IFT 3051 – PROJET D'INFORMATIQUE

Définition du langage PIL (Pattern Identification Language)

Rapport de projet remis à:

M. Petko Valtchev, Ph.D.
M. Yann-Ga?l Guéhéneuc, Ph.D.

Par:

Karim Lahrichi
Jean-Nicolas Malek

Département d'informatique et de recherche opérationnelle
Université de Montréal
Hiver 2004

Table des matières

1 – Introduction

Les patrons de conception sont des modèles de solution à des problèmes génériques de conception, applicables dans des contextes précis. Depuis leur apparition, et principalement grâce au travail de Gamma et al, ils ont suscité beaucoup d'intérêt.
Les patrons de conception sont en général représentés par des diagrammes avec des notations objet comme UML, accompagnés le plus souvent de descriptions textuelles et d'exemples de code pour compléter la description. Malheureusement, l'utilisation et/ou l'application d'un patron peut s'avérer difficile ou inexacte; en effet, les descriptions existantes ne sont pas des définitions formelles et laissent parfois planer une certaine ambiguïté sur le sens exact des patrons.

Notre projet consiste à essayer de donner langage permettant d'écrire des spécifications formelles de l'aspect dynamique des patrons de conception. Dans ce rapport, nous décrivons les différentes étapes de notre cheminement.

Nous commençons par donner un aperçu critique de quelques approches de modélisation existantes. Ensuite, nous présentons nos propres tentatives en vue d'obtenir un langage de description des patrons de conception comportementaux. Enfin, nous décrivons le langage final obtenu à travers plusieurs exemples.

2 - Les approches existantes

Parmi les approches existantes, nous en avons retenu trois principales.

2.1 - Première approche : LayOM

LayOM est un langage qui permet d'utiliser les patrons de conception, au niveau de l'implémentation, en tant "qu'éléments de première classe", c'est à dire que les patrons de conception sont explicitement représentés dans le langage. Cette initiative est motivée par les nombreux problèmes auxquels il faut faire face au moment d'implémenter un patron de conception. Ces problèmes, très bien décrits dans l'article de J. Bosch, sont les suivants:

1 - "Traceability": les langages de programmation actuels ne prennent pas directement en compte le concept de patrons de conception. Par conséquent, ceux-ci sont souvent noyés dans le code, et il est très difficile de les distinguer ou de les "retracer".

2- "Reusability": comme les patrons de conception ne sont pas des éléments de première classe dans le langage, leur implémentation ne peut pas être réutilisée telle quelle. Le programmeur doit donc les recoder à chaque fois.

3- "Implementation Overhead": l'implémentation des patrons de conception nécessite souvent de rajouter du code qui leur est spécifique, ce qui demande

plus d'efforts au programmeur et rend souvent le programme moins lisible dans son ensemble.
Le langage LayOM se base sur le concept de “layers” qui encapsulent les objets et qui interceptent les messages reçus et envoyés par ces objets.
Le patron de conception *Strategy* représente un algorithme par un objet, de manière à ce que les objets utilisant ce patron puissent modifier leur comportement en remplaçant l'objet *Strategy* utilisé.

Dans LayOM, un layer de type *Strategy* a la syntaxe suivante:

<id> : Strategy(delegate [<mess-sel>+ |] to <class> [set by <mess-sel>]);

La figure ci-dessous représente une classe Sensor avec des stratégies “update” et “calculate”:

Figure 1 : Une classe Sensor en LayOM

```
class Sensor
  layers
    update : Strategy( delegate to UpdateStrategy set by newUpdateStrategy);
    calculate : Strategy( delegate to CalculationStrategy set by newCalculationStrategy);
end;
```

Le patron de conception *Observer* est utilisé lorsqu'un ensemble d'objets doit être informé de tout changement d'état d'un autre objet.

La syntaxe du patron *Observer* dans LayOM est la suivante:

<id> : Observer(notify [before|after] on <ness-sel>+ [on aspect <aspect>]);

La figure 2 illustre une classe sujet ObservablePoint.

