémantique et la construction de cette base ;
— une classe représentant le directeur de la construction.
Les classes représentant respectivement le directeur et le builder abstrait seront affectées au sous-paquetage `sysexp.builders`. Les composants logiciels liés à une grammaire spécifiques feront l'objet d'un sous-paquetage de `sysexp.builders` : par exemple, les composants associés à la grammaire LORRAINE seront affectés au sous-paquetage `sysexp.builders.lorraine`.

2.2 Le modèle

Nous désignons par modèle les hiérarchies de classes/interfaces représentant :
— les faits ;
— la base de faits ;
— les règles ;
— la base de règles ;
— le moteur d'inférence.
L ensemble de ces composants logiciels sera affecté au sous-paquetage `sysexp.modele`.

2.2.1 Les faits

La caractéristique commune à tous les faits est leur nom (chaîne de caractères). Ils se subdivisent ensuite en fonction de leur valeur (et donc de leur type). Leur modélisation est alors très classique :
— une classe abstraite permettant de consulter le nom du fait ;
— des classes dérivées pour chaque type de fait, ces classes proposant un accesseur permettant de consulter la valeur du fait. Il est ici possible de factoriser le nombre de lignes de code à écrire en utilisant les possibilités génériques du langage de programmation utilisé si celui-ci en possède (ce qui est votre cas).

Cette façon de procéder est relativement courante mais induit un traitement un peu lourd (et sujet à des erreurs de programmation) pour la manipulation des faits, traitement appelé « downcast ». En effet, puisque la valeur d'un fait n'apparaît pas dans la classe ou l'interface représentant le sommet de la hiérarchie, il faut, pour pouvoir accéder à cette valeur :
— identifier le type effectif du fait en question ;
— convertir explicitement le fait dans ce type effectif ;
— accéder enfin à sa valeur.

Le mécanisme du downcast est souvent décrié en programmation par objets car sa mise en œuvre est basée sur une alternative multiple (`switch`) portant sur le type effectif du fait : or, cette alternative est

la marque de fabrique du paradigme de programmation procédural. Les problèmes posés par ce type de construction sont malheureusement connus :

— risque d'oublier une étiquette (`case`) lors de l'ajout d'un nouveau type de fait dans l'application ;
— résidus de types de faits ayant été retirés depuis de l'application ;
— la pertinence de la conversion est laissée à la seule appréciation du programmeur qui n'est jamais infaillible ;
— etc.

Certains préconisent une alternative totalement objet via l'implémentation du behavioral design pattern `Visitor`. Cependant, nous ne l'utiliserons pas ici pour les raisons suivantes :

— elle est un peu lourde à mettre en uvre alors que les faits sont pratiquement des coquilles vides (un nom et une valeur, les deux étant des constantes) ;
— le pattern `Visitor` sera utilisé pour modéliser les conclusions et les prémisses de règles.

2.2.2 La base de faits

La base de faits est triviale : un simple dictionnaire (conteneur associatif de la bibliothèque standard du langage) permettant d'associer un fait à un nom. Ce conteneur permet de :

— rechercher un fait via son nom ;
— ajouter un nouveau couple (nom, fait) ;
— balayer tous les couples (nom, fait) pour, par exemple, afficher le contenu d'une base de faits sur la sortie standard.

2.2.3 Les expressions entières

Prémisses et conclusions entières font intervenir une expression en partie droite respectivement de leur comparateur et du signe d'affectation. Ces expressions sont modélisées classiquement comme des arbres binaires via l'implémentation conjointe du structural design pattern `Composite` et du behavioral design pattern `Interpreter`, tous deux présentés ci-dessous.

Pattern `Composite`

Les structural patterns permettent de composer des objets ou des classes pour obtenir des structures plus complexes, ce qui est le cas d'un arbre binaire. Ainsi, dans le diagramme de classes de la figure 8, la classe abstraite `Component` représente indistinctement une feuille ou un nœud interne de notre arbre. La classe `Leaf` représente une feuille tandis que la classe `Composite` représente un nœud interne : par conséquent, cette dernière propose des méthodes permettant de :

— ajouter un nouveau composant ;
— retirer un composant ;
— consulter un composant particulier.

