

Introduction aux GRASP Patterns

Par Pierre Chauvin 

Date de publication : 11 avril 2004

Dernière mise à jour : 1 mai 2004

Introduction aux GRASP Patterns.

I - Introduction.....	3
I.1 - Historique.....	3
I.2 - Présentation des GRASP patterns.....	3
I.3 - Pré-requis.....	3
II - Les GRASP Patterns : Les responsabilités dans la conception objet.....	5
II.1 - Expert (en Information).....	5
II.2 - Exemple.....	5
II.3 - Créateur.....	6
II.4 - Faible couplage.....	7
II.5 - Forte Cohésion.....	7
II.6 - Contrôleur.....	8
III - Liens.....	10

I - Introduction

I.1 - Historique

Les patterns furent utilisés bien avant que les systèmes d'informations modernes n'apparaissent, dans d'autres disciplines comme la construction et l'architecture de bâtiments.

Les premiers travaux appliqués à l'informatique datent de 1987, dans une étude réalisée par Beck et Cunningham et s'attachant à définir des patterns pour des interfaces utilisateurs. Dans les années 90, 4 personnes consacrèrent leurs recherches aux design patterns : Gamma, Helm, Johnson et Vlissides éditérent un livre qui fait toujours référence « Design Patterns : Elements of reusable object oriented software » en 1994. Les Patterns fondamentaux décrits dans cet ouvrage sont couramment appelés Patterns Gof pour « Gang of Four ».

Une autre contribution de référence est fournie par Craig Larman qui décrit des modèles de conception plus intuitifs : les Patterns GRASP. A partir de tous ces modèles et des modèles actuellement proposés, des nombreuses classifications peuvent être rencontrées dans la littérature. Le présent document est dédié à la présentation des patterns GRASP de Craig Larman, et ne vous noiera pas dans des considérations fondamentales plus complexes.

I.2 - Présentation des GRASP patterns

Tout d'abord que signifie GRASP ? Il s'agit de l'acronyme de *General Responsibility Assignment Software Patterns*, ou patterns généraux d'affectation des responsabilités. Il s'agit de patterns fondamentaux qu'il est nécessaire d'appliquer pour une bonne conception. Ces patterns sont, pour la plupart, directement issus du bon sens du concepteur, et leur représentation est tout aussi intuitive.

Qu'est ce qu'un Pattern ? Un Pattern est un modèle de conception. Au fur et à mesure de leurs développements, des principes généraux répétitifs sont remarqués, et des idiomes courants ont ainsi été identifiés. En formalisant ceux-ci, sous la forme de la présentation d'un problème de conception, et de la solution pour le résoudre, est appelé Pattern.

Ce document a pour objectif d'introduire une formalisation de cette catégorie de Patterns, défini par Craig Larman. Il s'agit juste d'une codification de patterns répétitifs et courants, qu'un concepteur de niveau moyen peut trouver simpliste et basique, mais il s'agit bien du but intrinsèque des GRASP Patterns.

Un pattern doit répondre à certains aspects essentiels :

- Un pattern a un nom évocateur, qui représente bien l'essence même de son existence.
Exemple : le pattern GoF « Singleton »
- Un pattern résout un problème.
Exemple : une classe ne peut avoir qu'une et une seule instance.
- Un pattern fournit une solution.
Exemple : il faut créer une méthode statique de la classe qui retourne l'instance unique ou « Singleton ».

I.3 - Pré-requis

Ce document s'appuie sur des exemples de notation UML, et il serait préférable de connaître les bases de Unified Modeling Language pour comprendre les quelques diagrammes utilisés. De la même façon, des notions sur les systèmes orientés objets sont probablement nécessaires pour mieux appréhender les différents concepts étudiés. Je vous conseille donc de consulter les tutoriaux sur UML de Developpez.com que vous trouverez à l'adresse suivante : <http://uml.developpez.com>, ou alors de prendre directement connaissance des spécifications fournies par l'Object Management Group (OMG - <http://www.omg.org/uml/>).

Les exemples utilisés pour illustrer les présentations des différents patterns ont été réalisés avec Gentleware Poséidon 1.6 Community Edition, que vous pouvez télécharger librement : <http://www.gentleware.com/products/poseidonCE.php3> .

II - Les GRASP Patterns : Les responsabilités dans la conception objet

II.1 - Expert (en Information)

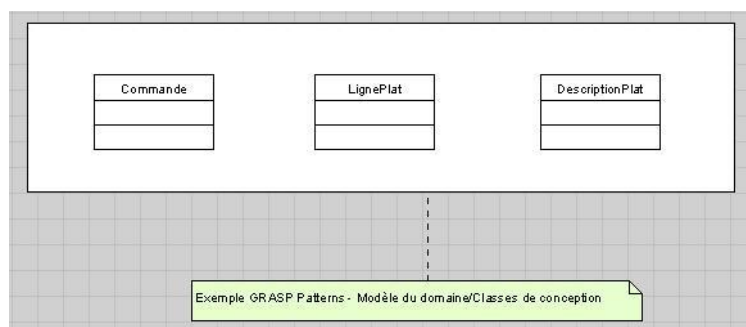
L'objectif de ce pattern est d'affecter au mieux une responsabilité à une classe logicielle. Afin de réaliser ces responsabilités, cette classe doit disposer des informations nécessaires.

