

La détection manuelle des Design Patterns pour le projet Ptidej

Petit guide pratique

par

Nicola Grenon
Abdeljabar Hammoda
Rafik Ouanouki

Septembre 2006

Introduction

Bonjour chers étudiants membre du projet Ptidej.

Si vous lisez ce document, c'est probablement que Yann-Gaël Guéhéneuc a fait une nouvelle victime et que vous êtes inscrits pour la session dans un projet académique sous sa supervision. Projet ayant pour objectif la détection de Design Patterns dans du code déjà en fonction afin de permettre au reste du groupe de Ptidej d'avoir une banque de donnée pour leurs travaux.

Nous aussi avons dû affronter l'adversité (sic) et avons mené à bien cette entreprise, mais non sans mal. En fait, ayant trimé dur pour parvenir à nos fins, nous nous sommes dit que nous pourrions sans doute éviter les mêmes embûches à d'éventuels successeurs en la matière. Ce petit guide se veut donc un amalgame de petits trucs et astuces pour vous sauver du temps et, nous l'espérons vraiment, vous rendre la tâche un peu plus facile et intéressante.

Dans les pages qui suivent, vous trouverez donc des commentaires sur les Design Patterns (le cœur du projet), un livre de référence essentiel, les logiciels à utiliser et quelques autres références qui vous aideront sûrement.

Dernier point avant de commencer: Veuillez pardonner le ton employé pour la rédaction de ce texte. Il ne se veut pas du tout un ouvrage de référence précis, mais bien un guide menant rapidement aux points importants. Il y a donc quelques généralisations qui sont utilisées ici dans le but d'en venir directement à l'essentiel (notamment au sujet des design patterns, leurs nombre, types, etc.)

Bonne session et bonne chance à vous!



Abdeljabar ,



Nicola

et



Rafik

Les Design Patterns «Patrons de conception»

Premier truc, même si ça peut sembler une évidence, apprenez à connaître les patrons de conception AVANT de vous lancer dans leur détection, ça évite de faire du travail en double inutilement.

Ce que c'est:

«Les patrons de conception décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels.» - Wikipedia

En gros donc, ce sont les meilleures recettes qui ont été trouvées pour organiser les interrelations entre les classes de façon efficace.

Les types:

Selon le modèle du Gang of Four (GoF), il y a trois types de design patterns: Les creational, les structural et les behavioral. Leur nom indique assez clairement leur objectif, mais précisons un peu:

Creational: Sont utilisés pour standardiser la création d'objets par un logiciel, rendant ceux-ci plus aisément manipulables par la suite.

Structural: Servent à établir la façon dont vont interagir plusieurs classes pour former des structures plus imposantes.

Behavioral: Visent à définir les communications entre les objets et les algorithmes utilisés pour les calculs complexes (on peut ainsi changer d'algorithmes pour effectuer un travail précis plus aisément (ex.: algo de tri)).

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				223 CR Chain of Responsibility	163 CP Composite	175 D Decorator
87 AF Abstract Factory	325 TM Template Method	233 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Façade
97 BU Builder	315 SR Strategy	283 MM Memento	305 ST State	257 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

Livre de référence

Titre: Design Patterns
Elements of Reusable Object-Oriented Software
Écrit par: Erich Gamma, Richard Helm, Ralph Johnson,
John M. Vlissides.
Publié par: Addison Wesley Professional.
ISBN: 0-201-63361-2 / 978-0-201-63361-0
Édité: 31 octobre 1994



→ Si vous n'êtes pas 100% en confiance avec le concept de Design Pattern ←
→ Lisez ce livre! ←

Il n'est pas nécessaire de l'apprendre par cœur, mais au moins d'en comprendre l'essentiel, à savoir:

- L'intérieur des deux pages couvertures (début et fin) (Super aide-mémoire!)
- p.81, la définition d'un Creational Pattern
- p.137, la définition d'un Structural Pattern
- p.221, la définition d'un Behavioral Pattern

Les patterns:

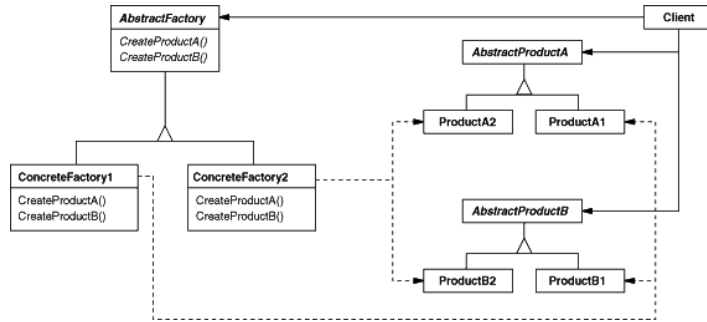
Voici maintenant un petit résumé (très très résumé) (en français!) de ce que dit le livre de tous les patrons qu'il présente sous forme de mini fiches. Le but est de voir d'un coup d'œil les relations entre les patrons, ainsi, à la découverte d'un patron en particulier, on sait en bloc ce qu'il faut observer aux alentours pour peut-être en trouver d'autres ou ne pas le confondre avec un autre patron similaire:

Exemple de mini fiche: [Code couleur: **Creational**, **Structural**, **Behavioral**]

Nom («autre noms»): Cas d'utilisation 1
Cas d'utilisation 2
*exemple concret
→ Souvent associé à cet autre patron
~ Quelquefois en conjonction avec
X Ne pas confondre avec ce patron X
Schéma de structure!

