Patrons de Conception GRASP

RAZAFIMAHATRATRA Hajarisena
Docteur en Informatique

Année : 2022

Patron de conception

- En génie Logiciel, un patron de conception (Design Pattern):
- Concept destiné à résoudre les problèmes récurrents suivant le paradigme objet;
- Visant à résoudre des problèmes récurrents d'architecture et de conception logiciel;
- Ce n'est pas un algorithme;
- Structure générique permetant de résoudre le problème référencé.

2

Patrons de conception GRASP (1/2)

- Créés par Craig Larman;
- GRASP: General Responsability Assignment Software Patterns (schémas généraux d'affectation des responsabilités);
- Décrivent des règles affecter les responsabilités aux classes d'un programme orienté objet pendant la conception;
- C'est la recherche du meilleur responsable pour une action;
- Les objets collaborent, ont des rôles, sont responsables;
- Une responsabilité n'est pas une méthode, mais des méthodes s'acquittent de responsabilités.

3

Patrons de conception GRASP (2/2)

- Objectifs:
  - ✓ Penser systématiquement le logiciel en termes de :
    - Responsabilités
    - Par rapport à des rôles (des objets)
    - Qui collaborent
  - ✓ Réduire le décalage entre représentation « humaine » du problème et représentation informatique;
  - ✓ Programmer objet de façon claire, méthodique, rationnelle et explicable.

4

Responsabilité en GRASP

- Responsabilité imputée à :
  - ✓ Un objet seul;
  - ✓ Un groupe d'objets qui collaborent pour s'acquitter de cette responsabilité.
- GRASP aide à :
  - ✓ Décider quelle responsabilité assigner à quel objet (à quelle classe);
  - ✓ Identifier les objets et responsabilités du domaine
  - ✓ Identifier comment les objets interagissent entre eux;
  - ✓ Définir une organisation entre ces objets.

5

Types de responsabilité

Les responsabilités:

- **Faire:** faire quelque chose (un calcul, un autre objet), déclencher une action sur un autre objet, contrôler et commander les activités d'un autre objet;
- **Savoir:** savoir les valeurs de ses propres attributs, connaitre les objets qui lui sont rattachés, savoir comment obtenir des résultats de ces objets.

6

Les 4 premiers patrons de conception GRASP

- **Expert:** Affecter la responsabilité à la classe qui détient l'information;
- **Créateur:** La responsabilité de créer une classe incombe à la classe qui agrège, contient, enregistre, utilise étroitement ou dispose des données d'initialisation de la classe à créer;
- **Faible couplage**: Affecter les responsabilités de sorte que le couplage (entre Classe) reste faible;
- **Forte cohésion**: Affecter les responsabilités de sorte que la cohésion (dans la Classe) demeure forte.

7

Expert en information (1/3)

- Assigner la responsabilité d'une requête à la classe qui détient les informations nécessaires;
- Créer la méthode là où les données se trouvent;
- Appliquer le principe «Celui qui sait le fait» ou de mettre les services avec les attributs.

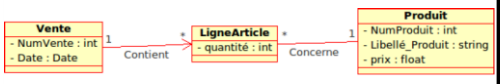
8

Expert en information (2/3)

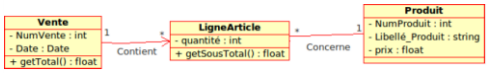
- **Problème :** Quel est le principe général d'affectation des responsabilités aux objets ?
- **Solution :**
  - ✓Affecter la responsabilité à l'expert en information;
  - ✓la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité.

9

Expert en information (3/3)



- Dans l'exemple, si on veut obtenir le total général concernant une vente, comment s'y prendre ? Qui est responsable de ce calcul ? C'est la classe Vente, qui offrira une méthode **getTotal()**.. Mais comment cette méthode pourrat-elle faire le calcul ?



10

Créateur (1/3)

- Guider l'affectation de la responsabilité de création d'objets, une tâche très fréquente dans les systèmes orientés-objets.
- Déterminer la classe qui a le plus de raisons d'être associée à la classe à créer.

11

Créateur (2/3)

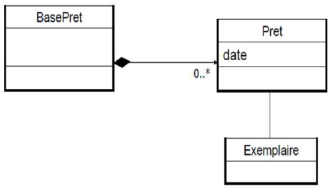
- **Problème :** Qui doit avoir la responsabilité de créer une nouvelle instance d'une classe donnée ?

- **Solution :** La classe qui répond à un (ou plusieurs) des critères suivants:
  - ✓contient (ou agrège) des objets A;
  - ✓enregistre des objets A;
  - ✓possède les données d'initialisation de A;
  - ✓utilise étroitement des objets A.