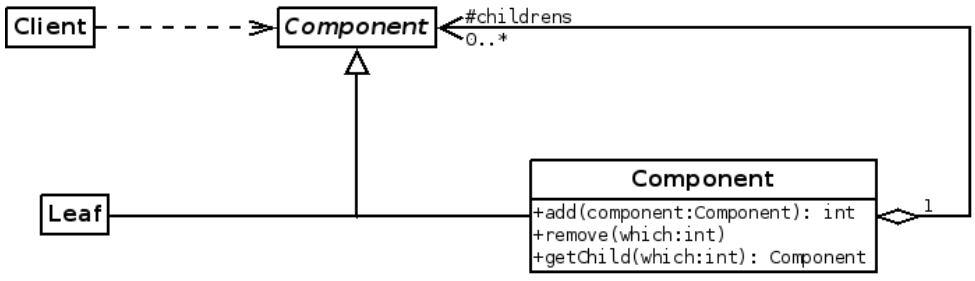

FIGURE 8 – Diagramme de classes du structural design pattern `Composite`.

Pattern `Interpreter`

Tel qu'il a été initialement proposé, sa motivation est d'évaluer les phrases d'un langage en permettant à chaque symbole du texte, terminal ou non terminal, de s'auto-évaluer. La figure 9 présente son diagramme de classes.

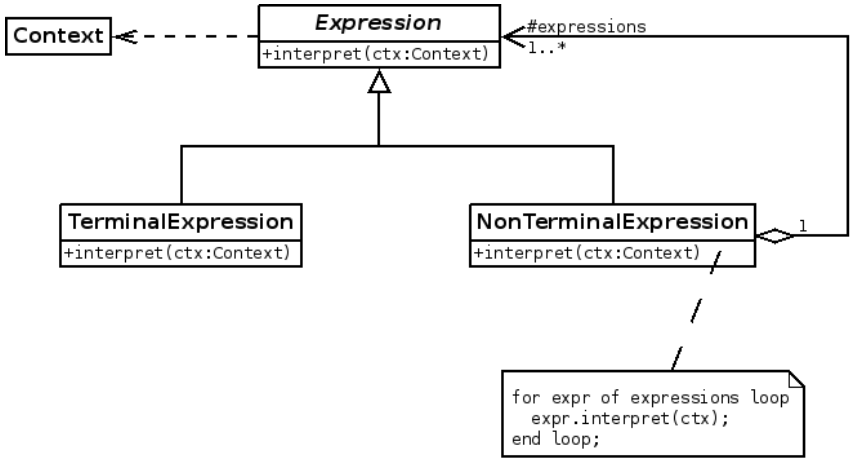

FIGURE 9 – Diagramme de classes du behavioral design pattern `Interpreter`.

Dans ce diagramme, nous reconnaissons immédiatement le pattern auquel `Interpreter` est souvent associé : `Composite`. Ainsi, la classe abstraite `Expression` représente une expression pouvant se décliner en un terminal ou un non terminal du langage. Cette expression est évaluée via un contexte (il peut s'agir d'une pile, d'une file, etc.) fournie à sa méthode abstraite `interpret`, celle-ci étant redéfinie dans les classes dérivées `TerminalExpression` et `NonTerminalExpression`. Dans le cas de cette dernière, la redéfinition de la méthode `interpret` se contente de balayer les expressions du nœud non terminal et d'invoquer pour chacune d'elles leur méthode `interpret`.

Toujours dans le diagramme de la figure 9, nous remarquons tout de suite une spécificité du mariage `Composite+Interpreter` : un couplage très fort entre la structure de données et les opérations qui lui sont associées. Ainsi, si le contexte devait être changé, il faudrait modifier chaque classe du pattern `Composite`, ce qui, en termes de génie logiciel, est un point particulièrement négatif. Par conséquent, ce mariage n'est utilisé que lorsque nous avons la quasi certitude que les opérations demandées à la structure de données n'évolueront pas. Dans le cas contraire, il faut se tourner vers un mariage ou le couplage est beaucoup plus faible : `Composite+Visitor`.