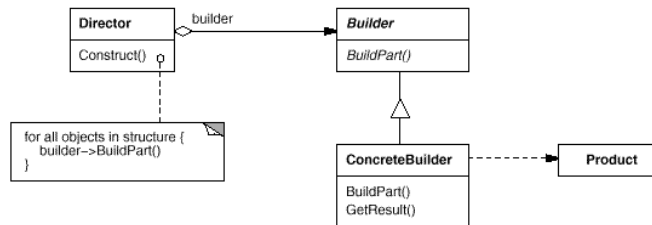
Abstract factory:
 («Kit»)

Système indépendant de ses produits.
 Système configurable pour différentes familles de produits.
 Famille de produits à faire fonctionner ensemble.
 Pour fournir uniquement l'interface d'une classe de produits.
 * Look-and-feel
 → Factory method
 → Prototype



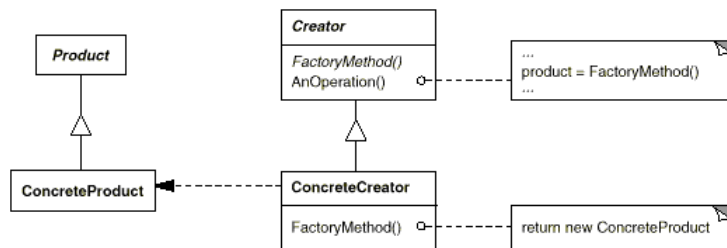
Builder:

Création indépendante des parties et de l'assemblage.
 Différentes représentations de l'objet construit.
 * Parseur d'un compilateur
 → Abstract factory
 → Composite



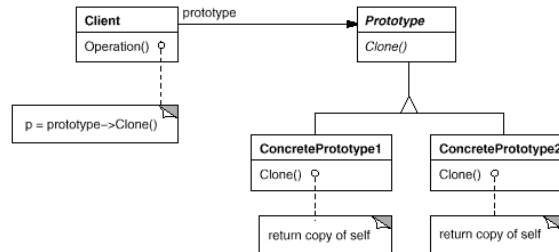
Factory method:
 («Visual constructor»)

La classe ne sait pas ce qu'elle va devoir créer.
 Les sous-classes choisiront quoi créer.
 Centraliser l'information à partir de quelles sous-classes on a créé quoi.
 → Factory method
 → Prototype



Prototype:

Classes à instancier connues seulement à l'exécution.
Évite une hiérarchisation parallèle.
Seulement quelques combinaisons d'état alors clones mieux adaptés qu'instanciations.
* Objets "modèles" stockés pour être utilisés ensuite
X Abstract factory X
→ Composite / Decorator

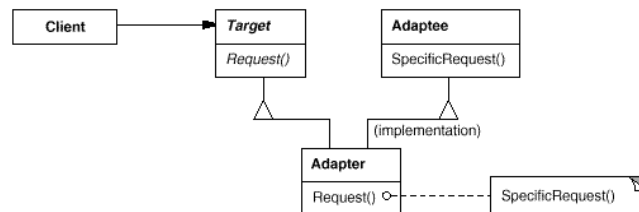


Singleton:

On s'assure qu'il n'y qu'une seule instance de l'objet.

Adapter:

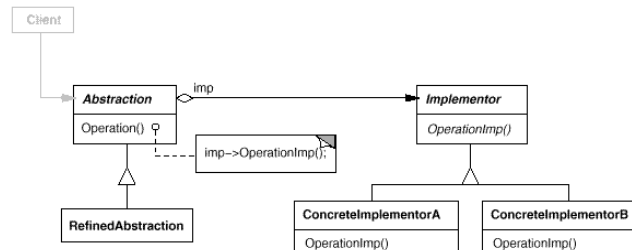
Interface d'une classe inappropriée.
Pour utiliser plusieurs classes dont l'interface n'est pas compatible (et peut-être de futures).
Utiliser plusieurs sous-classes via leur parent.
* User interface elements
X Bridge X
X Decorator X
X Proxy X



Bridge:

(«Handle / Body»)

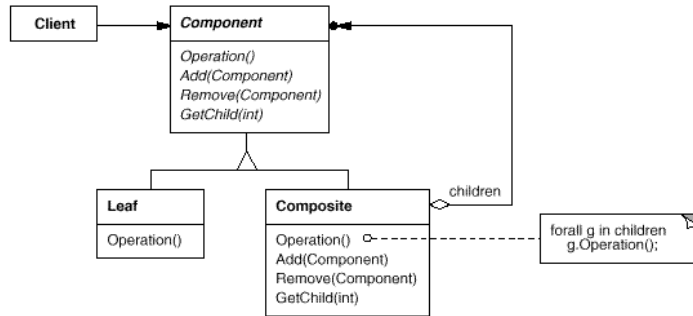
Lien temporaire entre abstraction et implémentation.
L'implémentation et l'abstraction en sous-classes et on s'attache indépendamment à certains.
Modifier l'implémentation mais pas le code d'une abstraction.
Partage d'une seule implémentation pour plusieurs objets.
* Memory management
* Common datastructures
→ Abstract factory
X Adapter X



Composite:

Représenter partie / ensemble d'une hiérarchie.
Pas de différence d'utilisation entre objet individuel et structure de plusieurs.