Lors de l'établissement du modèle de conception (modèle du domaine), il est très fréquent d'obtenir des centaines de classes logicielles, et l'étude approfondie des cas d'utilisation engendre l'extraction de centaines de responsabilités. En général, on impute la réflexion sur l'attribution des responsabilités dès la conception des diagrammes d'interaction entre les objets. C'est à ce moment que l'on définit quelle classe a quelle(s) responsabilité(s), ce qui détermine sans aucun doute une bonne ou une mauvaise conception. En effet, si les responsabilités sont mal réparties, les classes logicielles vont être difficilement maintenables, plus dure à comprendre, et avec une réutilisation des composants peu flexible.

Ainsi, comme d'autres pattern GRASP, le modèle Expert est un pattern relativement intuitif, qui en général, est respecté par le bon sens du concepteur. Il s'agit tout simplement d'affecter à une classe les responsabilités correspondants aux informations dont elle dispose intrinsèquement (qu'elle possède) ou non (objets de collaborations).

II.2 - Exemple

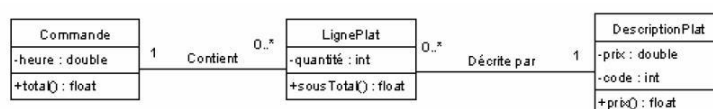
Dans cet exemple simpliste, nous nous situons dans une démarche de gestion de commandes (au restaurant). Nous allons étudier une application de ce pattern sur le modèle du domaine suivant :



La question est de savoir quelles classes peuvent avoir la responsabilité de donner le total d'une commande. Il semble logique que la classe Commande soit la meilleure candidate à cette responsabilité. Il ne faut cependant pas oublier qu'une commande peut être composée de plusieurs plats, dans des quantités éventuellement différentes. Une autre application de Expert consiste alors à affecter à la classe LignePlat la responsabilité d'afficher le sous total d'une ligne.



L'ensemble de ses remarques permet au concepteur d'ajouter un certain nombre de méthodes identifiées (abstraction faite des accesseurs):



Les responsabilités dans cet exemple sont donc :

Commande	Connaître le total de la commande
LignePlat	Connaître le sous total d'une ligne
DescriptionPlat	Connaître le prix d'un plat

Il ne faut pas confondre Responsabilité et Cohésion (pattern vu plus loin). Par exemple dans le cas d'un historique des commandes (via persistance Base de données ou Mapping Objet par exemple), la plupart des informations à conserver sont connues par la classe Commande, qui devient alors une candidate idéale à l'opération de sauvegarde et à la responsabilité de sa persistance.



Un problème se pose donc : en plus d'être une classe conceptuelle issue du modèle du domaine, doit-on lui imputer une responsabilité de nature différente ? La réponse est non et une telle représentation serait médiocre. Il ne faut pas « hétérogénéiser » les responsabilités d'une classe, la cohésion en est affectée, tout comme le couplage (la classe Commande utilisera et sera associée à des classes JDBC ou de Mapping XML par exemple), qui deviendra fort et limitera la réutilisation. Pour aider à la détection de cette confusion, nous étudierons plus loin le Pattern architectural MVC, qui permet de séparer les différentes couches et leurs interactions.

II.3 - Créateur

Le pattern Créateur est lui aussi un modèle de conception relativement intuitif. Il consiste en la détermination de la responsabilité de la création d'une instance d'une classe par une autre.

Exemple :

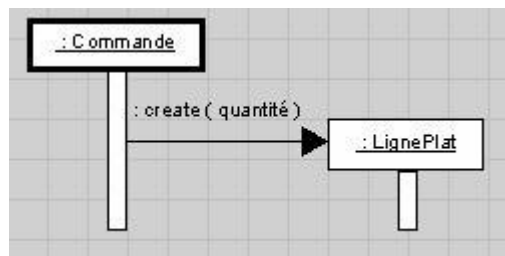
- Une classe A agrège des instances de la classe B
- Une classe A contient des instances de la classe B
- Une classe A utilise des instances d'une classe B
- Une classe A possède les données nécessaires à l'initialisation des instance de classe B, etc.

Dans ces différents cas, la classe A a la responsabilité de créer des instances de la classe B. La création d'objets dans un système orienté objet étant une tâche très fréquente, est guidée par ce pattern qui « guide » l'affectation des responsabilités de création d'instance.