Implémentation du couple `Composite+Interpreter`

Dans notre cas, nous savons que les opérations associées aux expressions entières n'évolueront : par conséquent, le mariage `Composite+Interpreter` est légitime. Le contexte dans lequel les nœuds d'une expression s'interprètent est tout simplement la base de faits. Plus précisément :

- s'il s'agit d'une feuille de type constante entière alors la méthode `interpret` retourne la valeur de cette constante (nous utiliserons le type `long` pour prévenir tout débordement numérique) ;
- s'il s'agit d'une feuille de type fait entier, alors il faut interroger la base de faits pour savoir si elle contient le fait concerné. Si oui, nous le récupérons puis retournons sa valeur. Dans le cas contraire, il faut indiquer que l'expression n'est pas évaluable dans l'immédiat. Comme nous sommes dans un contexte récursif, le mieux est de lever une exception (que vous définirez) pour interrompre l'évaluation de l'expression concernée ;
- s'il s'agit d'un opérateur alors il faut tout simplement évaluer les branches gauche et droite puis combiner les résultats correspondants en fonction du type d'opération (il peut être judicieux d'exploiter le behavioral design pattern `Template Method` afin de factoriser un maximum de lignes de code au niveau de la classe mère des opérateurs : ce pattern sera abondamment vu en TD et TP).

Le problème qui peut survenir lors de l'évaluation d'une expression entière et la division par zéro. Il s'agit d'une erreur sémantique qui ne peut être interceptée au moment du parsing de la base de connaissances et la construction de la base de règles. Par conséquent, lors de l'écriture de la classe représentant l'opérateur de division, il faut vérifier la valeur du dénominateur et lever une exception (que vous définirez) si ce dernier est nul. De fait, votre équivalent de la méthode `interpret` devra mentionner deux exceptions dans sa signature :

— l'une, non critique et ne servant qu'a indiquer que l'expression ne peut être évaluée dans l'immédiat puisqu'elle fait référence à un fait inconnu de la base de faits (mais qui pourra y apparaître plus tard) ;
— l'autre, fatale, indiquant une division par zéro.

2.3 Les prémisses et conclusions

Avant d'aborder la modélisation des prémisses et conclusions, nous commençons par présenter une caractéristiques importante de notre moteur d'inférence : une règle à priori déclenchable (toutes les prémisses de sa conclusion sont vérifiées) mais dont la conclusion ne peut être exécutée. Pour cela, considérons la base de connaissances de la figure 10. Cette base ne déclare que trois faits entiers `a`, `b`, et `c`. Les deux dernières règles (lignes 6 et 7) possèdent des conclusions qui ne pourront jamais être exécutées car comme aucune autre règle ne conclue sur le fait `c`, ce dernier ne pourra jamais se retrouver dans la base de faits. Or, les prémisses de ces règles sont systématiquement vérifiées (les règles sont donc à priori déclenchables) puisque, dans le premier cas, il n'y en à pas alors que dans le second cas, la condition `a > 5` vaut vraie puisque le fait `a` a pour valeur 10. En fait, si nous prenons l'exemple de la règle `b = a * c`, il faut la lire comme :

— si le fait `c` appartient à la base de faits (quelle que soit sa valeur) alors `b = a * c`.

```
1 faits_booleens =;
2 faits_symboliques = ;
3 faits_entiers = a , b, c ;
4
5 a = 10;
6 b = a * c;
7 si a > 5 alors c = 2 * c ;
```

FIGURE 10 – Exemple de règles contenant des conclusions non exécutables.

Par conséquent, une conclusion peut induire une prémisse supplémentaire et vous devez impérativement tenir compte de cette caractéristique dans la modélisation des conclusions.

Venons-en à la modélisation proprement dite. Il existe six types de prémisses : deux booléennes (affirmation et négation), deux symboliques (selon que leurs parties droites soient des constantes ou des noms de faits) et deux entières (selon que leurs parties droites soient des expressions ou des noms de faits). À chaque type de prémisse correspond un type de conclusion. Par conséquent, nous avons un minimum de douze classes concrètes à écrire pour les représenter, sans compter les interfaces qu'elles réalisent ou les classes mères abstraites dont elles dérivent. Or, comme nous l'avons indiqué plus haut, prémisses et conclusions possèdent de nombreux points communs :

— pratiquement aucune différence entre prémisses et conclusions booléennes : les premières effectuent un test tandis que les seconde effectuent une affectation ;
— en grossissant à peine le trait, les prémisses symboliques et entières emportent un comparateur supplémentaire par rapport aux conclusions symboliques et entières.

Cette constatation fait qu'il ne faut surtout pas se lancer dans l'écriture de deux hiérarchies de classes séparées mais au contraire, trouver un moyen de faire partager un maximum de lignes de code entre prémisses et conclusions.