- * Fichiers / dossiers
- * Feuille / Branche / Arbre
- Chain of responsibility
- Decorator
- Iterator
- ~ Visitor
- ~ Flyweight



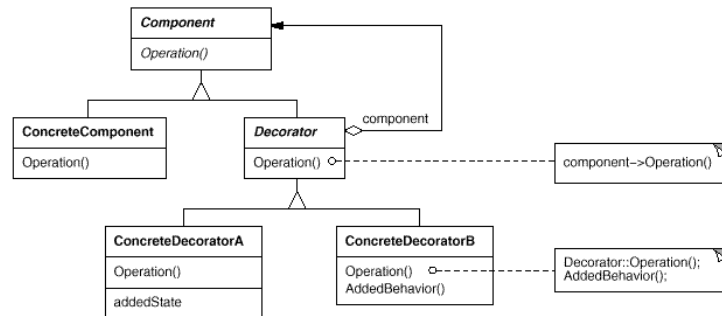
Decorator:
(«Wrapper»)

Ajouter des responsabilités à des objets dynamiquement et de façon transparente.

Responsabilités révocables.

Sous-classage pas pratique (trop de combinaisons).

- * Interfaces usagers orientées objets
- * Stream interfaces
- X Adapter X
- X Composite X
- X Strategy X



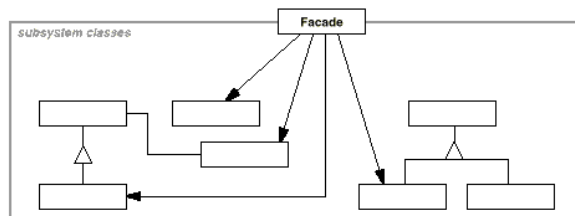
Façade:

Interface simplifiée / unifiée pour système.

Augmenter l'indépendance d'un sous-système.

Mis en couche d'un sous-système.

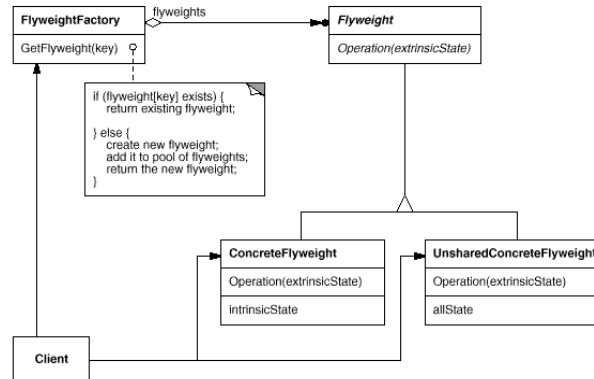
- * O/S, compilateur, etc.
- Abstract factory
- X Mediator X



Flyweight:

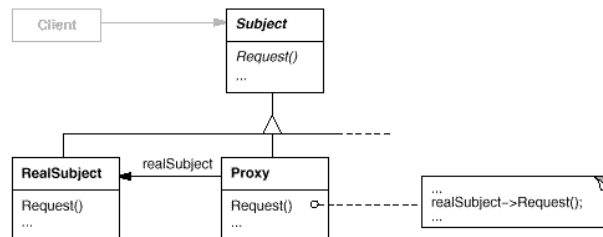
L'application utilise beaucoup d'objets.
Beaucoup d'objets pourraient être remplacés par quelques uns "partagés".

- * Lettres d'une ligne de texte, une classe par catégorie
- Composite
- State
- Strategy



Proxy:

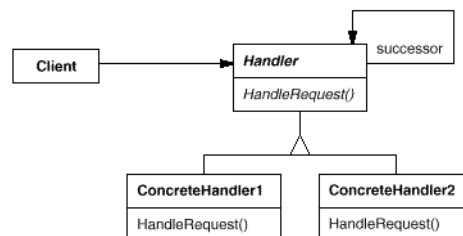
Protéger l'objet original.
Éviter le coût de l'objet simulé.
Représentant local d'un objet distant.
Garder le compte des accès aux objets distants afin de libérer.
Changer l'objet seulement au besoin.
Vérifier si l'objet est verrouillé.
X Adaptor X
X Decorator X



Chain of responsibility:

Évaluation / réaction en contexte.
Plusieurs objets peuvent réagir à une requête.
Le choix du répondant n'est pas connu d'avance et sera choisi automatiquement / dynamiquement.

