IFT 3051 – Projet d'informatique.

Identification de microarchitectures similaires aux patrons de conception proposée par le « Gang of Four »

présenté à :

Yann-Gaël Guéhéneuc, Ph.D.& HOUARI A. SAHRAOUI, Ph.D.

Département d'informatique et de recherche opérationnelle

Laboratoire de Génie Logiciel

par :

Duc-Loc Huynh

Janice Ka-Yee Ng

UNIVERSITÉ DE MONTRÉAL

Version révisée Hiver 2005

TABLE DES MATIÈRES

CONTEXTE

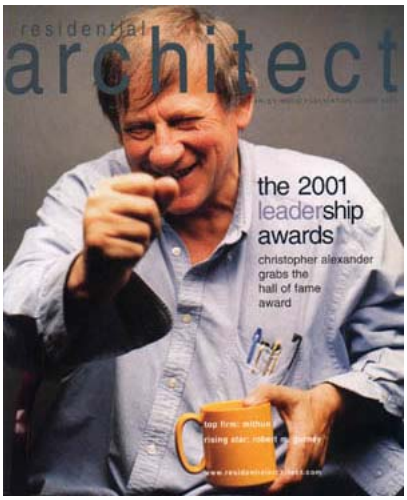
Les langages de programmation ont constamment évolué depuis qu'ils existent. De nos jours, on distingue différents styles de programmation, allant de impératif (Perl), fonctionnel, orienté-objet (Eiffel, C++, Java, …), logique, à concurrent. Les langages de programmation orientés-objet connaissent un essor grandissant depuis les années 80. De plus en plus, les concepteurs bâtissent des programmes complexes où un grand nombre de composants interagissent entre eux en suivant la tendance POO.

Dans cette suite d'idées, peut-on réutiliser un ensemble de classes collaborant pour accélérer le processus de développement de logiciels, pour faciliter la maintenance de programmes complexes, pour promouvoir un bon style de programmation, bref, pour augmenter la qualité d'un programme? C'est ainsi que, dans le but de formuler des solutions générales pour des problèmes qui surviennent de manière récurrente dans la conception des logiciels, la notion de design pattern est naît. Une préoccupation de ce concept réside en la conception de logiciels orientés-objet *réutilisables*. Les « *designs patterns* » encouragent la réutilisation de solutions proposées pour des problèmes rencontrés auparavant par des développeurs !

Christopher Alexander, docteur gradué en architecture, auteur du livre « A Pattern Language », a vu les méthodes et techniques décrites pour être appliquées initialement dans le monde de la construction, migrées vers le domaine de l'informatique par rapport à la programmation orientée-objet. Dans le but idéaliste de permettre à tous les citoyens de construire sa propre maison, il a proposé une définition au mot « pattern » de la manière suivante :

« Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such way that you can use this solution a million times over, without ever doing it the same way twice. […] Each pattern is a three part rule, which express a relation between a context, a problem, and a solution. »

Christopher Alexander, [Alexander, 1977]



Par la suite, cette notion de patrons de conception a été poussée pour être appliquée explicitement dans le domaine de l'informatique. En l'occurrence, Gamma et al ont proposé une collection de patrons de conception à laquelle nous nous référons dorénavant dans le projet.

Mais dans quelle mesure peut-on affirmer que l'utilisation de patrons de conception a permis effectivement d'améliorer la qualité d'un programme ? Le calcul de métriques directes telles que la complexité cyclomatique sur les programmes, et l'utilisation de modèles prédictifs de qualité permettant d'interpréter les résultats des métriques directes, nous permettront de mesurer indirectement la qualité des programmes.

OBJECTIFS

Le travail dont il est question dans ce projet consiste à retracer des patrons de conception que les programmeurs ou « architectes » auraient appliqués, consciemment ou inconsciemment, durant la conception du logiciel. Le choix des programmes à analyser se restreint évidemment à des logiciels écrits avec un langage orienté-objet, plus précisément en Java, puisque nous voulons déterminer l'impact des patrons de conception sur les programmes à objets. Quant aux patrons de conception à identifier, nous nous référons aux vingt-trois (23) motifs que nous ont proposés Gamma et *al* [i]. Bien entendu, il existe dans la littérature de *patterns* une multitude d'autres patrons présentés par d'autres auteurs, mais étant donné un compromis nécessaire entre le degré de difficulté du travail à réaliser et le temps alloué pour le projet, il est amplement suffisant d'examiner seulement la vocabulaire de Gamma et *al*.

```
<microArchitecture number="95">
  <actors>
    <clients>
      <client roleKind="AbstractClass"><entity>com.taursys.html.test.HTMLComponentFactoryTest</entity></client>
    </clients>
    <abstractFactories>
      <abstractFactory roleKind="AbstractClass"><entity>com.taursys.xml.ComponentFactory</entity></abstractFactory>
    </abstractFactories>
    <concreteFactories>
      <concreteFactory roleKind="Class"><entity>com.taursys.html.HTMLComponentFactory</entity></concreteFactory>
    </concreteFactories>
    <abstractProducts>
      <abstractProduct roleKind="AbstractClass"><entity>com.taursys.xml.Component</entity></abstractProduct>
    </abstractProducts>
    <products>
      <product roleKind ="Class"><entity>com.taursys.xml.DispatchingContainer</entity></product>
      <product roleKind ="Class"><entity>com.taursys.xml.DocumentElement</entity></product>
      [...]
   </products>
  </actors>
  <comment>
    ComponentFactory may be extended as specialized XML documents are needed. Therefore, it is possible
    to create our own extension of CompenentFactory as to generate Components for specialized XML document,
    just like " HTMLComponentFactory is used to automate the creation of Components based on the HTML Document
    and its Elements. "
  </comment>
</microArchitecture>
```

Figure 1. Représentation d'une micro-architecture identifiée dans le programme MapperXML similaire au patron *Abstract Factory*.

Nous compilons l'ensemble de résultats dans un fichier XML. Chaque motif identifié est présenté sous forme d'une micro-architecture, qui possède une nomenclature définie par le responsable de projet au début du travail. Voici une illustration d'une micro-architecture similaire à l'application du *design pattern* « Abstract Factory » :

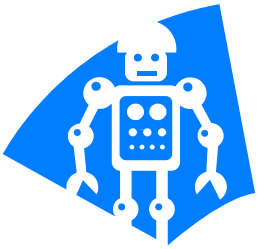
Outre que de fournir des données aux futurs étudiants qui étendront ce sujet vers un niveau plus haut dans l'analyse de l'impact des patrons de conception sur les programmes à objets par le calcul de métriques directes sur les programmes analysés, l'intérêt principal de notre travail est de pouvoir proposer une méthodologie d'identification de patron de conception dans un programme donné. Ainsi donc, nous avons développé diverses techniques durant notre chasse aux patrons, l'une plus fiable que l'autre, que nous allons tenter d'exposer dans les quelques pages qui suivent. Nous essayons de proposer une méthodologie pouvant contribuer, d'une manière ou d'une autre, à automatiser l'identification de patrons de conception dans les logiciels.

En résumé, voici les objectifs qui entourent ce projet :

- Analyser l'impact des patrons de conception appliqués sur les programmes à objet par le calcul de métrique et les modèles de qualité *(réalisable par une autre équipe d'étudiants)*;
- Automatiser l'identification de patrons de conception *(contribution en partie seulement, puisque notre niveau de connaissance n'est pas suffisamment avancé dans ce projet pour proposer une solution à l'automatisation, si celle-ci est possible)*.
- Proposer des recettes afin d'identifier les patrons de conception.