La première idée qui vient à l'esprit est d'utiliser le pattern `Interpreter` exploité par les expressions entières :

— les prémisses s'évaluent dans le contexte d'une base de faits (en lecture seule) et indiquent si elles y sont vérifiées ou pas. Par conséquent, l'équivalent de la méthode `interpret` retourne un résultat booléen. Cette méthode doit également lever une exception pour chaque type d'erreur susceptible d'être rencontrée lors de l'évaluation d'une prémisse. Ici, il n'y en a qu'une : la division par zéro lors de l'évaluation d'une expression entière ;
— les conclusions s'évaluent dans le contexte d'une base de faits (en lecture-écriture) et indiquent si elles sont parvenues à s'exécuter ou pas. Par conséquent, là encore, l'équivalent de la méthode `interpret` retourne un résultat booléen. Comme pour les prémisses, cette méthode doit lever une exception pour chaque type d'erreur susceptible d'être rencontrée lors de exécution d'une conclusion. Ici, il y en a deux :
  — la division par zéro lors de l'évaluation d'une expression entière ;
  — une incohérence dans les conclusions de la base de règles. Il s'agit d'une erreur sémantique (qui ne peut être détectée, ou alors très difficilement, lors du parsing de la base de connaissances et la construction de la base de règles) puisque deux conclusions proposent des valeurs différentes pour un même fait.

Par conséquent, nous avons les différences suivantes :
— la base de faits est en lecture seule dans le cas des prémisses et en lecture-écriture dans le cas des conclusions ;
— les exceptions levées sont différentes.

Nous pourrions nous accommoder de la première différence (après tout, certains langages tel que Java ne font pas ce distinguo sur les types classes) mais la seconde nous chagrine beaucoup plus. Pour tenter de résoudre ces problèmes (il n'y a pas de solution miracle), nous proposons une autre approche basée sur l'implémentation du behavioral design pattern `Visitor` dont la figure 11 présente le diagramme de classes. Cette solution va nous permettre :
— de résoudre proprement le problème des différences d'exceptions entre prémisses et conclusions ;
— de transformer les classes concrètes représentant les prémisses et conclusions en de simples coquilles vides puisque le code des méthodes `interpret` du pattern `Interpreter` est cette fois-ci concentré dans une seule et même classe : le visiteur.

Pattern `Visitor`

Ce pattern permet de découpler une structure de données des opérations qui lui sont appliquées. Plus précisément, il permet à une instance d'une classe externe appelée « visiteur » (ici toute classe dérivant de la classe abstraite `NodeVisitor`), d'accéder aux attributs des instances d'autres classes appelées « visitées » (ici toute classe concrète dérivant de la classe abstraite `Node`). Pour ce faire, chaque classe devant être visitée met à disposition une méthode `accept` prenant comme argument un visiteur. Cette méthode invoque la méthode `visit` de son visiteur avec pour argument le visité : le visiteur peut alors accéder au visité pour effectuer n'importe quel type de traitement.

Le pattern `Visitor` permet d'ajouter à une structure de données des fonctionnalités qui n'étaient pas prévues au départ et ce sans la remettre en cause : de fait, le couplage entre la structure de données et les opérations qui s'y appliquent est beaucoup plus faible qu'avec le motif `Interpreter`. La contrepartie est une centralisation de toutes ces opérations dans la classe représentant le visiteur, classe pouvant rapidement devenir la « classe à tout faire ». Dans la pratique, `Interpreter` et `Visitor` sont souvent utilisés de concert :
— `Interpreter` : toutes les opérations prévues initialement ;
— `Visitor` : une porte laissée ouverte sur la structure de données pour toute opération non prévue initialement.

Implémentation du pattern `Visitor`

Le point commun entre prémisses et conclusions est tout simplement le fait sur lequel elles portent. Par conséquent, nous pouvons définir une classe abstraite (appelons là `Forme`) qui soit commune aux prémisses et aux conclusions. Cette classe représente l'équivalent de la classe abstraite `Node` du diagramme de classes de la figure 11 : de fait, elle prévoit une méthode `accept` prenant comme argument toute instance dont la classe dérive d'un équivalent de la classe abstraite `NodeVisitor` (classe que nous expliciterons ci-dessous).