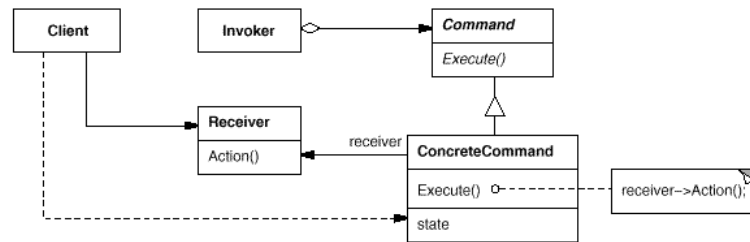
- * Bibliothèques de classes ⇔ user events
- * Transmission sur la chaîne...
- Composite



Command:

Encapsuler une requête dans un objet.

- * Menus / boutons dans une interface usagers
- * Délais, spécifications, queues de requêtes, durée de vie, undo, historique, redo, log...
- * Regroupement complexe d'opérations de base
 - Composite (macros)
 - Memento
- ~ Prototype

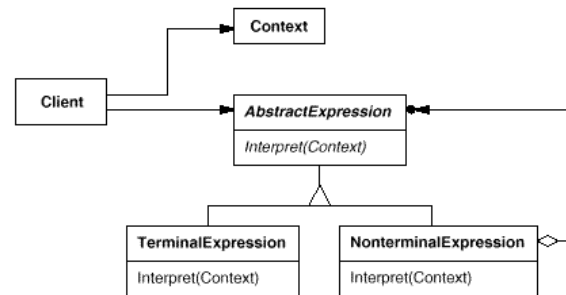


Interpreter:

Interpréter un sous langage.

Grammaire simple / efficacité peu importante.

- * Compilateur OO
- Composite
- Flyweight
- Iterator
- Visitor



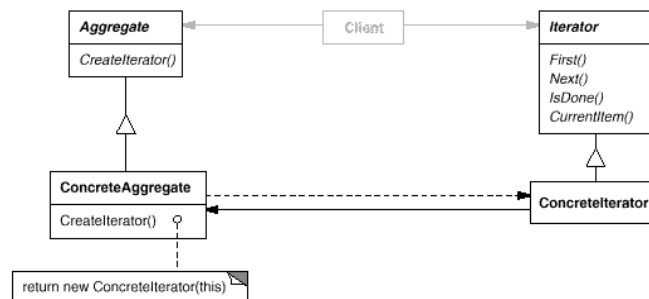
Iterator:

(«Cursor»)

Supporter plusieurs traversées d'un agrégat d'objets.

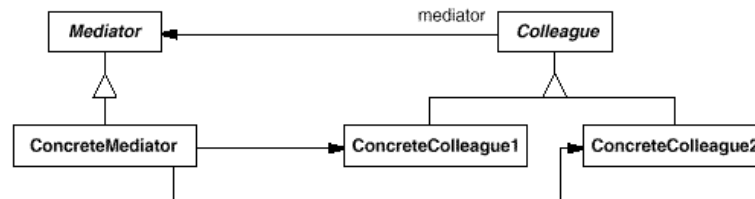
Supporter la traversée d'agrégats polymorphiques.

→ Composite / (...)



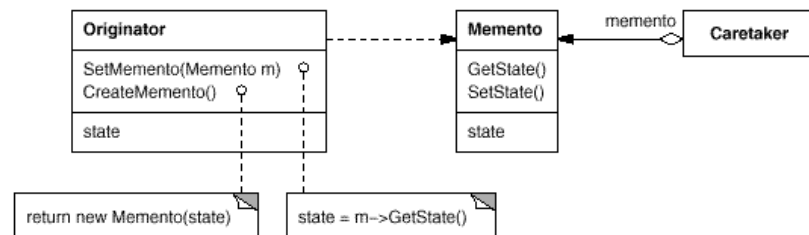
Mediator:

Diminuer le couplage.
Un ensemble d'objets communique de façon complexe.
Aider à la réutilisabilité.
Aide à distribuer un comportement sans sous classage.
* Ensemble de mises à jour indépendantes
→ Observer
X Façade X



Memento:

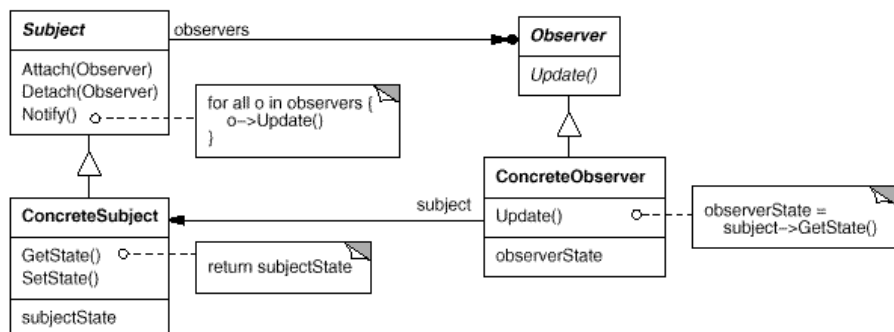
Sauvegarder l'état d'un objet (restaurable).
Protéger l'encapsulation (get / set state).
→ Command
→ Iterator