Figure 2: Une classe ObservablePoint en LayOM

```
class ObservablePoint
  layers
    st : Observer(notify after on setX on aspect “x-axis”, notify after on setY on aspect”Y-
    axis”, notify after on moveTo on aspect “Location”
…
  methods:
    setX(newX : Location) returns Location
        begin … end;
    setY(newY : Location) returns Location
        begin … end;
    moveTo(move : Location2D) returns Location2D
        begin … end;
    …
end; // class ObservablePoint
```

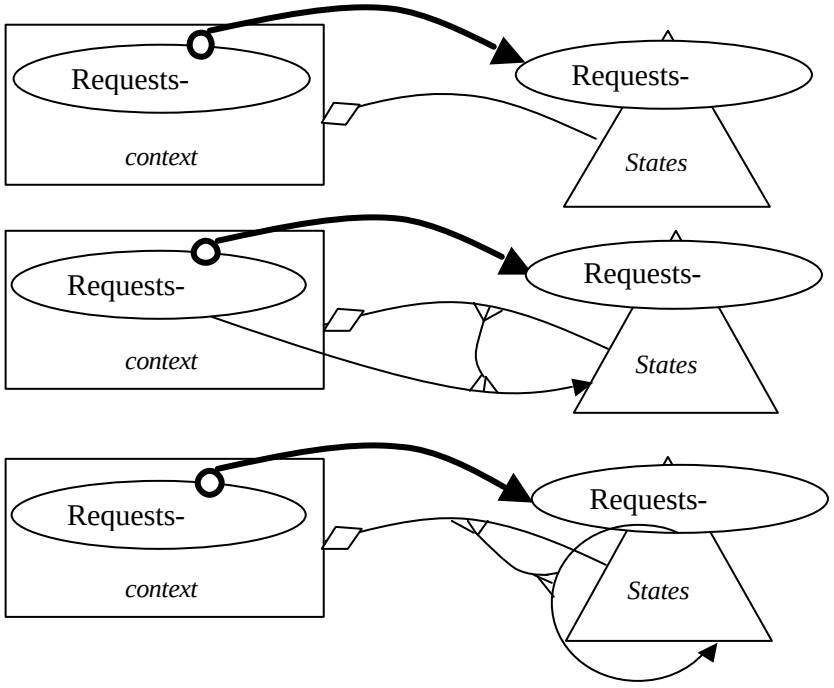
langage LayOM bénéficie d'un mécanisme de traduction automatique qui transforme les classes LayOM en code C++.

Cependant, la solution proposée, bien que très intéressante, donne seulement des exemples d'implémentation des patrons de conception. De notre côté, nous cherchons plutôt à donner des définitions formelles de ces patrons, afin d'en faciliter l'analyse, l'utilisation automatisée et l'extension.

2.2 - Deuxième approche : LePUS

LePUS est un langage de spécification des patrons de conception qui se veut compacte, formel et visuel. Les patrons de conception y sont représentés par des diagrammes qui correspondent à des formules logiques du second ordre (voir figure 3 pour le diagramme de *State*).

Figure 3: Diagrammes LePUS du patron State



Actuellement, les seuls moyens de spécifier les patrons de conception sont les notations objets (comme par exemple UML), les descriptions textuelles et les exemples de programmes. Le langage LePUS réussit à palier l'ambiguïté et le manque de précision de ces méthodes, au prix d'une grande complexité. Ainsi, il s'avère plus simple de comprendre le fonctionnement et le but d'un patron décrit par les méthodes précédentes, que d'essayer de décrypter un diagramme LePUS.
De plus, l'aspect dynamique est quasiment absent dans les descriptions LePUS. Or, dans le cadre de notre projet, c'est justement ce que nous cherchons à modéliser, à savoir: le déroulement de l'exécution des patrons.

2.3 - Troisième approche : DisCo

Ce langage décrit un patron de conception avec un ensemble d'actions et relations.
Chaque spécification DisCo est une description du comportement séquentielle d'un patron de conception basée sur la logique temporelle des actions.