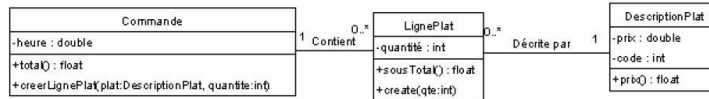
L'avantage de suivre cette ligne directrice est le respect d'un autre pattern et principe fondamental : le faible couplage. Le faible couplage (voir le pattern Faible couplage) implique des dépendances amoindries donc les classes sont plus maintenables et la réutilisation accrue.


Exemple :

Dans le même exemple que le pattern Expert en Information, nous pouvons nous poser la question : Qui à la responsabilité de créer une instance de la classe logicielle LignePlat ? . Selon les 5 principes évoqués plus haut, la classe Commande paraît la meilleure candidate. Le diagramme d'interaction (de séquence) suivant semble ainsi cohérent :



Cette nouvelle affectation de responsabilité entraîne le concepteur à définir une nouvelle méthode créerLignePlat(plat, quantité) dans la classe Commande :




 Dans certains cas, lors de l'utilisation d'une instance de façon non triviale (création de l'instance conditionnelle, réutilisation de l'instance créé afin de gagner en performance, etc.), il est conseillé de laisser la responsabilité de la création à une classe annexe, issue de l'application du Pattern Fabrication (pattern GoF).

II.4 - Faible couplage

Le faible couplage a pour objectif de faciliter la maintenance en minimisant les dépendances entre éléments. Pourquoi le couplage ?

Le couplage exprime la relation étroite qu'un élément (Classe, Système ou sous système) entretient avec un ou des autres éléments. Un élément faiblement couplé ne possède pas ou peu de dépendances vis à vis d'autres éléments. Un couplage trop fort entre deux éléments peut entraîner lors de la modification d'un de ses éléments une modification d'un élément lié, ou bien une compréhension plus difficile des éléments pris séparément, ou encore une réutilisation contraignante du fait de la quantité d'éléments à intégrer pour en utiliser un seul.

Bien entendu, il ne faut pas tomber dans le piège de décider de concevoir des éléments tous indépendants et faiblement couplés, car ceci irait à l'encontre du paradigme objet définissant un système objet comme un ensemble d'objets connectés les uns aux autres et communiquant entre eux.

Un système abusivement faiblement couplé sera peu cohésif avec de nombreuses similitudes internes et des éléments de dimensions importantes.

Exemple de couplage :

- Une sous-classe héritant d'une autre classe est un couplage fort.
- Une classe qui se veut très générique doit être faiblement couplée.

En général ce pattern est appliqué inconsciemment, c'est davantage un principe d'appréciation qu'une règle. Il faut essayer de trouver le juste équilibre et le rapport optimum avec les patterns Expert en Information, Forte Cohésion afin d'obtenir des classes :

- Cohésives
- Réutilisables
- Compréhensibles
- Maintenables
- Indépendantes de la modification d'autres composants

II.5 - Forte Cohésion

Qu'est-ce que la cohésion ?

La cohésion mesure le degré de spécialisation des responsabilités d'un composant / classe. Comme dans le pattern Faible couplage, la cohésion médiocre altère la compréhension, la réutilisation, la maintenabilité et subit toute sorte de changements.

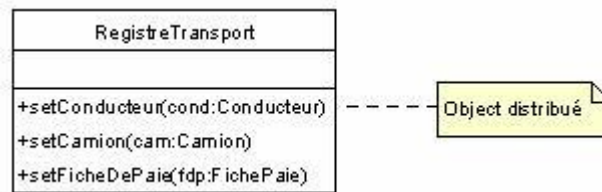
Par exemple, on ne peut exiger d'un réfrigérateur qu'il fasse radiateur et range-disques en même temps. La multiplication des disciplines à responsabilité accroît exponentiellement le risque d'erreurs intrinsèques à cette classe.

Un contre exemple est tiré d'une pratique courante dans les systèmes à communication par objets distribués (CORBA, RMI).

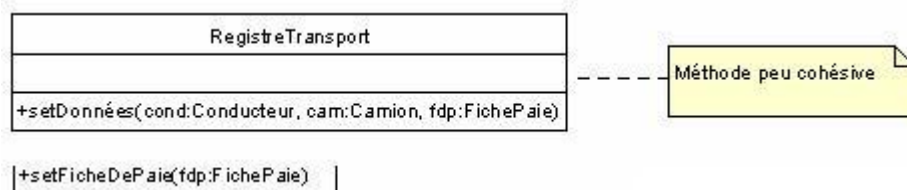


Ainsi, pour justifier et « amortir » le coût d'un appel à un objet distant, et favoriser les performances, les objets distribués sur le serveur seront moins nombreux et faiblement cohésifs. Cela permet ainsi d'interfacer avec un seul objet plusieurs opérations de nature différentes. Ce principe s'appelle la « granularité grossière ».