Observer:

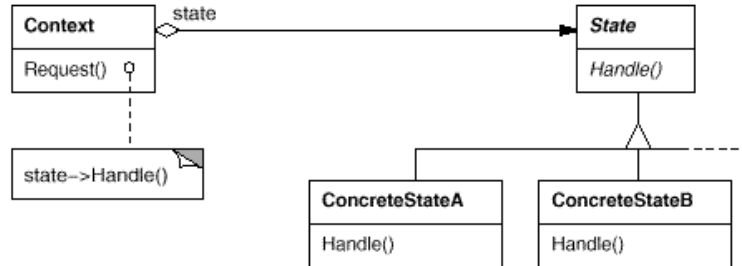
Client / Serveur (synchro de data).
On ne sait pas combien de clients.
Permet à un objet d'en informer d'autres sans les connaître ou en savoir le nombre.

MVC.
* Interfaces
→ Singleton
~ Mediator



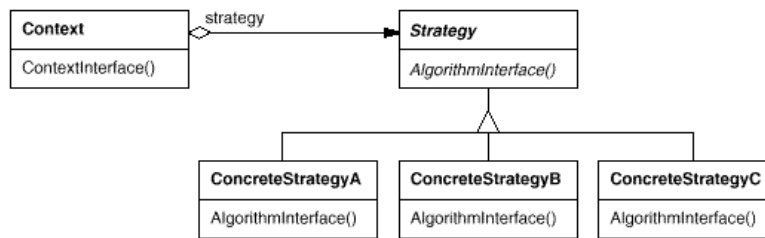
State:

Changement de fonctionnement suite à un changement d'état.
* Connexions
* "Outils" dans un dessin vectoriel (curseur)
→ Flyweight
~ Singleton



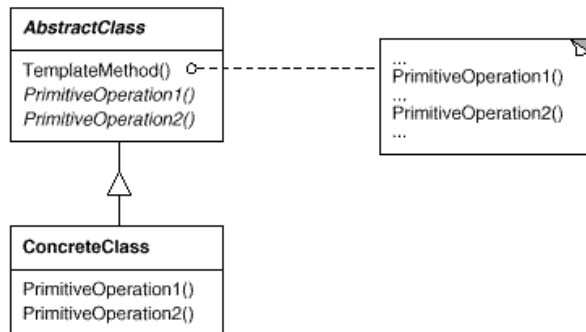
Strategy:
(«Policy»)

Algorithmes interchangeable.
Plusieurs classes différant seulement par interactions...
Cacher des algorithmes complexes.
* Découpage de stream
* Différentes stratégies de traitement
→ Flyweight



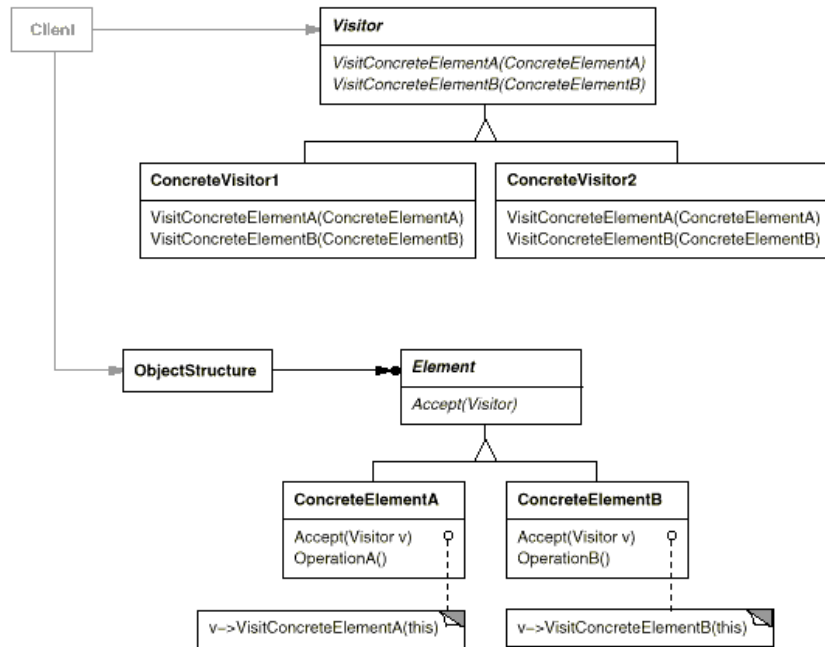
Template method:

Raffinage en sous-classes d'un algorithme complexe.
Pour différencier le comportement d'un algorithme pour différents types de sous-classes.
* Classes abstraites (presque toujours).
→ Factory method
X Strategy X



Visitor:

Opération à appliquer sur une structure.
Structures avec objets à interfaces différentes.
Plusieurs opérations sans liens à effectuer à des membres d'une structure.
On veut pouvoir définir de nouvelles opérations dans ce même contexte.
→ Composite
→ Interpreter



Logiciels

Pour ce travail, fonctionner sur une machine Windows simplifie grandement les choses. Les ordinateurs du labo ont déjà la majorité des logiciels nécessaires et s'il y en a de nouveaux qui pourraient vous servir il est possible de les installer ou de les faire installer. (Pierre Leduc?)