Figure 4: *Observer dans le langage DisCo*

```
A ttach( s:Subject; o: Observer):
        ¬s.Attached.o
        ?   s.Attached.0,

Detach(s:Subject; o:Observer):
        s.Attached.o
         ?  ¬s.Attached'.o
        ^¬s. Updated'.o.

Notify(s:Subject. d):
        ?   ¬s. Update',class Observer
        A s.Data' = d,

Update(s:Subject; o*:Observer; d):
        s.Attached.o
        ^ ¬s. Updated.o
        ^ d = s.Data
        ?   s' Updated.o
        ^ o.Data' = d.
```

Cette approche nous parait la plus intéressante dans le cadre de notre projet. En effet, elle tient compte de l'aspect temporel. DisCo nous semble être un bon point de départ pour notre langage, qui insistera d'avantage sur le déroulement de l'exécution du programme.

3 – Différentes tentatives de langage

Nous présenterons dans cette partie les différentes approches explorées pour la conception de notre langage.
Suite à nos lectures, nous étions indécis sur la voie à suivre pour notre projet. Nous avons donc un peu tâtonné au départ.

3.1 – Langage à base de predicats

Notre première tentative a été inspirée du langage LePUS, et se basait donc sur la logique des prédicats. Nous n'avons pas eu de mal à décrire les patrons de conception *TemplateMethod* et *Strategy*. Cependant, nous avons vite réalisé que cette approche omettait entièrement l'aspect dynamique des patrons de conception, et nous avons alors commencé à nous orienter vers l'aspect temporel des patrons de conception.

3.2 – Logique temporelle

Certains patrons de conception, tels que *Singleton* et *Observer*, présentent un aspect temporel marqué.
Pour le patron *Singleton*, il s'agit de vérifier qu'une fois qu'un objet d'une certaine classe est créé, aucun autre objet de la même classe ne le sera.
Pour le patron *Observer*, il faut vérifier qu'après tout changement d'état du sujet, les observers associés en sont informés.

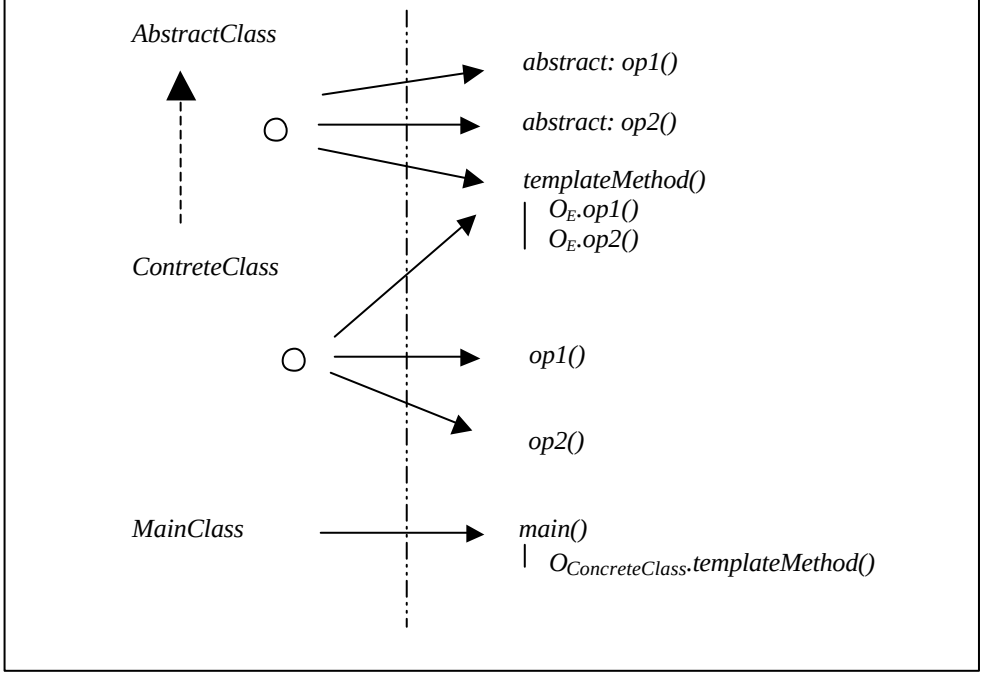
Si cette approche réussit à décrire certains patrons de conception, elle n'est pas adaptée à des patrons où l'aspect structurel est prédominant par rapport à l'aspect temporel, tels que *TemplateMethod* et *Strategy*.