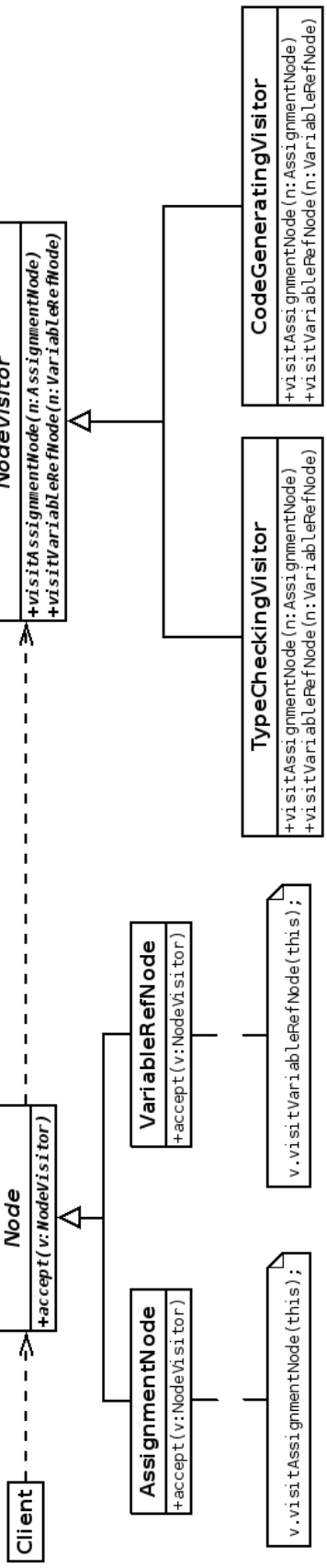

FIGURE 11 – Diagramme de classes du behavioral design pattern `Visitor`.

Notre classe abstraite `Forme` étant définie, nous pouvons à présent représenter nos six types de prémisses et nos six types de conclusions par des classes concrètes qui redéfinissent toutes la méthode `accept` de leur classe de base (il peut être judicieux d'intercaler des classes abstraites dérivant de `Forme`, pour, par exemple, embarquer un comparateur dans le cas des prémisses symboliques ou entières. Mieux : il peut être judicieux de définir des classes abstraites génériques permettant de représenter la partie droite de la forme, cette partie étant instanciée avec de la chaîne de caractères pour tout ce qui est symbolique et de l'expression entière pour tout ce qui est entier). Toutes les classes impliquées dans cette hiérarchie sont des coquilles pratiquement vides puisque les opérations qui leur sont associées sont délocalisées dans un visiteur. Comme ce dernier peut effectuer n'importe quel type de traitement (par exemple du comptage d'un certain type de prémisse, de conclusion, etc.), il est passé en lecture-écriture à la méthode `accept`.

Notre visiteur étant destiné à visiter des formes, l'équivalent de la classe `NodeVisitor` (appelons là `VisiteurForme`) est une classe abstraite qui propose une méthode permettant de visiter chaque forme concrète. Comme ces dernières sont visitées pour effectuer tout type de traitement, les conditions suivantes doivent être vérifiées :

- la forme est passée en lecture-écriture ;
- la méthode concernée ne mentionne aucune exception dans sa signature (encore moins une exception abstraite permettant de représenter plusieurs exceptions concrètes).

Nous définissons à présent un visiteur concret exploité par le moteur d'inférence (appelons cette classe `VisiteurFormeMoteur`) pour :

- savoir si une prémisse est vérifiée dans une base de faits ;
- tenter d'exécuter une conclusion dans une base de faits.

Un visiteur concret est bien souvent un couteau suisse : par conséquent, nous pouvons le doter de tous les attributs nécessaires à un traitement particulier. Dans notre cas, nous aurons besoin de :

- un attribut représentant la base de faits sur laquelle opère le moteur d'inférence ;
- un attribut permettant de savoir si :
  - la dernière prémisse visitée est vérifiée dans la base de faits ;
  - la dernière conclusion visitée a pu s'exécuter dans la base de faits.
- comme les méthodes permettant de visiter une forme ne mentionnent aucune exception dans leur signature, nous définissons un attribut représentant un code d'erreur (le type correspondant sera un type énuméré fortement typé `enum class`). Ce code sera initialisé après chaque visite, ce qui permettra au client du visiteur de savoir si quelque chose s'est mal passé.