Note: Il ne sert à rien de chercher un logiciel qui détecte les patrons automatiquement, il n'y en a pas de bons (à l'heure actuelle) et c'est un des objectifs du projet Ptidej!

Eclipse <http://www.eclipse.org>

Comme nous travaillons dans un univers objet, et par le fait même que le code étudié est le java, le programme de choix recommandé pour l'étude des patrons de conception est évidemment Eclipse. Il est déjà en place sur les machines du labo, mais si vous voulez travailler à la maison, il est téléchargeable gratuitement.

Que vous installiez Eclipse chez vous ou que vous initialisiez votre configuration sur un ordinateur du labo, pensez à suivre les étapes de configuration du CVS (voir ressources). Certain installent une nouvelle version d'Eclipse directement sur le lecteur R:/ et ça fonctionne très bien (et c'est pratique pour l'accès d'un peu partout). Pensez toutefois à demander à faire augmenter l'espace maximum de stockage auquel vous avez droit. (Yann peut envoyer un courriel pour vous à ce sujet au responsable du réseau.)

Dans Eclipse, la recherche par mot clé (via le menu search) est simple et efficace, mais il y a un autre outil qui pourra ensuite vous permettre d'y voir plus clair: Le class viewer qui permet de vous faire une idée rapide de la structure des classes du projet observé.

Oxygen <http://www.oxygenxml.com/>

Bien que vous puissiez sans manuellement mettre à jour les informations que vous trouverez dans le fichier XML voulu, il s'agit là d'un bon logiciel pour travailler le XML.

Omondo <http://www.omondo.com/>

Ce plugin pour Eclipse permet d'afficher les schémas UML. C'est vraiment un outil très intéressant, nous vous le recommandons fortement.. Grâce à ce dernier, vous pourrez avoir un aperçu du schéma des classes du code étudié... et en jouant avec les classes un peu, vous pourrez voir les relations et les associer aux schémas du livre.

Microsoft Visio

D'autres équipes avant nous ont utilisé ce logiciel, il leur a été utile pour visualiser les relations entre les classes.

Describe

Un autre outil apprécié par de précédents collègues pour effectuer de la rétro-ingénierie.

Personnes ressources

Toutes les personnes du labo sont à la base une ressource intéressante. Il ne faut pas être trop timide, la serviabilité est contagieuse par ici. Vous avez grand avantage à discuter avec eux plutôt que de chercher une solution miracle sur le web!

Yann-Gaël Guéhéneuc [guehene@IRO.UMontreal.CA]

<http://www.yann-gael.gueheneuc.net/Work/Info/>

Local: 2345 (aux dernières nouvelles)

En espérant que vous savez qui il est si vous avez pris un projet avec lui ;-)



Stefan Monnier [monnier@IRO.UMontreal.CA]

<http://www.iro.umontreal.ca/~monnier/>

Local: 3353

Pas directement lié au projet, il est fera (probablement) partie de l'équipe d'évaluation.

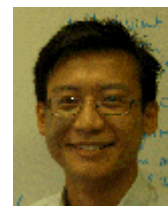


Duc-Loc Huynh

<http://ptidej.iro.umontreal.ca/participants/Members/huynhduc>

Local: aléatoire :-)

Très aimable individu très ferré dans le domaine des patrons de conception et dans le projet Ptidej. A conçu une super liste d'exemples et a déjà fait ce même projet par le passé. Il est une source d'inspiration sans fin!



Pierre Leduc

Pour les ressources informatiques.



Méthodologies

Voici les quelques méthodologies que nous avons utilisées et qu'on a cernées dans les rapports des équipes précédentes. Le nombre d'étoiles représente l'efficacité que nous attribuons à chacune. Ceci dit, souvent une méthode simple et peu efficace vaut la peine d'être employée pour faire «un premier passage», mais il faut s'attendre à devoir confirmer nos résultats.

Recherche par mots-clés: *

Les programmeurs commentent souvent leur code ou utilisent un pattern facilement identifiable dans le nom des classes. Ainsi, une recherche par mot-clé sur "FactoryXXX" pourrait donner une liste intéressante de résultats. Attention, un programmeur qui annote son code ainsi n'a pas forcément bien identifié son patron!

Recherche des interfaces: ***

Encore une fois, par mot clé on va ici rechercher "INTERFACE". Une fois qu'on a obtenu la liste des interfaces développées, on peut en choisir une et analyser quelles sont les classes qui dérivent de cette interface et leur relations pour en extraire un patron. Notons que presque tous les patrons de conception ont une INTERFACE en racine.

Les singletons: *

(Dans les conseils, vous noterez qu'il sont de peu de valeur, mais quelques uns ça fait pas de mal.) Une recherche par mot-clé sur "==NULL" est un bon outil, car au moment d'instancier l'objet, il y aura toujours un test de ce type.