Exemple :



Sur un objet distant, au lieu d'invoquer 3 méthodes cohésives on préfère appeler une seule méthode avec toutes les données en paramètres et moins cohésives.



II.6 - Contrôleur

Le pattern Contrôleur est un modèle très utilisé dans les systèmes actuels. Ce modèle est le premier de ce document qui n'est pas applicable aussi intuitivement que Expert, Forte Cohésion, Créateur et Faible Couplage.

Le pattern Contrôleur consiste en l'affectation de la responsabilité de la réception et/ou du traitement d'un message système à une classe. Une classe Contrôleur doit être créée si elle répond à l'un des cas suivants :

- La classe représente un contrôleur de façade, c'est à dire l'interface d'accès à l'ensemble d'un système.
- La classe représente le scénario issu d'un cas d'utilisation. On le nomme en général « Session », et est chargé de traiter tous les événements systèmes contenus dans un scénario de cas d'utilisation.

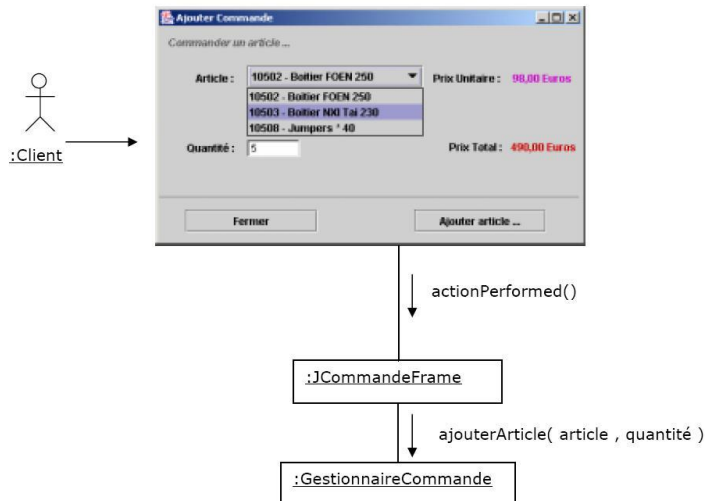


Attention

- La plupart des classes d'interaction homme machine (IHM) ne doivent pas être des contrôleurs car leurs responsabilités primaires résident dans la capture d'événements et messages. Néanmoins elles utilisent un contrôleur pour coordonner les différentes opérations.
- Le contrôleur peut déléguer des tâches à d'autres objets, il existe surtout dans le but de coordonner les actions et événements. Par exemple le client ne pourra pas ajouter d'articles à son panier si celui-ci n'existe pas. Les classes contrôleur doivent conserver un minimum de cohésion, afin de pouvoir le réutiliser et le comprendre aisément.

Exemple :

Dans un contexte de commande d'articles sur un site de e-commerce, ou via un Webservice, nous obtiendrons pour le cas d'utilisation « saisir des articles » :



Ici, la couche IHM est assurée par la classe `JCommandeFrame`, qui utilise le contrôleur `GestionnaireCommande`, pour transmettre les messages systèmes.

Le pattern contrôleur, fait parti du pattern architectural MVC (Model View Controller) implémenté par certains frameworks comme Struts. Il permet d'organiser en couches une application, en respectant le modèle du domaine d'un côté, l'interface graphique utilisateur (GUI) et la logique applicative. Cela permet, de réutiliser les composants, et de pouvoir changer aisément l'IHM (Browser Internet, Application Swing, etc.).

III - Liens

World Wide Web

OMG	http://www.omg.org
Site de Craig Larman	http://www.craiglarman.com
The HillSide Group	http://www.hillside.net/patterns/
Papers on Patterns	http://choices.cs.uiuc.edu/sane/dpatterns.html
The Server Side	http://www.theserverside.com/patterns
Gentleware Poséidon CE	http://www.gentleware.com
NetBeans	http://www.netbeans.org

Bibliographie

UML et Les Design Patterns (670 p. Fév 2002)	Craig Larman □ CampusPress
Design Patterns par la pratique (278 p. Sept 2002)	A. Shalloway, J. R. Trott - Eyrolles
Design Patterns (480 p.)	Gof - Vuibert
Design Patterns : Java Workbook (496 p Juil 2002)	Steven John Metsker □ Pearson Education France

Versions de ce document

Release 1.0 - 11/04/2003	Présentation de 5 patterns GRASP : Expert, Forte Cohésion, Faible couplage, Créateur, Contrôleur
Release 1.1 □ 14/04/2003	Correction de fautes mineures
Release 2.0 □ courant Avril-Mai 2003	Ajout de 4 autres GRASP Patterns