12

Créateur (3/3)

- Dans l'exemple: Qui doit être responsable de la création de Prêt dans un Bibliothèque? Base Prêt contient des Prêt : elle doit les créer.



13

Faible couplage (1/7)

- Les classes, très génériques et très réutilisables par nature, doivent avoir un faible couplage.

14

Faible couplage (2/7)

- **Problème:**
  - ✓Comment réduire l'impact des changements ?
  - ✓Comment améliorer la réutilisabilité ?
- **Solution:**
  - ✓Affecter les responsabilités des classes de sorte que le couplage reste faible.

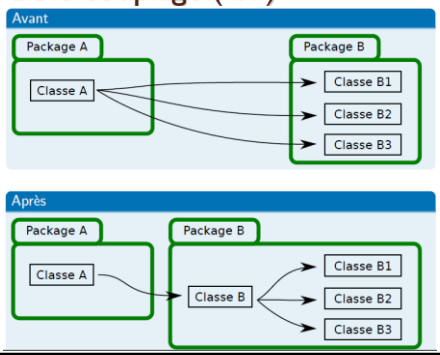
15

Faible couplage (3/7)

- Le couplage mesure à quel point les éléments sont interconnectés;
- Un couplage faible engendre les avantages suivants :
  - ✓Peu de dépendances entre les classes;
  - ✓Un changement dans une classe a un faible impact sur les autres classes;
  - ✓Forte ré-utilisabilité potentielle.

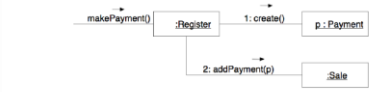
16

Faible couplage (4/7)



17

Faible couplage (5/7)

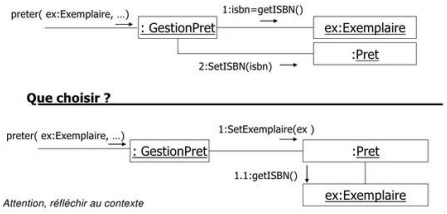


Que choisir ?

18

Faible couplage (6/7)

- Pour l'application de bibliotheque, il faut mettre l'ISBN d'un Exemple dans le Pret.



Que choisir ?

Attention, réfléchir au contexte

19

Faible couplage (7/7)

- Problèmes du couplage fort :
  - ✓Un changement dans une classe force à changer toutes ou la plupart des classes liées;
  - ✓Les classes prises isolement sont difficiles à comprendre;
  - ✓Réutilisation difficile : l'emploi d'une classe nécessite celui des classes dont elle dépend;
  - ✓Bénéfices du couplage faible Exactement l'inverse.

20

Forte cohesion (1/5)

- La forte cohésion favorise :
  - ✓la compréhension de la classe;
  - ✓la maintenance de la classe;
  - ✓la réutilisation des classes ou modules.

21

Forte cohesion (2/5)

- **Problème:** Comment s'assurer que les objets demeurent compréhensibles et simples, tout en contribuant à diminuer le couplage?
- **Solution:** Attribuer les responsabilités de telle sorte que la cohésion reste forte.

22

Forte cohesion (3/5)

- Problèmes des classes à faible cohésion:
  - ✓Elles effectuent :
    - Trop de taches
    - des taches sans lien entre elles
  - ✓Elles sont :
    - Difficiles à comprendre.
    - Difficiles à réutiliser.
    - Difficiles à maintenir.
    - Fragiles, constamment affectes par le changement.

23

Forte cohesion (4/5)

- Une classe qui est fortement cohésive:
  - ✓a des responsabilités étroitement liées les unes aux autres
    - petit nombre de méthodes
    - fonctionnalités entre méthodes liées
  - ✓n'effectue pas un travail gigantesque
- Un test
  - ✓ Peut-on décrire la classe avec une seule phrase?

24

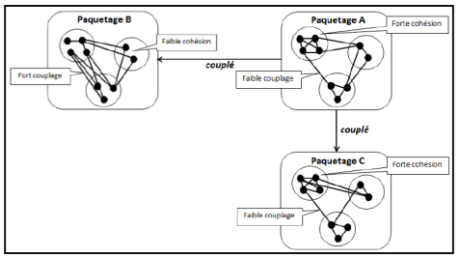
Forte cohesion (5/5)

- On rend GestionPret partiellement responsable de la mise en place des ISBN;
- GestionPret sera responsable de beaucoup d'autres fonctions.