3.3 - Approche a base de diagrammes dynamiques

Nous avons cherché à décrire l'exécution des patrons de conception, c'est-à-dire l'aspect dynamique plutôt que statique. Nous nous sommes aidés pour cela du livre *Programming for the java virtual machine*, qui décrit les opérations effectuées par la machine virtuelle lors de l'exécution d'un programme java. Nous avons donc essayé de décrire de cette manière le patron de conception *TemplateMethod.* Nous avons choisi une description sous forme de schéma, qui souligne l'aspect dynamique de la description. Nous avons aussi cherché à souligner l'aspect dynamique de l'héritage et du polymorphisme en programmation objet, c'est-à-dire comment ces deux concepts se traduisent à l'exécution.

Voici le schéma du patron de conception *TemplateMethod*.

Figure 5 : TemplateMethod

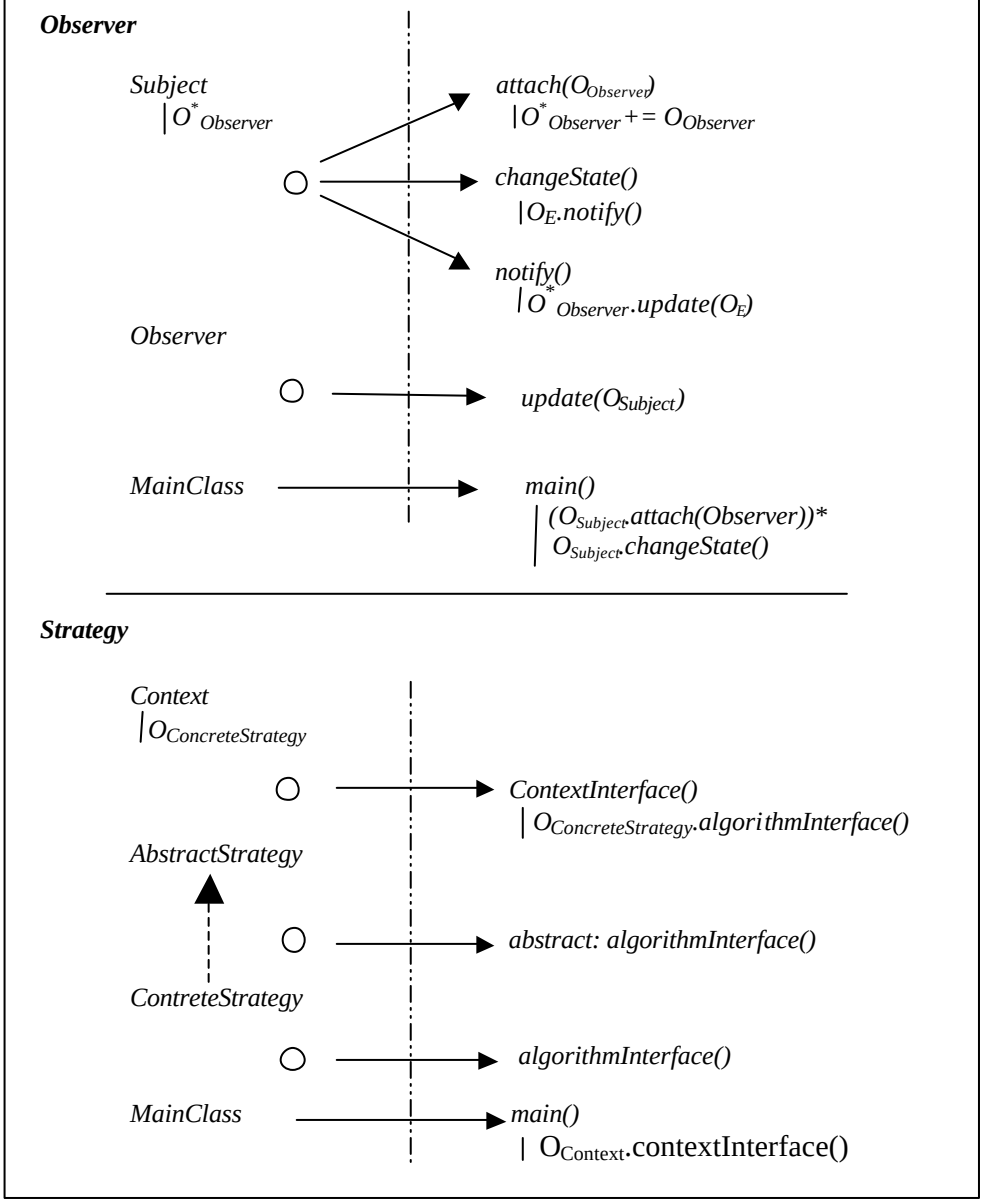


Pour comprendre l'exécution du patron de conception, il suffit de lire le schéma en suivant les flèches. On part la méthode *main* de la *MainClass*. Celle-ci fait appel à la méthode *templateMethod* d'un objet o de la classe *ConcreteClass*. On se place donc du point de vue de l'objet o. Les méthodes *op1* et *op2* sont définies dans cette classe, il y a donc un lien entre o et ces deux méthodes. La méthode *templateMethod* est quand à elle héritée de la classe *AbstractClass* et n'a pas été redéfinie.

Les instructions *Oe.op1* et *Oe.op2* sont donc exécutées, où *Oe* désigne l'objet qui a invoqué la méthode en cours, soit *o*. Ce sont par conséquent les méthodes *op1* et *op2* liées à o (donc de la classe *ConcreteClass*) qui sont exécutées.

Cette approche semble adaptée pour décrire les patrons de conception d'un point de vue dynamique. Nous l'avons donc utilisée pour les patrons de conception *Observer* et *Strategy*

Figure 6: Observer et Strategy



Nous n'avons cependant pas réussi à décrire des patrons tels que *Singleton* et *State*, car le langage utilisé ne nous a pas permis d'exprimer les boucles et les branchements conditionnels.

3.4 – Langage de programmation

Nous avons essayé de trouver un langage qui ressemblerait d'avantage à un langage de programmation.