Analyse

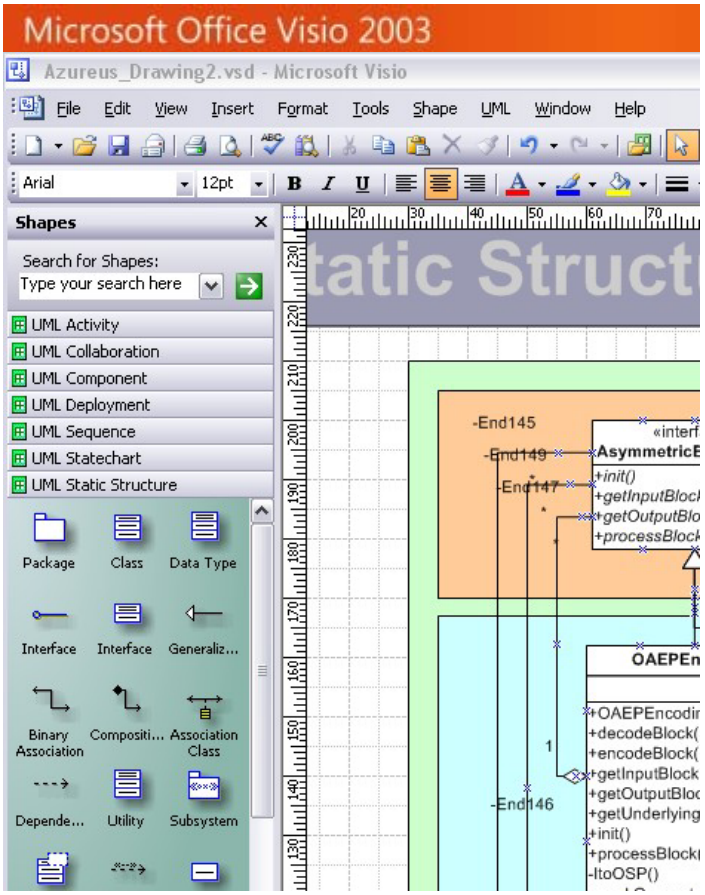
Automatisation

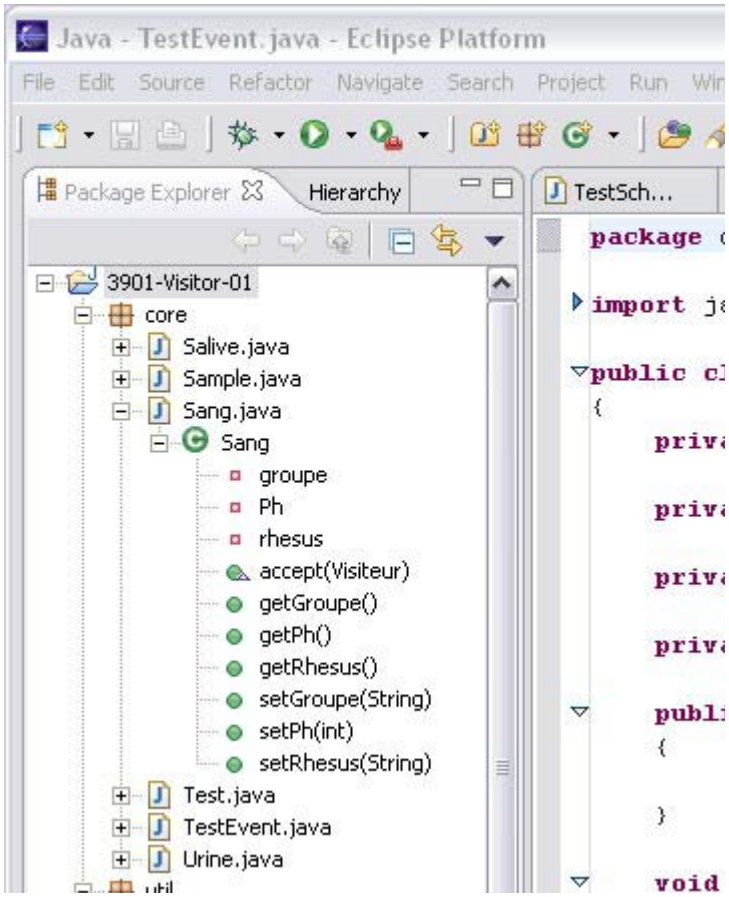


Recette

ENVIRONNEMENT DE TRAVAIL

Le travail que nous avons effectué a été essentiellement réalisé à travers l'environnement de développement pour Java, fourni par la plate-forme Eclipse. De plus, l'emploi de l'outil Visio offert par Microsoft s'est avéré utile au cours de notre projet.





MÉTHODOLOGIE

Dans les quelques pages qui suivent, nous allons tenter d'exposer les trois méthodologies que nous avons développées au cours de ce projet, afin de pouvoir proposer des recettes simplifiant le processus d'identification de patrons de conception dans des logiciels complexes. De plus, les descriptions sont agrémentées de nos critiques. Nous remarquerons qu'il existe un progrès, une évolution constante, dans la description d'une méthodologie par rapport à la précédente, au fur et à mesure que nous devenons plus à l'aise avec cette tâche difficile que représente le retraçage de patrons de conception.

Nous sommes tous deux étudiants au baccalauréat en informatique, orientation génie logiciel. Ainsi, nous avons déjà assimilé certaines notions des patrons de conception présentées dans les cours de base, soient IFT2251, IFT3901 et IFT3902. Cependant, étant donné que la portée de ces trois cours ne nous ont pas permis d'appliquer les concepts théoriques enseignés dans des cas pratiques de l'ampleur telle que celle visée par le projet, nous pouvons affirmer que le défi qui nous est lancé en choisissant ce projet n'est pas négligeable.

Initialement, notre expérience face à la reconnaissance des patrons de conception est qualifiée de quasi nulle. Ainsi, une recherche d'informations sur l'Internet nous paraît être la première ressource disponible. Cependant, nous nous sommes rendus compte très rapidement que la littérature des patrons de conception n'était pas unique dans leur domaine. Par exemple, deux mêmes noms pouvaient désigner un même patron de conception. Par conséquent, il nous est plus approprié de nous convenir sur une seule nomenclature, et y faire référence pour le reste du projet, qui, en l'occurrence, se trouve à être celle définie par Gamma et al dans « *Design Patterns - Elements of Reusable Object* ». Cet ouvrage est le recueil le plus complet et détaillé sur lequel nous sommes tombé au moment où notre projet a lieu. Il constitue notre livre de référence par excellence : c'était notre phare d'Alexandrie. Les autres références qui sont disponibles dans la bibliographie de l'énoncé du projet nous ont également servi à clarifier et à structurer notre travail[ii].

Dans ce qui suit, nous allons décrire les trois méthodologies que nous avons acquises et développées durant le projet, en constante évolution selon une période précise : -

3 *FIN DU PROJET*

2 *MILIEU DU PROJET*

1 *DÉBUT DU PROJET*

➢ Méthodologie N° 1 – Début du projet

Description

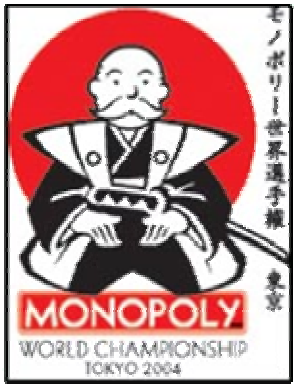
De façon indéterministe, nous avons commencé à l'ancienne !!

1. C'est-à-dire, que nous nous sommes fiés aux noms de classes pour définir leur appartenance à une structure de patron ;
2. Papier et crayon pour mettre en évidence les relations inter-classes ;
3. Établissement des correspondances entre les diagrammes obtenus et ceux du livre ;
4. Une fois un patron identifié, nous nous référons au diagramme des relations entre patrons de conception dans l'espoir de trouver d'autres bêtes ;
5. Mise à jour du XML pour des fins de comparaisons entre nous ;
6. Retour à l'étape 1.

Programmes analysés

Étant donné qu'à cette étape du projet, nous n'en sommes qu'à notre premier contact flou avec les patrons de conception , le choix d'un logiciel Java en contenant représentait un défi pour nous. Lequel risquerait de contenir des patrons de conception ?

Nous avions décidé d'attaquer, pour notre premier programme, un logiciel de taille relativement petit - **`Java Monopoly`** – d'environ 20 classes. La raison première qui nous avait poussé à choisir un logiciel de telle taille était que nous pensions pouvoir faire une analyse rapide de ce dernier, afin de vérifier avec notre responsable que nous étions sur la bonne voie. En effet, il serait plus facile de comprendre la structure du code et le fonctionnement du logiciel, et ainsi donc, retracer plus efficacement des patrons de conception s'ils sont effectivement présents.



Résultat : échec total, nous n'avons rien trouvé.

Explications / Suppositions : -

- Le logiciel **`Java Monopoly`** a été écrit par un programmeur qui n'avait pas l'intention ni la contrainte de prévoir des mises à jour.
- Le logiciel est trop petit pour nécessiter, de façon critique, une structure stricte.

Suite à ce petit échec, nous nous sommes orientés vers une recherche plus rigoureuse et fiable de programmes à objets, via http://www.sourceforge.net, un site contenant un très grand nombre de logiciels avec code source ouvert. La recherche du programme était essentiellement basée sur les mots clés *patterns* et Java. Ainsi donc, nous nous sommes lancés dans l'analyse d'un programme de taille (relativement) plus grand - **`Mapper XML`** – d'environs 200 classes. Cette fois, nous nous sommes fiés à la documentation, qui nous assurait que les programmeurs se sont forcés à suivre les règles des patrons de conception.