Relations inter patrons: ***

En se fiant à la page de couverture en fin de livre ou à la liste des patrons de ce guide, on voit les liens qui existent souvent entre les patrons de conception. Alors lorsqu'on trouve un patron, avant de repartir dans une chasse à l'aveugle, vérifiez les patrons à proximité!

À partir d'une petite classe: **

En trouvant une petite classe dans le code du projet étudié, on isole ensuite les classes qui l'instancie et que celle-ci instancie. (Recherche par mots-clés sur "new cetteclasse" et les new dans le code de cette classe-ci.) On illustre en parallèle sur une feuille un schéma des classes que l'on lie peu à peu. Attention ici de pas «tout» mettre (ça vient avec l'expérience), mais bien les classes *user* qui ont un lien logique étroit avec la première. Il ne restera qu'à comparer le schéma avec les diagrammes de structure des patrons du livre. En analysant un peu le code et son utilité, on peut restreindre alors la recherche à une seule catégorie de patrons.

À partir d'une grosse classe: **

Si vous trouvez une classe qui semble contenir beaucoup de commentaires ou qui a un code plus long que la moyenne et que vous prenez le temps d'en analyser l'utilité, les objectifs, etc. Vous serez probablement à même de reconnaître les exemples types cités dans le livre (ou le résumé des patrons de ce guide). Parfois aussi ce sera le nom d'un rôle dans un patron qui deviendra évident, il ne restera plus qu'à retrouver à quel patron il appartient.

Omondo: ****

En affichant le diagramme des classes sur un package, on peut directement tenter de reconnaître le schéma de structure des patrons (via le livre). Attention toutefois au fait que le logiciel ne permet que de voir le contenu d'un seul package à la fois, ce qui peut scinder des patrons et les rendre inidentifiable. Une façon de contourner le problème est de créer un package vide et d'y glisser des classes une à une qui proviennent des autres packages. Cela peut toutefois entraîner beaucoup de tâtonnements, mais par contre être un très bon outil de vérification.

Autres ressources

Principalement des liens web et l'accès technique aux ressources dont vous aurez besoin:

- Page du projet Ptidej: <http://ptidej.iro.umontreal.ca/>
 - Page permettant d'activer le CVS dans Eclipse:
<http://ptidej.iro.umontreal.ca/material/development/cvseclipse>
 - Articles Wikipedia intéressants:
http://fr.wikipedia.org/wiki/Patron_de_conception
http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29
http://en.wikipedia.org/wiki/Design_Patterns
 - Le livre complet en ligne:
<http://ici.cs.ubbcluj.ro/~raduking/Books/Design%20Patterns/>
 - Vous voulez avoir ce livre moins cher? Amazon!
http://www.amazon.ca/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=sr_11_1/702-8370313-5175231?ie=UTF8
 - Le site web de notre projet: <http://ift3051.igt.net/>
-
- Il y a une imprimante à laquelle vous avez accès depuis le labo, il vous suffit de demander le code de la porte (c'est à 3 mètres à gauche puis à droite en sortant du labo) et de ne pas exagérer.
 - Nous joignons à ce document quelques livres et documents informatiques (sur CD) que nous n'osons pas mettre en ligne par soucis de copyrights.
 - Dans le CVS du projet, il y a une série d'exemples très intéressants pour chaque modèle de patron. Gros merci à Duc-Loc Huynh. En les survolant, vous aurez un premier aperçu de ce à quoi peut ressembler ce que vous cherchez.

Derniers conseils

- N'hésitez pas à poser beaucoup de questions aux personnes ressources que vous avez identifiées – surtout au début du projet –.
- Il n'y a pas toujours des patrons identifiables dans du code.
- Un exemple de la structure en java de chacun des patrons du livre se trouve sur le CVS du projet Ptidej. Prenez quelques minutes pour survoler le tout, c'est intéressant.
- Il n'est pas très utile de trouver beaucoup de Singleton. Ils sont faciles à trouver et nombreux, mais de peu de valeur pour le projet.
- Commencez par un PETIT projet. Se lancer dans un trop gros projet en commençant est peu efficace et démoralisant. Un petit projet permet de cerner plus rapidement quelques patrons et de prendre ainsi de l'expérience.
- Commencez par trouver les patrons Creational pour vous habituer, les Behavioral sont les plus complexes à cerner, comme ça vous vous ferez la main.
- Trouver un patron, ça peut être très long. Il ne faut pas se décourager.
- Différencier deux patrons similaires peut parfois être complexe. Nous suggérons alors de lire la motivation de ce patron dans le livre pour mieux le cerner.
- Souvent les patrons plus gros sont un peu déformés et/ou il manque un ou des morceaux... Ça ne veut pas dire que ce n'en est pas un! Il faut simplement alors être plus attentif et révérifier ses résultats pour être sûr de notre découverte.
- Le mot d'ordre pour ce projet est la qualité avant la quantité. De mauvais patrons vont nuire à l'ensemble du projet, il est donc primordial de vérifier ses résultats.