Son fonctionnement est similaire à celui d'un parseur: quand une méthode est appelée certaines conditions doivent être vérifiées pour qu'un patron de conception soit identifié.
La figure 7 présente la description des patrons *Singleton*, *TemplateMethod*, *Startegy* et *Observer* dans un tel langage.
Voici une explication des notations utilisées :

init: initiation globale
when: à l'appel d'une méthode
before, after: avant et après l'exécution de la méthode
assert: condition(s) pour que le patron de conception soit respecté
matchFlow: vérifie qu'une suite d'instructions ont été exécutées, pour que le patron conception soit respecté

Figure 7: Langage de programmation

```
Singleton:
    init:
      n = 0;
    when:
      C.new     //création d'un objet de la class C
    after:
      n++;
      assert(n==1);

TemplateMethod:
    when:
      ConcreteClass.templateMethod on o
    after:
      matchFlow(AbstractClass.templateMethod on o;
                {ConcreteClass.op on o;}+);

Strategy
    when:
      Context.method on o1
    after:
      matchFlow(AbstractStrategy.strategy on o2;

Observer:
    init:
      sync = true;
    when:
      Subject.method on o1
    before:
      a1 = o1.attributeSet;
    after:
      a2 = o1.attributeSet;
      if(a1 != a2){
        assert(sync == true);
        sync = false;
      }
    when:
      Observer.update on o2
    after:
      sync = true;
```

3.5 – Notation XML

Afin d'obtenir des descriptions plus simples et plus lisibles, nous nous sommes inspirés du langage XML. Ainsi, une méthode est représentée par une balise <*method …*/> (ou <*mutator-method …*/> si la méthode modifie l'objet qui l'appelle) avec des attributs pour la classe, l'objet appelant, les paramètres de la méthode. Nous avons aussi inclus un attribut *name* pour pouvoir faire référence à une méthode spécifique.
Toutes les méthodes entre les balises <*> et </*> doivent être exécutées au moins une fois.
L'attribut *depends*, lui, signifie qu'à l'intérieur d'une balise <*>, une méthode *m* dépend d'une variable *x* (qui peut être une classe, un objet ou une autre méthode), c'est-à-dire que si *x* change (d'une itération à l'autre) alors *m* change aussi.