Résultat : avec un logiciel de telle taille, nous avons réussi à identifier 15 micro-architectures similaires aux motifs de design. Mentionnons que l'identification des motifs n'est pas

Explications / Suppositions : le nombre de micro-architectures identifiées s'est nettement amélioré par rapport à notre première recherche dans **`Java Monopoly`**. Ceci peut être expliqué en partie par un plus grand nombre de classes dans le logiciel **`Mapper XML`**, et surtout par l'application explicite de la notion de patrons durant la conception du programme. Selon nous, un programme plus grand nécessite une structure plus claire, afin de faciliter la maintenance. Ainsi donc, les motifs voient bien leur place dans le logiciel.

Commentaire

À l'étape 1

Nous mentionnons le nom des classes qui peuvent être un indice de la présence de patrons de conception. Cette technique de faire repose essentiellement sur la volonté des programmeurs à identifier le nom des classes du logiciel selon la structure du patron de conception, si le cas s'applique. En voici un exemple tiré de nos résultats :

Ici, nous avons tout d'abord repérer une classe dont le nom invoque la présence d'un patron de conception. Selon une déduction qui suit l'instinct naturel, la classe *ComponentFactory* suggère l'utilisation d'une « usine » pour produire des objets *Components*, d'où le nom *CompenentFactory*. Les patrons Abstract Factory et Factory Method nous viennent en tête alors. C'est ainsi que, à partir de cette classe, nous avons cherché les classes qui y sont reliées, pour ensuite tracer les liens entre elles. Enfin, à partir du diagramme obtenu, on essaie d'établir une correspondance avec la structure du(es) patron(s) visé(s) initialement.

Figure 2. Illustration de l'application de la méthodologie n°1.

À l'étape 2

Après avoir sélectionné la classe de départ, nous en déterminions les relations inter-classes (figure 3), telles que les agrégations, héritages, etc.

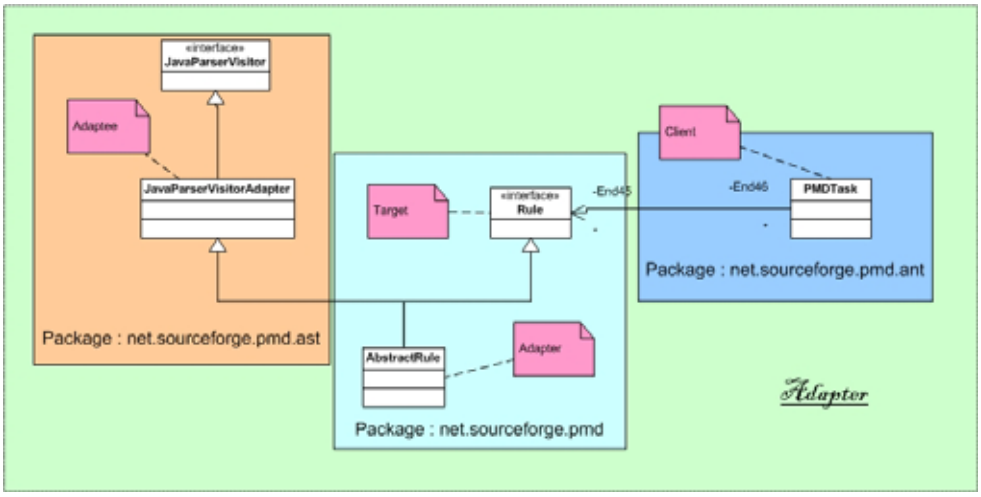


Figure 3. Mise en évidence des relations inter-classes à partir d'une classe cible.

À l'étape 3

Sans aller trop loin, nous avons essayé de calquer la forme des diagrammes obtenus et ceux du livre. Un peu comme les puzzles pour enfants, où le but était de trouver la bonne forme.

À l'étape 4

Une fois un motif repéré, nous sommes alors orientés pour la recherche d'un autre ! Le fait de tomber sur une autre mico-architecture du programme est moins un jeu de hasard. En effet, grâce au schéma d'interrelations entre les différents patrons de conception proposé par Gamma et al, nous sommes avertis des divers liens possibles. En l'occurrence, pour nous référer à la micro-architecture de la figure 2, nous avons réussi à la repérer une fois que le patron *Abstract Factory* est reconnue dans la structure du programme. Clairement, nous avons gagné en terme de temps !

À l'étape 5

Nous avons sûrement oublié de vous préciser que toutes les recherches ont été fait chacun de notre côté. Cela est dû à notre façon de penser et de travailler assez opposée l'une de l'autre. Nous sommes tellement bornés que nous aurions passé des jours à argumenter pour qu'à la fin nous rendre compte que nous parlions de la même chose. En fait ç'était arriver deux ou trois fois (peut-être quatre …). C'est pourquoi nous avons préféré confronter nos trouvailles que lorsque celles-ci étaient complètes.

En conclusion, pour ce qui est de l'applicabilité de la méthodologie n°1, cette dernière n'est fiable que sous la mesure où les auteurs de logiciels explicitent clairement l'utilisation de patrons de conception pour factoriser le programme en motifs similaires au *design patterns*. Par la suite, une vérification rigoureuse doit être effectuée, une fois la classe candidate ciblée, afin d'établir la correspondance exacte avec la structure du motif en question en établissant le diagramme de classe.

➢ Méthodologie N° 2 –Milieu du projet

Description

Cette fois avec une idée de ce que nous recherchons :

1. Recherche des interfaces et classes abstraites en fonction des relations entre patrons.
2. Établir les relations inter-classes à partir de ces dernières.
3. À partir des noms de classes ainsi que de leurs méthodes, nous pouvons nous risquer à supposer l'appartenance de ces structures à un groupe de patrons.
4. Ensuite par la silhouette du diagramme, nous comparons les patrons du groupe afin de déterminer LA créature.
5. Puis pour des fins de confirmation, nous observons les relations inter-méthodes.

   [À ce point nous sommes assurés à 99% de l'exactitude du patron.]

6. Mise à jour du XML pour des fins de comparaisons entre nous.

Programmes analysés

Cette fois, étant plus confiant et surtout grâce aux patrons trouvés précédemment, nous avons « sauté » dans les analyses, sans vraiment nous soucier de la présence ou non de patrons. En fait, nous partions de l'hypothèse que plus un logiciel est grand, plus il devra être structuré (entre autre pour débuguer). Et ce même si les programmeurs n'avaient pas prévu d'émettre des mises à jour.

Voici donc nos deux prochains défis, **PMD** et **Nutch** : logiciels de taille moyenne (environ 200 classes).

http://pmd.sourceforge.org/

http://www.nutch.org/

Au cours de nos analyses, nous avons observé (en fait nous avons plongé en plein dedans), une difficulté de plus dans notre tâche d'identification. Cela réside en la présence de sous-classes. Ainsi, cette difficulté nous a forcé d'aller encore plus profondément dans nos analyses de structure.

Commentaire

À l'étape 1, le déclique !

Au cours de nos précédentes analyses, nous avons pu observer des types d'objets qui revenaient très souvent. Nous voulons parler des interfaces et des classes abstraites. En fait ce sont ces deux éléments qui constituent la base principale des patrons de conception. Grâce à eux, le code se voit doter de règles (… de « templates » …) qui obligent, ou plutôt, qui guident les programmeurs dans une compréhension plus fine et surtout plus rapide des algorithmes et structure du code. Pour vous donner une idée, une classe qui crée des algorithmes d'encodages ne se mettra pas à créer des formes géométriques. Ainsi, nous n'avons plus besoin de chercher de midi à quatorze heures !

À l'étape 2, un « cheval » qui gagne !

Comme précédemment, le fait de dessiner les diagrammes nous a grandement aidé dans nos analyses. C'est pourquoi nous avons continué dans ce sens.

À l'étape 3, optimisation des comparaisons !

Une fois les diagrammes dessinés, nous pouvons passer à la phase « Identification ». Cette phase consiste toujours à comparer les diagrammes obtenues et ceux proposés par le GoF. Mais afin d'optimiser notre temps de recherche, nous n'avons considéré que les patrons d'un des trois groupes (créationnel / structural/ comportemental).

Notre choix du groupe a été en fonction du nom des classes ainsi que du nom des méthodes. En fait, nous avons considéré l'étymologie des noms. Ainsi, lorsque nous rencontrons un nom contenant le verbe « créer » ou bien un de ces synonymes, nous pouvons affirmer, dans 80% des cas, la présence d'un patron du groupe créationnel.

