**Conception Orientée Objet et Diagramme d’Interaction**

Dans le monde de l'orienté objet, les design patterns se présentent comme un catalogue de méthodes de résolution de problèmes récurrents.

Un pattern ou motif de conception est un document qui décrit une solution générale à un problème qui revient souvent.

Les patterns GRASP sont des patrons créés par Craig Larman qui décrivent des règles pour affecter les responsabilités aux classes d'un programme orienté objet pendant la conception, en liaison avec la méthode de conception BCE (Boundary Control Entity), en français MVC (Modèle Vue Contrôleur).

Les patterns GRASP ne sont pas des patterns logiciels comme ceux du [Gang of Four](http://fr.wikipedia.org/wiki/Patron_de_conception#Patrons_de_conception_du_GoF), GRASP signifie « General Responsibility Assignment Software Pattern » que l’on peut traduire en français par « patterns généraux d’affectation des responsabilités ».  
Ce sont des principes à appliquer, dans la conception orientée objet, qui permettent au développeur de maîtriser le plus rapidement possible les concepts de base de la  POO.

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 fourni une solution.

Exemple : il faut créer une méthode statique de la classe qui retourne l’instance unique ou « S*ingleton* ».

Les patterns GRASP sont au nombre de neuf :

1. Créateur
2. Expert en information
3. Faible couplage
4. Contrôleur
5. Forte cohésion
6. Polymorphisme
7. Fabrication pure
8. Indirection
9. Protection des variations

**Créateur**

Ce pattern permet de répondre à la question « qui doit avoir la responsabilité de créer une instance d’une classe donnée ? ».

Une affectation de cette responsabilité de manière judicieuse favorise un faible couplage, augmente la clarté du code et l’encapsulation.  
Si un objet agrège, enregistre, utilise étroitement, ou contient les données d’initialisation d’un autre objet alors il est un bon candidat pour la création de cet objet.

Pattern logiciels utiles : Fabrique concrète, Fabrique abstraite

**Expert en information**

Ce pattern permet d’affecter des responsabilités aux objets en répondant à la question « qui doit avoir la responsabilité de [connaître, faire, etc.] … ? »

Il faut donc affecter cette responsabilité à l’expert en information : l’objet qui contient les informations nécessaires à cette responsabilité. Par exemple un objet Panier à la responsabilité de connaître le total d’un achat, et un objet ArticlePanier aura la responsabilité de connaître le sous-total d’un article du panier.

**Faible couplage**

Ce pattern permet de garder à l’esprit, lors de la conception, le fait qu’un objet ne doit pas être dépendant d’un trop grand nombre d’éléments (classes, services, …). Un fort couplage induisant les problèmes suivant:

* Changement de code en cas de modification de classes liées
* Difficulté de compréhension
* Réutilisation plus difficile

**Contrôleur**

Quel est le premier objet qui reçoit et coordonne une opération ? Un contrôleur est un objet qui n’appartient pas à l’interface utilisateur et qui à la responsabilité de traiter un évènement système. C’est l’objet que l’on retrouve notamment dans le pattern logiciel [MVC](http://fr.wikipedia.org/wiki/Mod%C3%A8le-Vue-Contr%C3%B4leur).

Pattern logiciels utiles : Contrôleur, Commande, Façade

**Forte cohésion**

Permet de s’assurer que les objets restent compréhensibles, faciles à gérer. De plus, il contribue au faible couplage.  
Une forte cohésion, pour un objet, est la spécialisation des responsabilités. C’est à dire qu’un objet ne doit pas se charger d’effectuer des tâches sans liens entre-elles ou trop de tâches.

**Polymorphisme**

Ce pattern permet de gérer des alternatives dépendantes de type ou encore de créer des composants logiciels insérables facilement dans un programme existant.  
Il faut alors affecter les responsabilités en utilisant des opérations polymorphes aux types pour lesquels le comportement varie : les différents types implémentent une interface commune ou sont liés hiérarchiquement à une super classe commune.  
Le pattern polymorphisme permet de gérer facilement les nouveaux comportements.