Figure 8: Visitor, Singleton, TemplateMethod et Observer

```
Visior:
   <method class="Structure" param="Visitor v" />
   <*>
      <method class="Element" param="v" caller="e" />
      <method class="ConcreteElement" param="v" caller="e" />
      <method class="Visitor" param="e" caller="v" depends="ConcreteElement" />
   </*>

Singleton:
   <constructor class="C" />
   <!constructor class="C"/>

TemplateMethod
   <method class="ConcreteClass" name="templateMethod" />
   <method class="AbstractClass" name="templateMethod" />
   <*>
      <method class="AbstractClass" name="!templateMethod" />
      <method class="ConcreteClass" name="!templateMethod"/>
   </*>

Observer:
   <mutator-method class="Subject"caller="s" />
   <method class="Observer" depends="s" />
```

Le problème de cette méthode est qu'elle nécessite l'utilisation de nouvelles balises et de nouveaux attributs quasiment à chaque nouveau patron. De plus, pour le patron de conception *ChainOfResponsability*, nous n'avons pas réussi à exprimer le fait que chaque objet appelant doit être différent de l'objet appelant précédent.

4 – Résultat final

Notre langage final est orienté vers deux buts: la détection et la vérification. On veut vérifier qu'un certaine classe vérifie un certain patron de conception, ou encore on veut trouver dans un programme les classes qui satisfont un certain patron de conception, ce deuxième cas se ramenant évidemment au premier, sauf qu'on effectue la vérification pour toutes les classes au lieu d'un seule.
Ce que l'on vérifie, c'est qu'une suite d'instructions est exécutée; ces instructions sont génériques, dans le sens que les classes et les méthodes auxquelles elles font référence sont des variables, qui peuvent représenter respectivement n'importe quelle classe ou méthode du programme. Nous écrivons une instruction de la manière suivante :
Class.method(parameters) [on object]. *Class* et *method* représentent les variables classe et méthode; *parameters* est une liste optionnelle de paramètres; *object* est l'objet invoquant la méthode, et il n'est pas obligatoire non plus de l'inclure.
Enfin, on se donne par contre la possibilité de spécifier certaines contraintes s'appliquant sur ces variables et leurs relations.
Prenons tout de suite des exemples pour se familiariser avec le langage :

Figure 9: Visitor

```
Pattern Visitor{
   Repeat{
      Element.accept(Visitor v) on e
      ConcreteElement*.accept(v) on e
      Visitor.visit*(e) on v
   }
   SuchThat{
      subclass(ConcreteElement , Element)
      depends(visit , ConcreteElement)
   }
}
```

Le bloc d'instructions entre accolades précédé du mot clé *Repeat* peut-être exécuté une ou plusieurs fois. Avant d'entrer dans le bloc *Repeat*, on regarde s'il est suivi d'un bloc *SuchThat*; un bloc *SuchThat* spécifie une série de conditions (booléennes) ou propriétés qui doivent être vérifiées, conditions qui s'appliquent à des variables (classes ou fonctions) qui se trouvent à l'intérieur du bloc *Repeat*. Le bloc *SuchThat* peut se trouver soit à la suite d'un bloc *Repeat*, auquel cas les conditions énumérées sont vérifiées à chaque itération dans la boucle, soit à la suite d'un bloc Pattern (donc à la fin de la définition d'un patron), auquel cas les conditions sont vérifiées une seule fois. En général, on fera suivre un bloc *Repeat* d'un bloc *SuchThat* s'il y a dans le bloc *Repeat* des éléments qui peuvent changer d'une itération à l'autre. Sinon, les conditions à vérifier seront énumérées à la fin du bloc Pattern. Dans le cas ci-dessus, on a exprimé premièrement le fait que la classe *ConcreteElement* doit être une sous-classe de la classe *Element*, et deuxièmement que la méthode *visit* dépend de la classe *ConcreteElement* (c-à-d que si la classe *ConcreteElement* change, alors la méthode *visit* change aussi). Le fait que ces deux variables (la classe *ConcreteElement* et la méthode *visit*) peuvent changer se traduit par un astérisque à la fin de leur nom. Ainsi, à chaque itération dans le bloc *Repeat*, la variable *ConcreteElement* pourra représenter une classe différente, et la méthode *visit* pourra représenter une méthode différente, en autant que les conditions citées plus haut restent vérifiées.