Donc, en supposant que les personnes qui ont codées le logiciel ont été assez logiques dans le choix des noms de classes, ainsi que ceux des méthodes, nous avons pu émettre certaines certitudes concernant le groupe de patrons évoqué.

Voici des exemples montrant l'application concrète de l'analyse sémantique des objets (classes et méthodes) :

- *Par nom de classe* : à la figure 7 et 8, nous avons choisi les classes *SymbolFacade* et *NodeIterator* pour le nom qu'elles portent. À partir de celles-ci, nous avons recherché les classes reliées pouvant mener à la confirmation de la présence du motif « Facade » et « Iterator ».
- *Par nom de méthode* : référons nous à la figure 5 attaché en annexe. Dans la classe jouant le rôle de *Director*, il existe une méthode *createRenderer* qui crée les objets appropriés ayant une structure commune (interface *Renderer*).

Pour aller plus loin, nous avons vérifié les relations inter-méthodes pour les figures ci-dessus. Ceci nous a permis de préciser nos résultats.

À l'étape 4, moins de comparaisons, mais comparaisons quand même !

Par la suite, nous avons comparé les patrons un par un comme décrit dans la première méthodologie. Mais là, nous n'avons plus à faire une vingtaine de comparaisons pour chaque diagramme.

Et lorsque le patron n'a pas été trouvé, suivant notre humeur et notre disponibilité, nous choisissons ou non de comparer les patrons restant.

À l'étape 5, la qualité avant tout !

Jusqu'à présent, dans nos analyses, nous nous sommes fiés au nom des classes et de leurs méthodes afin de spéculer sur leur appartenance ou non aux différents patrons de conception.

Maintenant, afin d'augmenter notre certitude, nous vérifions aussi le comportement inter-méthodes : -

- si des méthodes retournent la classe elle même.
- si des méthodes retournent d'autre classe.
- si des méthodes ont besoins des ressourcent d'une autre classe.
- si des méthodes, de classes différentes, interagissent entre elles.
- ...

En faisant cela, nous avons pu confirmer l'existence d'un patron, non plus par sa structure physique, mais plutôt par sa structure logique.
Grâce à cela, nous avons pu observer des patrons assez inusités ... présentant des divergences au niveau physique, mais dont la logique est la même : -

- absence d'interface ;
- présence de classes intermédiaires entre les relations ;
- fragmentation de classes ;
- ...

À l'étape 6, les critiques !

C'est l'étape la plus intéressante (à notre point du vu). Car à cette étape, nous présentons et argumentons nos résultats. Ainsi, nous pouvons faire une mise à jour de notre méthodologie en prenant les meilleurs points utilisés.

➢ Méthodologie N° 3 – Phase finale du projet

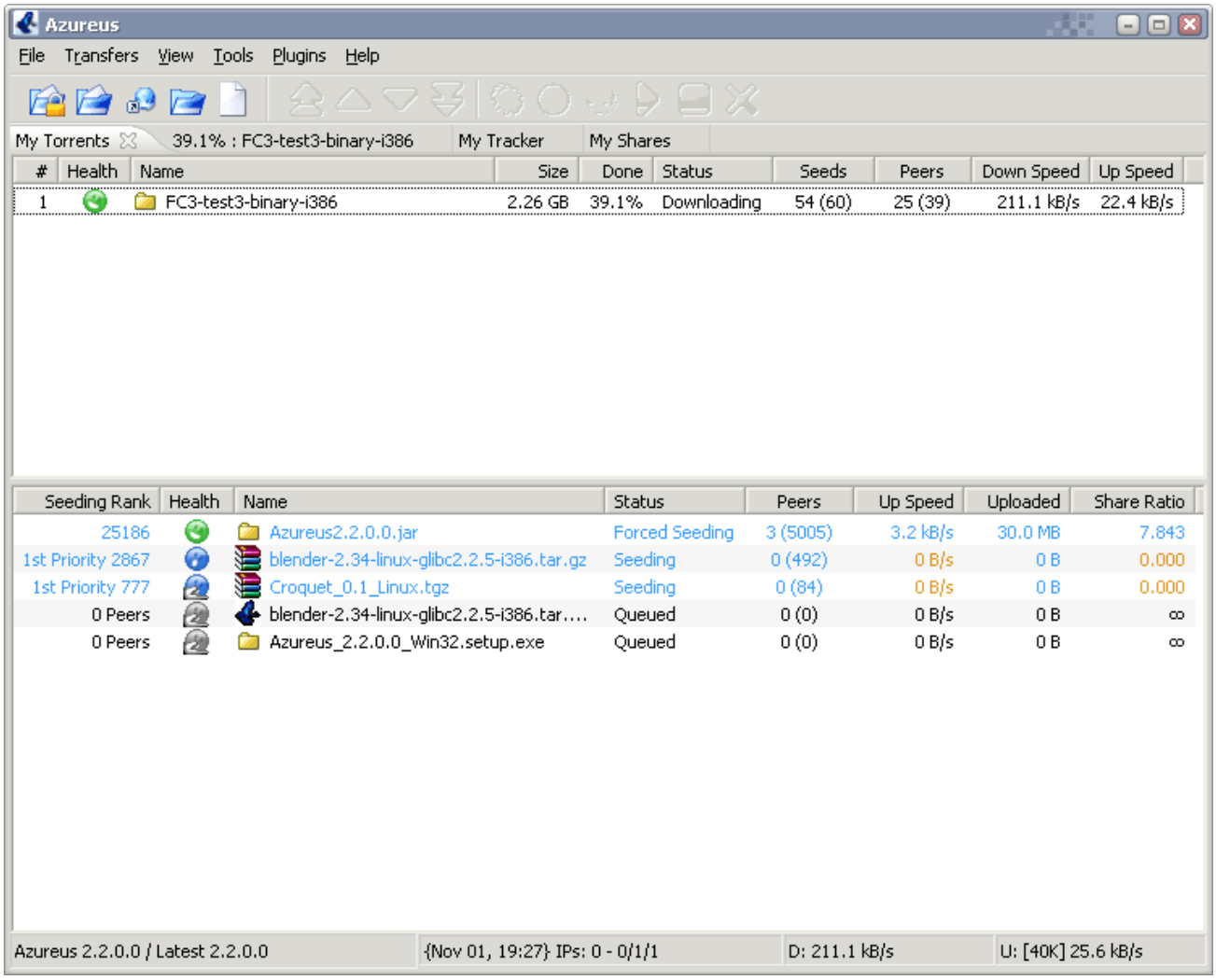
Description

La méthodologie qui est présentée à cette étape du projet est très similaire à celle décrite dans la méthodologie n°2.

1. Repérer une interface et/ou une classe abstraite ;
2. Identifier les classes qui y sont reliées et tracer les relations inter-classes : c'est le diagramme de classes ;
3. Si le diagramme dessiné inspire la présence d'un motif, alors, à partir des noms de classes et de leurs méthodes, présélectionner déjà le groupe de patrons de conception évoqué - créationnel, structural ou comportemental ;
4. Établir la correspondance exacte ou similaire avec LE motif qui lui ressemble le plus dans la catégorie choisie ;
5. Analyser le comportement inter-méthodes pour les méthodes clés du patron de conception en jeu ;
6. Mise à jour du fichier XML pour des fins de comparaisons entre nous.

Programme analysé

Sans doute, le programme analysé dans cette étape du projet reflète le sommet ultime de notre projet ! **`Azureus`** est composé d'environ 1300 classes. Pourquoi l'avoir choisi ? Pour nous poser le plus grand défi possible et imaginable durant ce projet, et pour satisfaire notre soif de motifs dans cette chasse aux patrons de conception …



Commentaire

La méthodologie qui est proposée ici est très similaire à la méthodologie nº2. La différence remarquable est indubitablement le nombre de fois où nous avons dû consulter le livre de GoF afin de vérifier la correspondance entre les deux diagrammes dont il est si souvent en question dans les trois méthodologies. En effet, à ce niveau, la structure des patrons de conception qui sont revenus le plus souvent durant notre travail est bien imprimée dans notre tête. Évidemment, pour les motifs qui sont moins fréquents, tels que Interpretor, Flyweight et State, le livre de référence nous est toujours utile.

Étant donné l'ampleur du programme **Azureus** par rapport aux trois logiciels précédents, l'introduction de Visio à cette étape-ci de notre projet nous paraît être d'une grande utilité. En effet, les diagrammes de classe que nous dessinons à la main alors pour les autres logiciels étaient encore de taille raisonnable.