- On délègue la responsabilité de mettre l'ISBN au prêt.

25

Relation entre Couplage et Cohésion



26

Les autres pattern GRASP

Les cinq patterns suivants sont plus spécifiques:

- **Contrôleur:** Affecter la responsabilité à une classe « façade » représentant l'ensemble d'un système ou un scénario de cas d'utilisation.
- **Polymorphisme:** Utiliser le polymorphisme pour implémenter les variation de comportement en fonction de la classe.
- **Fabrication pure:** Affecter un ensemble de responsabilités fortement cohésif à une classe artificielle ou de commodité qui ne représente pas un concept du modèle de domaine.
- **Indirection:** Affecter des responsabilités à un objet qui sert d'intermédiaire entre d'autres composants ou services pour éviter de les coupler directement.
- **Protection des variations:** Identifier les points de variations ou d'instabilité prévisibles. Affecter les responsabilités pour créer une interface stable autour d'eux.

27

Contrôleur (1/5)

Comment représenter le programme principal, le système lui-même ?

- Il faut gérer l'ensemble des actions, coordonner les différents éléments.
- Il faut une classe Contrôleur, qui déléguera ensuite aux divers objets spécialisés, et récupérera les résultats de leurs actions.
- Il ne s'agit pas de l'interface homme machine. Le pattern Contrôleur consiste en l'affectation de la responsabilité de la réception et/ou du traitement d'un message système à une classe.
- Il reçoit les demandes, et le redirige vers la bonne classe, celle qui en a la responsabilité.

28

Contrôleur (2/5)

- **Problème:** Quel objet a la responsabilité de recevoir les instructions de la couche Présentation?
- **Solution:** Donner la responsabilité à un objet qui représente le système ou qui représente un cas d'utilisation

29

Contrôleur (3/5)

- 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 (pas graphique, mais de gestion);
  - ✓ La classe **représente le scénario** issu d'un cas d'utilisation. On le nomme en général « Session », « gestionnaire » ou « coordonnateur »;
  - ✓ Elle est chargée de **traiter tous les événements** systèmes contenus dans un scénario de cas d'utilisation.

30

Contrôleur (4/5)

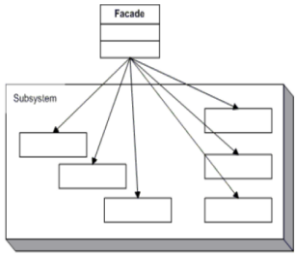
Contrôleur et 'Façade':

- Parfois, si il y a **beaucoup de cas d'utilisation**, ou **beaucoup d'actions** à coordonner, pour éviter de perdre le Pattern 'cohésion forte', on peut utiliser une classe Façade, qui utilisera des Contrôleurs qui géreront ensuite des cas spécialisés.
- Ainsi, sur une grosse application, si l'ensemble des actions possibles devenait trop important, on pourrait imaginer une **façade générale** qui utiliserait les **contrôleurs** 'GestionClients', 'GestionCommandes 'GestionProduits'.

31

Contrôleur (5/5)

Un contrôleur Façades et ses controleurs:



Le Contrôleur (ici, la version Façade, qui appelle d'autres sous contrôleurs...)

32

Polymorphisme (1/4)

- Concept idéal pour réaliser une variation du comportement des objets en fonction de leur type;
- Utile dans le cadre de prévisions de remplacement ou d'ajout de composants.

33

Polymorphisme (2/4)

- **Problème:**
  - ✓Comment gérer des alternatives dépendantes des types ?
  - ✓Comment créer des composants logiciels « enfichables » ?
- **Solution:**
  - ✓ Affecter les responsabilités aux types (classes) pour lesquels le comportement varie;
  - ✓ Utiliser des opérations polymorphes.
- **Polymorphisme:**
  - ✓ Donner le même nom à des services dans différents objets;
  - ✓ Lier le « client » à un supertype commun.

34

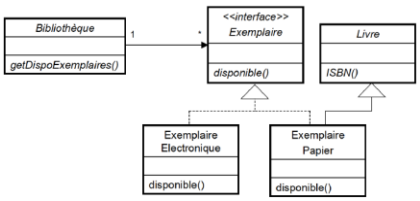
Polymorphisme (3/4)

- Principe:
  - ✓Tirer avantage de l'approche OO en sous-classant les opérations dans des types dérivés de l'Expert en information;
  - ✓L'opération nécessite à la fois des informations et un comportement particuliers.
- Mise en œuvre:
  - ✓Utiliser des classes abstraites pour définir les autres comportements communs; s'il n'y a pas de contre-indication (héritage multiple)
  - ✓Utiliser des interfaces pour spécifier les opérations polymorphes
  - ✓Utiliser les deux

35

Polymorphisme (4/4)

- Dans l'exemple d'une Bibliothèque: Qui doit être responsable de savoir si un exemplaire est disponible ?