2.3.1 Les règles

Le point commun à toutes les règles est de posséder une conclusion. Ces règles se subdivisent ensuite en deux catégories : celles qui possèdent une condition et celles qui n'en possèdent pas. Une hiérarchie de classes permet d'exprimer naturellement ce distinguo, la condition d'une règle n'étant au final qu'une simple liste chaînée de prémisses.

Afin de simplifier à l'extrême l'écriture du moteur d'inférence, nous allons faire en sorte que règles et base de règles soient étroitement imbriquées en implémentant le behavioral design pattern `Chain Of Responsability` dont la figure 12 présente le diagramme de classes.

Pattern Chain Of Responsability

Ce pattern permet à un client d'envoyer une commande (requête) sans nommer explicitement l'objet qui la traitera. Plus précisément, le client émet une requête à l'élément placé en tête de liste. Si ce dernier peut traiter la requête alors il le fait. Dans le cas contraire, il transmet la requête à l'élément suivant dans la liste et ainsi de suite. Ce pattern permet de réduire la dépendance entre le client et l'objet qui traitera la requête puisque ce dernier n'est jamais explicitement nommé.

Implémentation du pattern `Chain Of Responsability`

Le pattern `Chain Of Responsability` induit une structure de liste simplement chaînée. Si les maillons de cette liste sont des règles alors nous obtenons implicitement une base de règles. De fait, si l'équivalent de la méthode `handleRequest` est une méthode `iterer` telle que :

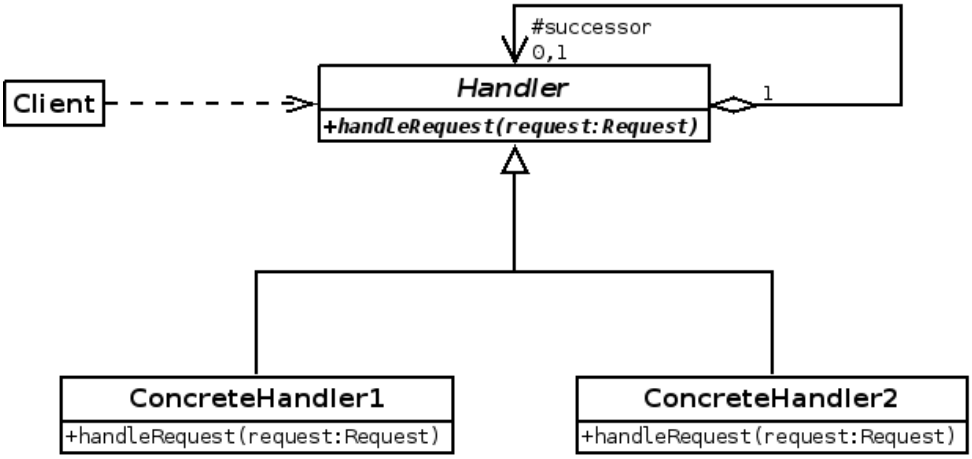

FIGURE 12 – Diagramme de classes du behavioral design pattern `ChainOfResponsability`.

— son argument est une base de faits en lecture-écriture ;
— elle retourne la valeur vrai si au moins une règle de la base de règles a pu se déclencher sinon la valeur faux,

alors nous avons écrit en une seule fois :

— les règles ;
— la base de règles ;
— le moteur d'inférence.

La méthode `iterer` exploite la classe `VisiteurFormeMoteur` pour :

— savoir si toutes les prémisses de la règle (si elle en possède) sont vérifiées dans la base de faits ;
— exécuter la conclusion de la règle dans la base de faits.

Cette méthode doit mentionner une exception pour chaque type d'erreur susceptible de survenir lors d'une tentative de déclenchement c'est à dire :

— une division par zéro ;
— une conclusion incohérente.

Là encore, il peut être judicieux d'implémenter le pattern `Template Method` pour simplifier l'écriture de la méthode `iterer`.

Puisque celle-ci est susceptible de lever deux types d'exceptions liées à un problème dans la base de connaissances, il faut fournir à l'utilisateur suffisamment d'information pour lui permettre de corriger ce problème. Cependant, dans notre cas, comme ce devoir est déjà assez conséquent, nous allons volontairement bâcler cet aspect en lui fournissant comme unique information le numéro de la règle dans la base de connaissances (ce n'est pas son numéro de ligne). Par conséquent, la classe mère des règles devra définir un attribut (de type `unsigned long` puisque le nombre de règles peut être très important) représentant ce numéro dont la valeur sera donnée par le builder de la base de règles.