Mais pour **Azureus**, pour bien observer les relations entre les classes et par conséquent identifier rapidement les motifs présents, nous étions tentés par l'outil de dessin que nous offre Visio. La visualisation des relations inter-classes est permise à une échelle plus large, et par conséquent, cet outil nous permet d'avoir une meilleure vue de la conception, de la structure du programme complexe qu'est **Azureus**. (À mentionner qu'il est également possible d'obtenir une telle visualisation sous Eclipse. Le choix des outils est tout à fait personnel …)

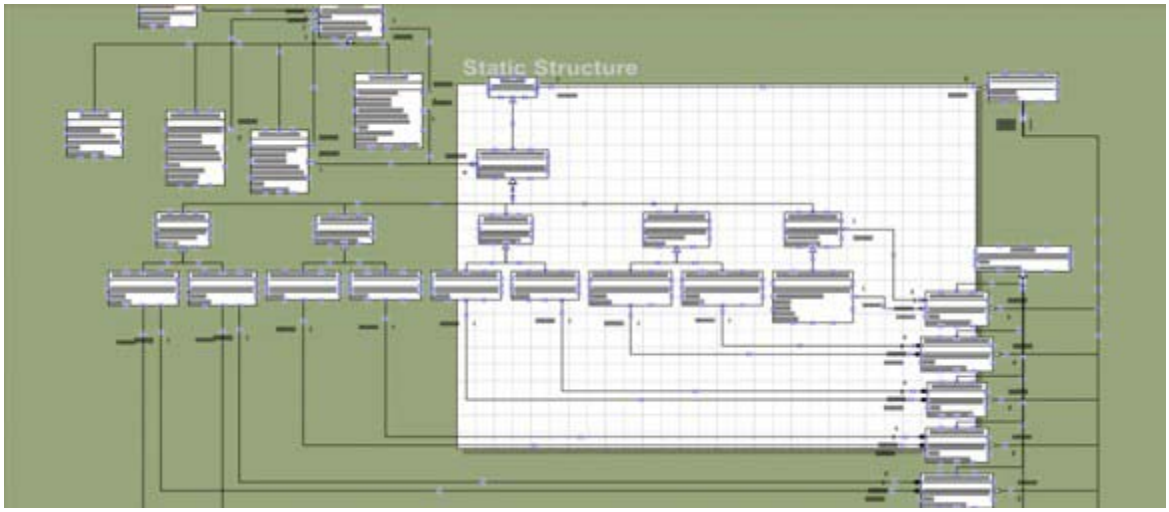


Figure 4. Diagramme de classe brouillon réalisée sous Visio.

Sans entrer dans les détails, nous avons mis ici la figure 3 pour refléter la complexité que peut prendre le programme. Il est alors important de pouvoir bien visualiser les relations inter-classes à un niveau plus général afin de pouvoir repérer les motifs que nous aurions pu manquer si l'analyse a été réalisée à un niveau microscopique. Il est alors très probable que dans un diagramme de telle complexité, nous parvenons à identifier plusieurs motifs candidats.

Dans le processus d'identification de motifs, il est très fréquent pour une structure de correspondre à un pattern sans que tous les acteurs, proposés par GoF, soient présents (c.f. figure 6).

En conclusion, notre sens critique envers la notion de patrons de conception est indéniablement bien aiguisée pour pouvoir confirmer / renier la présence d'un patron très rapidement par la silhouette d'un diagramme de classe et par le comportement des méthodes clées. Évidemment, notre évaluation ne peut être juste à 100%, étant donné notre expérience qui ne date que de quelques mois. Mais, la maîtrise du concept de patrons de conception est de plus en plus claire.

RÉSULTATS

Comme nous avons mentionné plus tôt, les résultats sont compilés dans un fichier XML qui est remis à notre responsable de projet. Ici, nous allons mettre le compte total des micro-architectures identifiées pour chaque logiciel analysé.

Tableau I. Résumé.

Mapper XML		Nutch		PMD		Azureus	
Abstract Factory	1	Adapter	2	Adapter	1	Bridge	1
Adapter	2	Bridge	2	Builder	2	Factory Method	1
Composite	1	Command	2	Composite	2	Flyweight	1
Facade	1	Iterator	1	Facade	1	Interpreter	2
Observer	1	Memento	2	Factory Method	3	State	1
Factory Method	1	Singleton	1	Iterator	1	Template Method	1
Singleton	3	Strategy	2	Proxy (1)	1	Visitor	1
Strategy	1	Template Method	3	Template Method	1		
Template Method	4			Observer	2		
				Visitor	1		

ÉVOLUTION ET COMMENTAIRE PERSONNELS

Janice Ka-Yee Ng

À la phase initiale du projet où je n'avais pas encore eu d'expérience pratique avec les patrons de conception, je croyais avoir saisi les notions bien enseignées durant les cours théoriques du baccalauréat. Je comprenais le principe des designs patterns et les objectifs qu'ils visent, sans pour autant être capable de visionner clairement toutes ces règles appliquées à l'échelle d'un logiciel complexe. Maintenant, suite à quelques mois de manipulation avec ces structures, ma vision a été complètement chamboulée. Indéniablement, j'ai pu développer une certaine maîtrise face au concept des *designs patterns* et de leurs caractéristiques.

À prime abord, il n'était pas évident de distinguer le rôle des vingt-trois patrons présentés par GoF. Une première lecture du livre ne fait qu'ajouter de la confusion. Vingt-trois motifs à digérer en un seul coup ! C'est du travail… Pourquoi pas nous aventurer réellement dans du code !!! C'est ainsi qu'ont débuté mes péripéties cet été !

L'outil qui nous a été proposé – Eclipse – n'est pas approuvé par tous les membres de l'équipe, mais je me suis bien contentée de cette plate-forme, avec laquelle je me suis familiarisée très rapidement. Avec le plugin d'Omondo, Eclipse offre la possibilité de tracer divers diagrammes à partir du code source, ce qui nous permet d'épargner du temps lorsque nous voulons tracer les diagrammes de classes. C'est un cas particulièrement intéressant pour les logiciels complexes. Évidemment, le diagramme généré n'est pas juste à 100%, d'où des vérifications rigoureuses nécessaires (le raisonnement humain est plus fiable ;-)), mais la base nous est fournie. Après tout, Eclipse est plus agréable à utiliser que Rational Rose, le premier outil qui m'avait été imposé pour réaliser les devoirs en génie logiciel (dans le bon vieux temps !).

Au moment où je rédige ce rapport, je peux confirmer que le projet a grandement contribué à l'enrichissement de mes connaissances, tant au niveau théorique que personnel.

Bien entendu, le travail d'équipe et les conflits entre personnels reliés reflètent déjà un risque considérable qui est survenu au début du projet. Rappelons-nous que nous étions quatre étudiants, lorsque l'équipe a été initialement formée. Nous ne nous connaissons à peine lorsque nous nous sommes réunis pour la première fois. Nous n'avons jamais réalisé de projets ensemble auparavant. Des oppositions d'idées ne sont pas rares tout au long du projet, mais je crois personnellement que les débats et discussions suscités ont contribué énormément à mon apprentissage dans le domaine précis des patrons de conception, et à élargir mes horizons à propos de ces derniers, et finalement, à développer des méthodes de travail plus rigoureuses.

Un autre risque qui est venu s'imposer est bien l'échéancier. Après quelques semaines écoulées de notre calendrier prévu initialement, un membre de l'équipe a dû quitter pour des raisons personnelles. Ainsi donc, nous étions confrontés à des problèmes de délais d'échéancier. Notre inexpérience face à la gestion de risques a fait sentir ses répercussions négatives. Les tâches qui étaient alors déléguées à ce membre doivent être redistribuées ou coupées, ce qui a perturbé (pas de manière définitive) le déroulement du projet.

Bref, la gestion des risques m'est encore une activité bien fraîche. Les risques que nous rencontrerons dans les entreprises seront d'autant plus grandes et plus sérieuses. Cette expérience du projet réalisé en équipe m'a bien donné un avant goût de ce qui nous est réservé ;-)

En conclusion, l'acquisition du concept des patrons de conception et de leur domaine d'application n'est pas donnée. Elle se développe graduellement avec l'analyse de logiciels, le plus grand nombre étant le plus favorable, mais surtout en restant ouvert aux points de vue des autres et à des discussions par rapport aux divergences de pensées. L'identification de motifs est une tâche qui requiert une attention particulièrement élevée, surtout lorsque la micro-architecture n'est pas exactement telle que définie dans les définitions théoriques des *design patterns*. Par exemple, les cas les plus fréquents sont les suivantes :

- les rôles des acteurs dans la structure originale n'ont pas tous été nécessairement remplis par une classe du logiciel analysé ;
- le comportement d'une méthode clée d'un acteur impliqué dans la micro-architecture diffère un peu de celui présenté théoriquement.