Humour

Pour vous donner espoir, aux moments les plus bas, souvenez-vous que vous n'êtes pas les premier à souffrir à cause des Design Patterns...

Resign Patterns
Ailments of Unsuitable Project-Disoriented Software
by
Michael Duell
mitework@yercompany.com

Abstract

Anyone familiar with the book of patterns by the Gang of Four [1] knows that the patterns presented in the book represent elegant solutions that have evolved over time. Unfortunately, extracting these patterns from legacy code is impossible, because nobody knew that they were supposed to be using these patterns when they wrote the legacy code. Hence, this work is a catalog of patterns for the masses. The patterns presented here represent abundant solutions that have endured over time. Enjoy reading the patterns, but please don't use them!

1 Cremational Patterns

Below is a list of five cremational patterns.

1.1 Abject Poverty

The Abject Poverty Pattern is evident in software that is so difficult to test and maintain that doing so results in massive budget overruns.

1.2 Blinder

The Blinder Pattern is an expedient solution to a problem without regard for future changes in requirements. It is unclear as to whether the Blinder is named for the blinders worn by the software designer during the coding phase, or the desire to gouge his eyes out during the maintenance phase.

1.3 Fallacy Method

The Fallacy method is evident in handling corner cases. The logic looks correct, but if anyone actually bothers to test it, or if a corner case occurs, the Fallacy of the logic will become known.

1.4 ProtoTry

The ProtoTry Pattern is a quick and dirty attempt to develop a working model of software. The original intent is to rewrite the ProtoTry, using lessons learned, but schedules never permit. The ProtoTry is also known as legacy code.

1.5 Simpleton

The Simpleton Pattern is an extremely complex pattern used for the most trivial of tasks. The Simpleton is an accurate indicator of the skill level of its creator.

2 Destructural Patterns

Below is a list of seven destructural patterns.

2.1 Adopter

The Adopter Pattern provides a home for orphaned functions. The result is a large family of functions that don't look anything alike, whose only relation to one another is through the Adopter.

2.2 Brig

The Brig Pattern is a container class for bad software. Also known as module.

2.3 Compromise

The Compromise Pattern is used to balance the forces of schedule vs. quality. The result is software of inferior quality that is still late.

2.4 Detonator

The Detonator is extremely common, but often undetected. A common example is the calculations based on a 2 digit year field. This bomb is out there, and waiting to explode!

2.5 Fromage

The Fromage Pattern is often full of holes. Fromage consists of cheesy little software tricks that make portability impossible. The older this pattern gets, the riper it smells.

2.6 Flypaper

The Flypaper Pattern is written by one designer and maintained by another. The designer maintaining the Flypaper Pattern finds herself stuck, and will likely perish before getting loose.

2.7 ePoxy

The ePoxy Pattern is evident in tightly coupled software modules. As coupling between modules increases, there appears to be an epoxy bond between them.

3 Misbehavioral Patterns

Below is a list of eleven misbehavioral patterns.

3.1 Chain of Possibilities

The Chain of Possibilities Pattern is evident in big, poorly documented modules. Nobody is sure of the full extent of its functionality, but the possibilities seem endless. Also known as Non-Deterministic.

3.2 Commando

The Commando Pattern is used to get in and out quick, and get the job done. This pattern can break any encapsulation to accomplish its mission. It takes no prisoners.

3.3 Intersperser

The Intersperser Pattern scatters pieces of functionality throughout a system, making a function impossible to test, modify, or understand.

3.4 Instigator

The Instigator Pattern is seemingly benign, but wreaks havoc on other parts of the software system.

3.5 Momentum

The Momentum Pattern grows exponentially, increasing size, memory requirements, complexity, and processing time.

3.6 Medicator

The Medicator Pattern is a real time hog that makes the rest of the system appear to be medicated with strong sedatives.

3.7 Absolver

The Absolver Pattern is evident in problem ridden code developed by former employees. So many historical problems have been traced to this software that current employees can absolve their software of blame by claiming that the absolver is responsible for any problem reported. Also known as It's-not-in-my-code.

3.8 Stake

The Stake Pattern is evident in problem ridden software written by designers who have since chosen the management ladder. Although fraught with problems, the manager's stake in this software is too high to allow anyone to rewrite it, as it represents the pinnacle of the manager's technical achievement.

3.9 Eulogy

The Eulogy Pattern is eventually used on all projects employing the other 22 Resign Patterns. Also known as Post Mortem.

3.10 Tempest Method

The Tempest Method is used in the last few days before software delivery. The Tempest Method is characterized by lack of comments, and introduction of several Detonator Patterns.

3.11 Visitor From Hell

The Visitor From Hell Pattern is coincident with the absence of run time bounds checking on arrays. Inevitably, at least one control loop per system will have a Visitor From Hell Pattern that will overwrite critical data.

4 References

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Michael Duell is an Engineer at AG Communication Systems, where his Resign Patterns have been rejected in favor of the Gang of Four Design Patterns.

"Resign Patterns: Ailments of Unsuitable Project-Disoriented Software," The Software Practitioner, Vol. 7, No. 3, May-June 1997, p. 14.

; -)