Figure 10: *Singleton*

```
Pattern Singleton{
   C._init
   Repeat{
      X*.m*()
   }
   SuchThat{
      !(X==C && m==_init)
   }
}
```

Le code qui précède (figure 10) décrit le patron *Singleton* comme suit:
après avoir exécuté *C._init* (i.e. après avoir crée un objet de la classe C), on peut exécuter n'importe quelle méthode de n'importe quelle classe, sauf la méthode *C._init*.
En pratique, si on voulait appliquer ce bout de code à un programme donné, voici comment ça devrait se passer :
On commence à la ligne « *C._init* »; à ce moment là, toute classe *C* qui crée un nouvel objet réussit à passer cette ligne et est une candidate potentielle pour le patron Singleton. Pour chacune de ces classes, on arrive au bloc *Repeat;* là, on observe toutes les méthodes invoquées par le programme; si l'une d'entre elles se trouve être *C._init*, alors la classe *C* en question est éliminée et n'est plus une candidate potentielle. Ainsi, à la fin du programme (ou à la fin de la période de vérification), les classes qui ont instancié un et un seul objet seront considérées conformes au patron de conception *Singleton.*

Figure 11: *TemplateMethod*

```
Pattern TemplateMethod{
   AbstractClass.templateMethod() on o
   Repeat{
      AbstractClass.op*() on o
      ConcreteClass.op*() on o
   }
   SuchThat{
      op != templateMethod
   }
}
SuchThat{
   subclass(ConcreteClass , AbstractClass)
}
```

Sur la figure 11, nous avons introduit l'utilisation du bloc *SuchThat* après un bloc *Pattern*. Comme les classes *AbstractClass* et *ConcreteClass* citées à l'intérieur du bloc *Repeat* ne changent pas, il n'est pas utile de vérifier la condition *subclass(ConcreteClass , AbstractClass)* à chaque itération dans la boucle; on la vérifie une seule fois, la première fois que ces variables sont rencontrées.
La méthode *op*, elle, peut changer, et c'est pourquoi on vérifie à chaque itération qu'elle est différente de la méthode *templateMethod.*

Figure 12: State

```
Pattern State{
   Context.request() on c
   State.handle(c) on s
}
SuchThat{
   field(s , c)
}
```

La seule nouveauté dans l'exemple de la figure 12 est le prédicat *field(o1, o2)*, qui signifie simplement que l'objet *o1* est un attribut de l'objet *o2*.

Figure 13: ChainOfResponsability

```
Pattern ChainOfResponsability{
   Repeat<n>{
        Handler.handle() on o*
   }
   SuchThat{
        o<n> != o<n-1>
   }
}
```

On utilise ci-dessus une version plus élaborée du bloc *Repeat*: le bloc *Repeat<n>;* cela nous permet, dans le bloc *SuchThat* qui le suit, de nous référer aux objets *o<n>* et *o<n-1>*, qui désignent respectivement l'objet que représente la variable *o* à la n-ème et (n-1)-ème itération dans le bloc *Repeat.*

Figure 14: Observer

```
Pattern Observer{
   Subject.m
   Repeat{
        Observer.update
   }
}
SuchThat{
   mutator(m)
}
```

Seule nouveauté: le prédicat *mutator(m)*, qui indique si une méthode modifie l'objet qui l'a invoqué.