C'est à ces moments là que l'analyse des patrons de conception s'avère intéressant ! Comment modifier le code – la ré-ingénierie – afin de pouvoir faire correspondre exactement les deux versions ? La ré-ingénierie est un tout autre domaine d'étude du génie logiciel, à la fois intéressant et complexe.

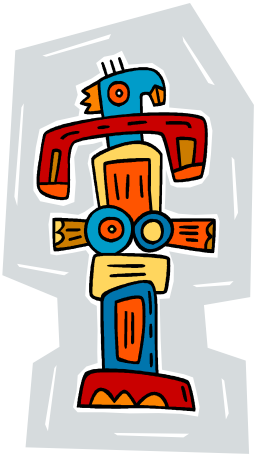
L'emploi abusif des patrons de conception dans les programmes est un piège dans lequel plusieurs développeurs tombent souvent. Ainsi donc, avant de décider de mettre des motifs par ci par là, il faut bien analyser les besoins et les spécifications du logiciel afin de faire valoir au maximum l'utilité des design patterns !

Duc-Loc Huynh

Cette page contient quelques conseils personnels, de moi à vous, qui reprenez le flambeau de la recherche et analyse des patrons de conception.

Mon premier conseil, qui est aussi le plus important est de ne PAS utiliser les logiciels « magiques »! Ces logiciels nous promettent de faire l'analyse des relations interclasses et d'identifier « automatiquement » les patrons de conception.

Il y a plusieurs raisons à cela :



- Si ces logiciels existaient … ben que feriez-vous dans ce projet ??!!
- Certains logiciels donnent des diagrammes assez convenables … quand ils ne sont pas trop complexes. Mais si vous rencontrez des diagrammes plus complexes… qui les feraient pour vous ?!
- Rien de tel que de commencer à la main (crayon et papier), à la limite avec Visio (par souci de l'environnement…) ainsi vous assimilerez, petit à petit, l' « Esprits de Patrons de Conceptions ».
- Dernière raison … vous allez voir, c'est bien plus fun de le faire à la main … vous pourrez ainsi critiquer le code au fur et à mesure, en vous disant … « hmm, j'ai enfin trouvé pire que moi en programmation ». Vous verrez, c'est très gratifiant. ☺

Mon deuxième conseil c'est la communication. Il est important d'échanger les résultats afin d'avoir une vue externe et critique sur nos propres résultats.

Les raisons :

- Il n'existe pas de règles Empiriques en ce qui concerne les Patrons (pas pour l'instant)
- L'identification de Patrons ne se fait pas seulement au niveau structural … « 2 Interfaces + 5 Classes concrètes == tel Patron ! ». Par expérience, le niveau structural ne fournit qu'un indice sur la présence ou non des patrons. Le niveau logique (ou ce que j'appelle personnellement le niveau Psychologique) est celui qui nous permet vraiment de dire si tel ensemble de classes forme ou non un patron. Et croyez moi, au cours des analyses, vous verrez des « vertes et des pas mûres ».

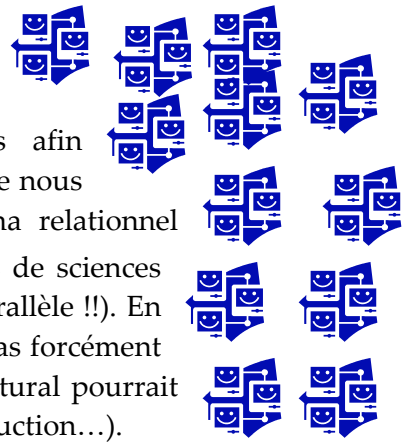
Troisième conseil harceler les professeurs de GELO du département !

Les raisons :

- Pas la peine de chercher sur Internet … c'est plus une décharge publique qu'autre chose. Où les bonnes idées côtoient les mauvaises. Ici, nous avons des « humains » à harceler et où le mot « Pourquoi » ne s'utilise pas dans le vide !
- Raison moins noble … se seront eux qui vous donnerons, ou influerons, votre notes. Donc si vous êtes sur la mauvaise voie se sera de leur fautes ! :-p

Quatrième point, si vous êtes à plusieurs, sur le projet, ne pas se partager les groupes de patrons.

Les raisons :



- Au tout début, nous pensions aussi que se serait une bonne idée que chacun recherche les patrons d'un seul des trois groupes afin d'optimiser le temps de recherche. Mais ce que nous n'avions pas pris en compte c'est le schéma relationnel proposé par le GoF. (Je précise pour les fans de sciences fictions, ce n'est PAS une carte de mondes parallèle !!). En fait un patron peut en cacher un autre … et pas forcément du même groupe. (Exemple : un Patron Structural pourrait servir à structurer plusieurs Patrons de Construction…).
- Pourquoi, de façon délibérée, ne considérer qu'un certain nombre de Patrons. Il n'y en a qu'une vingtaine, et comme on dit, « Il faut de tout pour faire un monde » !

Cinquième et dernier point, … en fait il n'y pas de cinquième point, c'est seulement pour vous souhaiter bonne chance dans votre projet !!

Et surtout de ne pas oublier ceci. Bien que ce projet fasse partie des travaux les plus abstraits (et surtout « plate ») que l'on propose au GELO, il n'en reste pas moins que le concept des Patrons de Conception est très important.

Aussi, … comme les règles ne sont pas explicitement définies, il n'en tient qu'à vous de rendre se sujet plus Fun !!

Voilà de la part de la première équipe de Patrons de Conception…

Bonne Chance! and Have Fun!

ANNEXES

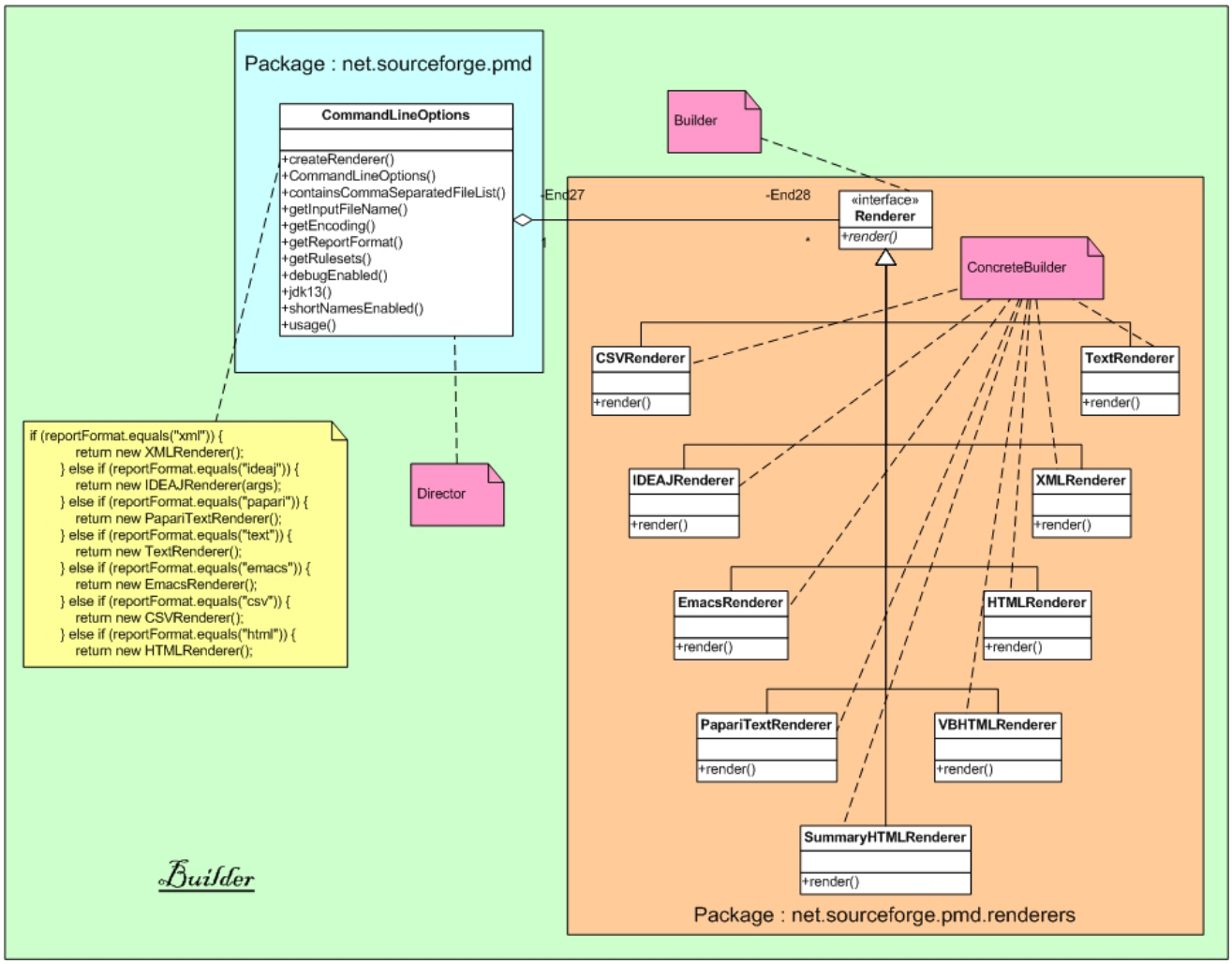


Figure 5. Illustration d'un diagramme de classes similaire au motif « Builder ».