2.3.2 Le moteur d'inférence

Il est déjà écrit et son exploitation peut être implantée directement dans le code du client (le programme principal) comme le montre la figure 13.

2.4 Le client

Le client est ici le programme principal dont la seule vocation est de tester le modèle. Le client récupérera le nom d'un fichier contenant la base de connaissances (la grammaire dans laquelle est rédigée ce fichier sera LORRAINE). Après avoir effectué les vérifications d'usage (nombre d'arguments de la ligne de commandes incorrect, nom de fichier inconnu, ouverture impossible, etc.), ce client effectuera les opérations

```
try {
  while (baseDeRegles.iterer(baseDeFaits));
}
catch(const ErreurIncoherence& erreur) {
  ...
  return;
}
catch(const ErreurDivisionParZero& erreur) {
  ...
  return;
}
```

FIGURE 13 – Exploitation du moteur d'inférence au niveau du client.

suivantes :

1. construire la base de règles à partir du fichier contenant la base de connaissances. Toute erreur syntaxique ou sémantique lors de cette opération fera l'objet d'un affichage sur la sortie erreur avec, comme informations, le numéro de ligne, le contenu de la ligne, la position (approximative) de l'erreur dans la ligne et le(s) mot(s)-clé(s) attendus ;
2. lancer le moteur d'inférence. Toute erreur sémantique détectée lors de cette phase fera l'objet d'un affichage sur la sortie erreur avec, pour informations, le numéro de la règle concernée et la cause de l'erreur ;
3. écrire le contenu de la base de faits sur la sortie standard.

Travail à rendre

L'objectif de ce devoir, à effectuer en binôme, est la programmation de l'application décrite ci-dessus. Ce travail, qui représente le contrôle continu, compte pour un tiers de la note finale. Il consiste à créer une archive `devoir.tar.gz` constituée de :

- un répertoire `src` contenant les sources. Chaque paquetage et sous-paquetage demandé fait l'objet d'un sous-répértoire dans lequel les définitions (`.cpp`) sont séparées des déclarations (`.hpp`). Ces sources devront être totalement commentées au format DOXYGEN (paquetages, classes, attributs, méthodes (y compris leur code respectif)) ;
- un script de compilation `cmake` ;
- un fichier `doxyfile` permettant de générer automatiquement la documentation au format `html` ;
- un exemple de base de connaissances (attention, testez votre application sur plusieurs exemples et non pas sur un seul) ;
- un fichier texte `noms.txt` contenant les noms et groupes de TD et TP des composantes du binôme ;
- un répertoire UML contenant le diagramme de classes de toute l'application au format `dia`. Vous pourrez partitionner en deux diagrammes : builder et grammaire d'un coté, modèle de l'autre.

Ce travail sera remis au plus tard le jour du dernier cours magistral (minuit dernier délai). Il sera déposé sur le système accessible à partir de `https://devoirs.info.unicaen.fr/`. Seul l'un des deux étudiants déposera une archive `devoir.tar.gz` avec les fichiers du devoir. L'autre déposera une archive contenant simplement le fichier `noms.txt`. Ce système (qui vous paraît certainement étrange) permet aux enseignants de :

- ne pas saturer inutilement l'espace disque du serveur sous-jacent ;
- envoyer aux deux composantes du binôme la correction détaillée par mail via le serveur ;
- disposer de la liste de tous les étudiants ayant rendu le devoir avec, pour chacun, la note correspondante. Cette liste est utilisée au moment de la collecte des notes de chaque unité de valeur en vue de la préparation des jurys.

La correction sera effectuée sur les machines de la salle S3 403-404. Tout manquement aux consignes données ci-dessus (absence ou quasi absence de commentaires, non respect de la conception objet demandée, absence de script de compilation ou script dépendant d'un environnement de programmation, format autre que dia pour les diagrammes de classes, etc.) se traduira par une réduction de la note attribuée au binôme. L'absence d'un fichier `nom.txt` se traduira par le retrait d'un point sur la note de l'étudiant fautif.

Références

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.