Pattern logiciels utiles : Adaptateur, Commande, Composite, Etat, Strategie

**Fabrication pure**

La conception objet est souvent une implémentation de concepts du monde réel sous forme de classes logicielles permettant de diminuer le décalage de représentation. Mais dans certaines situations, affecter des responsabilités uniquement aux classes du domaine apporte des problèmes de couplage ou de cohésion. La fabrication pure préconise l’affectation de ces responsabilités à une classe qui ne représente pas un concept du domaine. Les responsabilités affectées à cette classe de commodité prennent en charge la forte cohésion et le faible couplage si bien que sa conception est très propre, ou pure ; d’où le nom de ce pattern.  
Pour la sauvegarde d’objet en base de données, par exemple, nous pouvons appliquer ce pattern à l’aide d’une classe logicielle implémentant des méthodes d’insertion, de mise à jour recevant l’objet à sauvegarder en paramètre.

Pattern logiciels utiles : Adaptateur, Commande, Strategie

**Indirection**

Permet d’affecter une responsabilité, afin d’éviter le couplage entre différentes entités, en assignant cette tâche à un objet servant d’intermédiaire entre les composants. Par exemple, dans la fabrication pure, précédemment évoquée, la classe responsable de la sauvegarde des objets en base de données est une indirection, elle sert d’intermédiaire entre ces objets et la base de données.

Patterns logiciels utiles : Adaptateur, Pont, Façade, Observateur, Médiateur

**Protection des variations**

Permet de concevoir des objets ou systèmes n’ayant pas d’impact indésirable sur d’autre éléments.  Affecter les responsabilités autour d’une interface (interface au sens large), en identifiant les points de variation ou d’instabilités prévisibles.  
Par exemple, dans un système utilisant un service tiers pour s’acquitter d’une tâche, le point d’instabilité se trouve au niveau des différentes API externe. Le système doit être capable de s’intégrer avec d’autres services existants, ou qui seront développés dans le futur, en collaborant avec une interface stable qui permettra de masquer ces variations.

Pattern logiciels utiles : Adaptateur, Pont, Façade

La plupart de ces patterns relèvent du bon sens du développeur et se veulent assez intuitifs. Ils permettent au développeur de garder en tête les bonnes pratiques de conception logicielle. Ils ont été créés par Craig Larmann, un expert reconnu dans la conception orientée objet.

**Les diagrammes d’interaction d’UML**

**Objectif :**

Fournir une référence de la notation UML la plus utilisée, dédiée à la création des diagrammes d’interaction UML (diagrammes de séquence et diagrammes de communication).

**Introduction**

Le langage UML comprend des **diagrammes d’interaction** servant à représenter la manière dont les objets interagissent via des messages. Ces diagrammes sont utilisés dans le cadre de la **modélisation objet dynamique**. Il en existe deux principaux types : les diagrammes de séquence et les diagrammes de communication.

Ce terme «diagramme d’interaction» est la généralisation de deux types de diagrammes UML plus spécialisés :

* les diagrammes de séquence ;
* les diagrammes de communication.

Les deux types permettent d’exprimer les mêmes interactions. Les diagrammes de séquence sont ceux qui possèdent la notation la plus riche. Toutefois les diagrammes de communication ont aussi leur utilité, spécialement lorsqu’on esquisse sur un tableau blanc.

Les **diagrammes de séquence** représentent les interactions comme une sorte de « clôture » où chaque nouvel objet est ajouté à droite comme le montre la figure suivante :



**Comment pourrait-on interpréter le diagramme de séquence précédent ?**

Le diagramme de séquence précédent illustre la collaboration entre deux objets : une instance non nommée de A et une instance nommée « myB » de B. L’instance non nommée de A reçoit le message « doOne ». L’émetteur de ce message n’est pas spécifié. Au cours de l’exécution de l’opération correspondant au message « doOne », l’instance de A envoie les deux messages suivants à l’instance « myB » de B : « doTwo » et « doThree »

**Comment pourrait-on représenter ce diagramme sous forme de code ?**

Probablement que la classe A possède une méthode nommée doOne(), un attribut de type B nommé myB et que la classe B possède des méthodes nommées doTwo() et doThree(). La définition partielle de la classe A serait donc :

public class A {

private B myB = new B() ;

public void done() {

myB.doTwo();

myB.doThree();

}

//

}