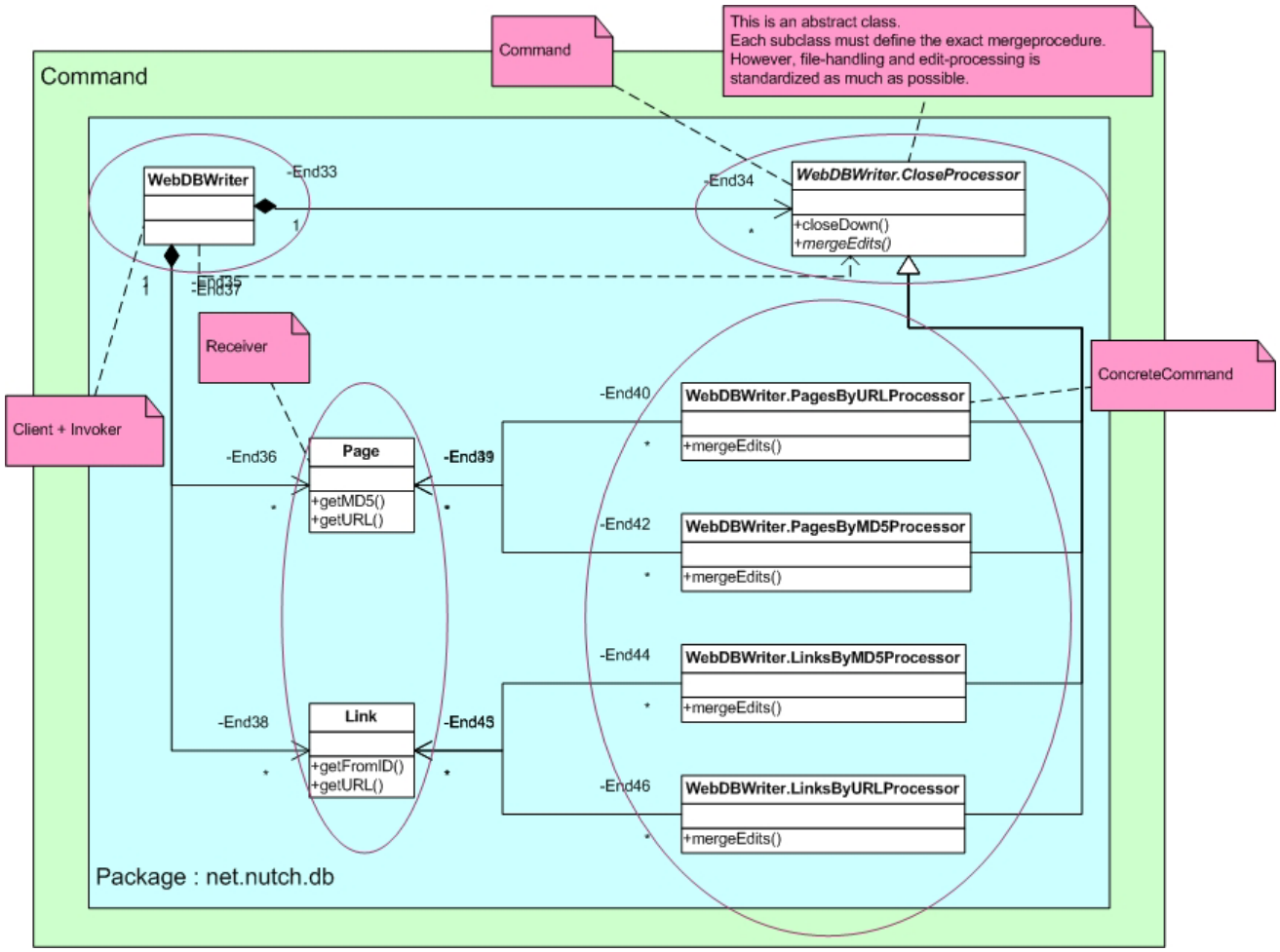


Figure 6. Illustration d'un diagramme de classes similaire au motif « Command ».

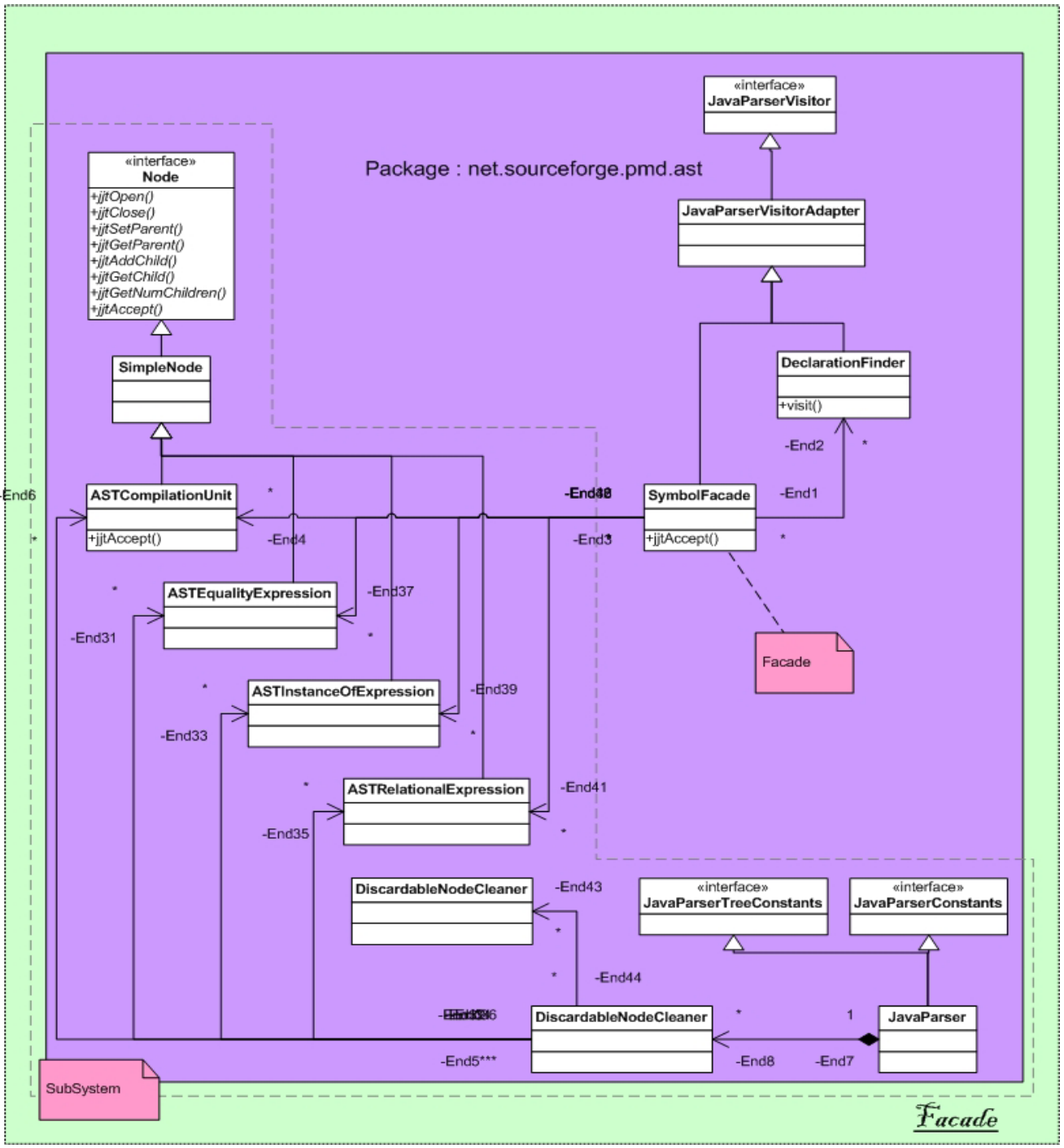


Figure 7. Illustration d'un diagramme de classes similaire au motif « Facade ».