Enfin, voici la description du patron de conception *Strategy*.

Figure 15: Strategy

```
Pattern Strategy{
   StrategyContext.m on o1
   AbstractStrategy.strategyMethod on o2
   ConcreteStrategy.strategyMethod on o2
}
SuchThat{
   AbstractStrategy != StrategyContext
   subclass(ConcreteStrategy, AbstractStrategy)
}
```

5 - Conclusion

Dans le cadre de ce projet, nous avons eu à nous documenter sur la description des patrons de conception en génie logiciel. Nous avons lu de manière critique et comparé différents articles traitant de ce sujet, en portant une attention particulière à l'aspect dynamique des descriptions.
Nous avons par la suite exploré différentes pistes pour concevoir notre propre langage. À travers nos tentatives, nous avons retenu les choses suivantes :

- L'utilisation des prédicats, même si elle élimine toutes ambiguïté possible, n'est pas adaptée pour décrire l'aspect dynamique des patrons de conception. De plus, elle donne lieu en général à des descriptions complexes et difficiles à comprendre ou à utiliser.
- On ne peut pas se passer totalement de la programmation. Les approches qui essayent de le faire finissent toujours par la remplacer par des moyens bien plus compliqués.
- Il ne faut cependant pas se placer à un niveau trop bas, sinon on aboutit aussi à des descriptions peu lisibles. L'idéal est donc de trouver le bon compromis, c'est-à-dire le niveau d'abstraction qui permet des descriptions simples mais exhaustives.

C'est ce que nous avons essayé de faire avec notre langage. Le résultat que nous avons obtenu est satisfaisant dans le sens qu'il parvient à décrire plusieurs patrons de conception comportementaux de manière simple et lisible. Ces descriptions résument correctement l'information contenue dans l'ensemble {Schéma de structure + Exemple de code} qui accompagne traditionnellement la définition des patrons de conception.
Le langage obtenu ne nous permet cependant pas de décrire certains patrons de conception comportementaux, à savoir : *Mediator*, *Command*, *Iterator* et *Memento*.
La difficulté de ces patrons de conception est que leur description est principalement sémantique, c'est-à-dire qu'on insiste plus sur leur contexte d'utilisation et leur objectif que sur leur modèle d'exécution. Notre langage, au contraire, permet de spécifier un modèle d'exécution, et c'est ce qui fait qu'il ne permet de décrire ces

patrons de conception. Pour cela, il faudrait un langage capable de définir formellement des notions telles que le sens et l'objectif d'un patron de conception, ce qui dépasse le cadre de notre projet.
De façon générale, ce projet nous aura permis d'acquérir de solides connaissances dans le domaine des patrons de conception, et de nous familiariser avec le travail de recherche.

Références:

[1] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides
Design Patterns, Edition: Hardcover, August 1994

[2] Data & Object Factory
http://www.dofactory.com/patterns/Patterns.aspx

[3] Hervé Abin-Amiot & Yann-Gaël Guéhéneuc,
Meta-modeling Design Patterns: application to pattern detection and code synthesis
http://www.yann-gael.gueheneuc.net/Work/Publications/Documents/ECOOP01AOOSDM.doc.pdf

[4] Jan Bosch,
Design Patterns as Language Constructs
http://citeseer.nj.nec.com/bosch98design.html

[5] Amnon H. Eden, Yoram Hirshfeld & Amiram Yehudai,
LePUS - A Declarative Pattern Specification Language
http://citeseer.nj.nec.com/112816.html

[6] Tommi Mikkonen,
Formalizing Design Patterns
http://portal.acm.org/citation.cfm?id=302175&dl=ACM&coll=GUIDE

[7] Michiaki Tatsubori & Shigeru Chiba,
Programming Support of Desing Patterns with Compile-time Reflection
http://citeseer.nj.nec.com/tatsubori98programming.html

[8] Joshua Engel
Programming for the java(TM) Virtual Machine, Edition: Paperback, June 1999