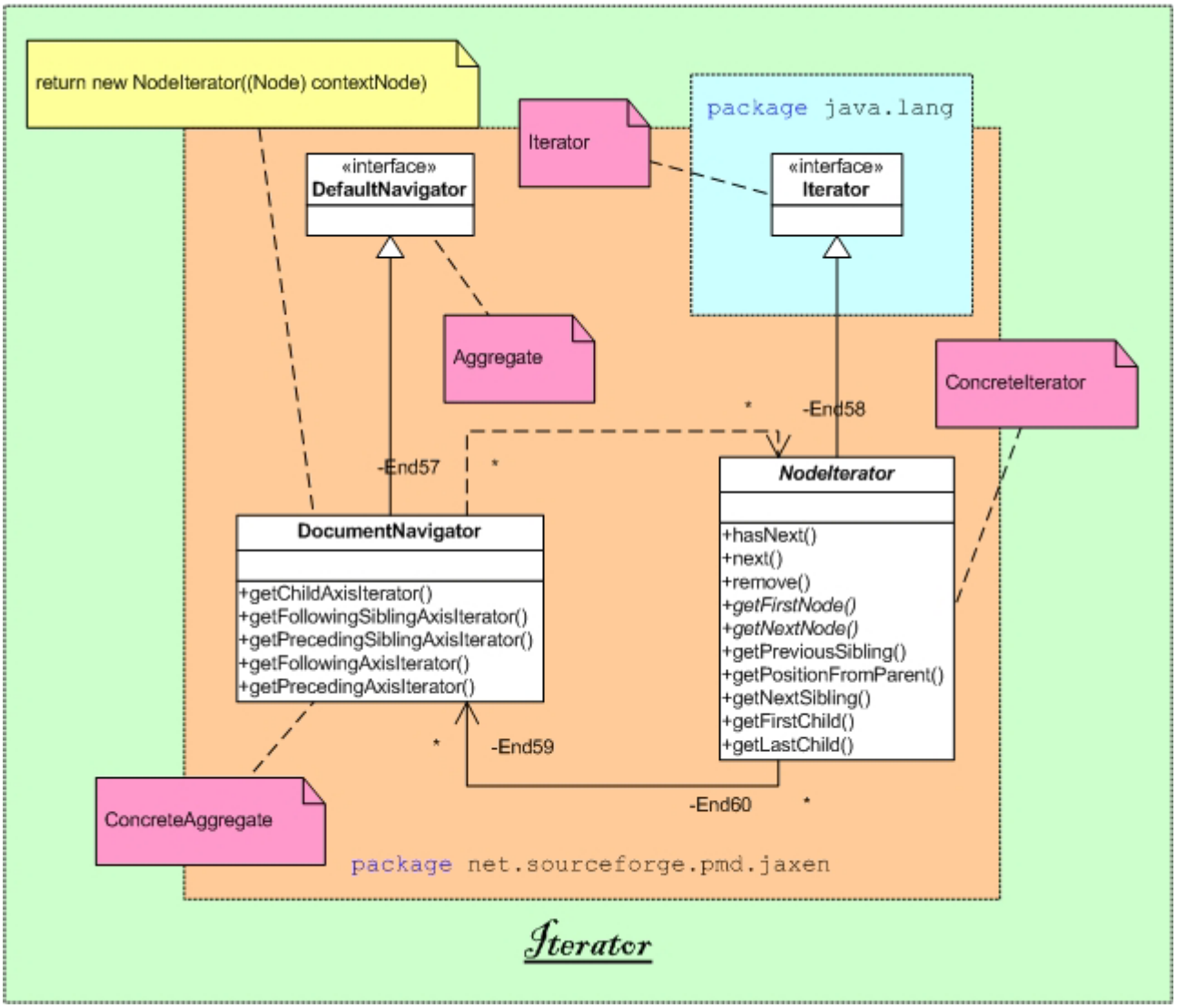


Figure 8. Illustration d'un diagramme de classes similaire au motif « Iterator ».

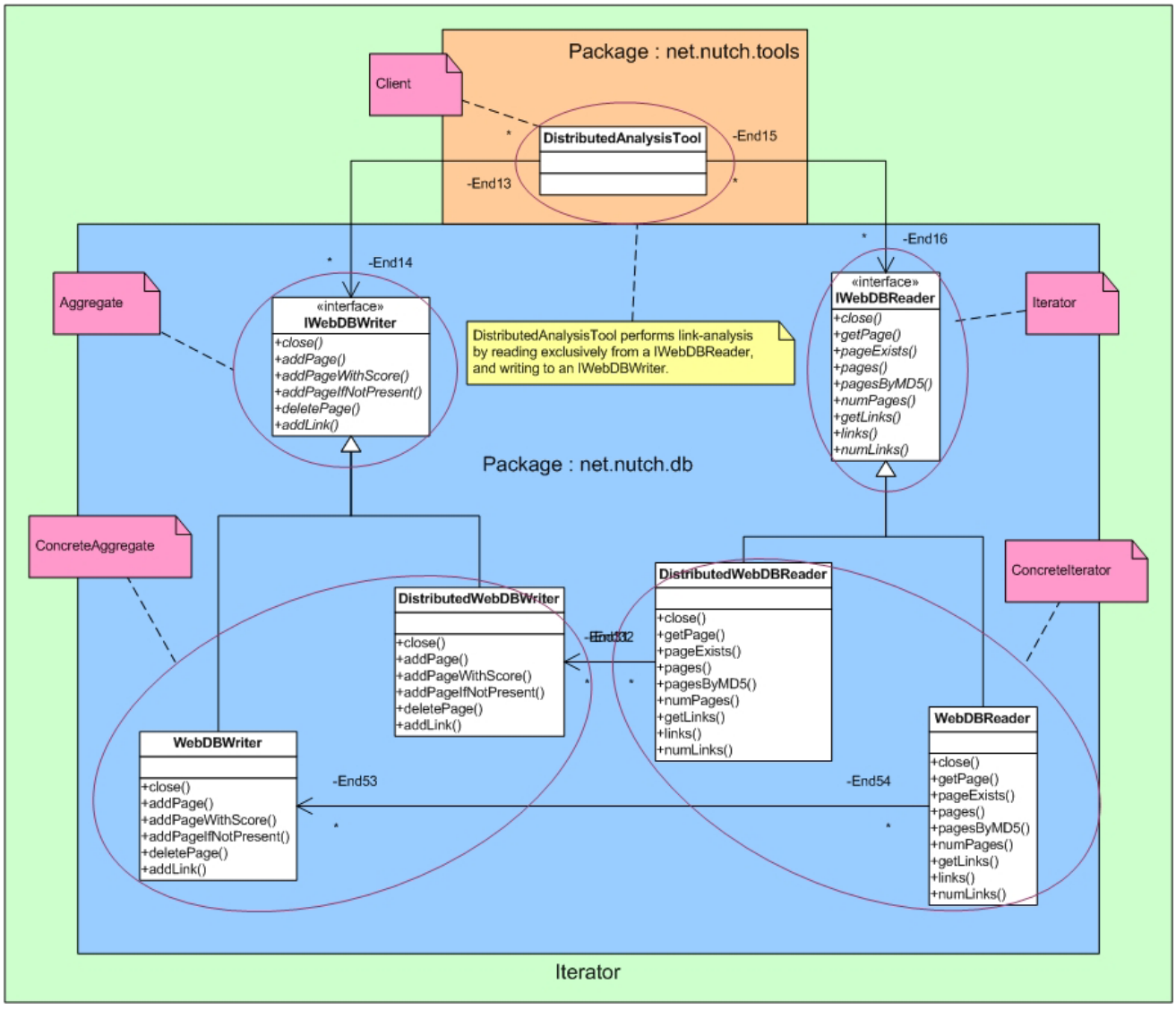


Figure 9. Illustration d'un diagramme de classes similaire au motif « Iterator »..

BIBLIOGRAPHIE

[i] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994. ISBN: 0-201-63361-2.

[ii] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. Technical report E53-315, MIT Sloan School of Management, December 1993.