36

Fabrication pure (1/4)

- Une réalisation ne se déduit pas du domaine;
- Utilisé lorsque les patrons ne parviennent pas à assurer les principes de forte cohésion et de faible couplage.

37

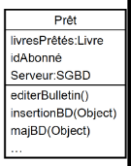
Fabrication pure (2/4)

- Problème:
  - ✓Que faire pour respecter le Faible couplage et la Forte cohésion;
  - ✓ Que faire quand aucun concept du monde réel (objet du domaine) n'offre de solution satisfaisante ?
- Solution:
  - ✓Affecter un ensemble fortement cohésif à une classe artificielle ou de commodité, qui ne représente pas un concept du domaine;
  - ✓Entité fabriquée de toutes pièces.

38

Fabrication pure (3/4)

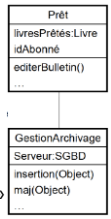
- Problème: les instances de Prêt doivent être enregistrées dans une BD.
- Solution initiale (d'après Expert):
  - ✓Prêt a cette responsabilité
  - ✓cela nécessite
    - un grand nombre d'opérations de BD
      - → Prêt devient donc non cohésif
    - de lier Prêt à une BD
      - → Le couplage augmente pour Prêt



39

Fabrication pure (4/4)

- Constat:
  - ✓L'enregistrement d'objets dans une BD est une tâche générique utilisable par de nombreux objets;
  - ✓Pas de réutilisation, beaucoup de duplication.
- Solution avec Fabrication pure:
  - ✓Créer une classe artificielle GestionArchivage
- Avantages:
  - ✓ Gains pour Prêt
    - Forte cohésion et Faible couplage
  - ✓ Conception de GestionArchivage « propre »
    - Relativement cohésif, générique et réutilisable



40

Indirection (1/4)

- Lorsque certaines classes ont **trop de responsabilités**, la cohésion tend à baisser (Patron forte cohésion) et la classe **devient plus difficile** à comprendre et à maintenir.
- Une solution à ce type de problème est de **déléguer** une partie des traitements et des données à d'autres classes en ajoutant un niveau d'**indirection**.

41

Indirection (2/4)

- **Problème:**
  - ✓Où affecter une responsabilité pour éviter le couplage entre deux entités (ou plus) ?
    - de façon à diminuer le couplage (objets dans deux couches différentes);
    - de façon à favoriser la réutilisabilité (utilisation d'une API externe) ?
- **Solution:**
  - ✓Créer un objet qui sert d'intermédiaire entre d'autres composants ou services
    - l'intermédiaire crée une indirection entre les composant;
    - l'intermédiaire évite de les coupler directement.

42

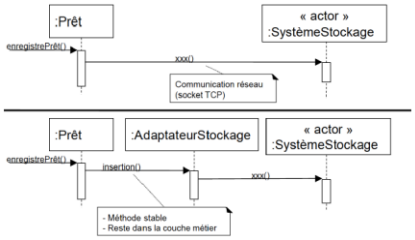
Indirection (3/4)

- **Utilité:**
  - ✓ Réaliser des adaptateurs, façades, etc. (Patron Protection des variations) qui s'interfacent avec des systèmes extérieurs;
  - ✓ Exemples : proxys, DAO, ORB;
  - ✓ Réaliser des inversions de dépendances entre packages.
- Mise en œuvre:
  - ✓ Utilisation d'objets du domaine;
  - ✓ Création d'objets;
  - ✓ Classes : cf. Fabrication pure;
  - ✓ Interfaces : cf. Fabrication pure + Polymorphisme.

43

Indirection (4/4)

- **Bibliothèque : accès à un système de stockage propriétaire**



44

Protection des variations (1/8)

- Assigner les responsabilités afin que des variations dans des classes (instabilité) ne produisent aucun effet indésirable sur le reste du système;
- Utilisé en prévision d'évolutions de l'application logicielle, afin d'éviter que de nouvelles fonctionnalités ne viennent perturber l'existant.

45

Protection des variations (2/8)

- **Problème:**
  - ✓Comment concevoir des objets, sous-systèmes, systèmes pour que les variations ou l'instabilité de certains éléments n'aient pas d'impact indésirable sur d'autres éléments ?
- **Solution:**
  - ✓Identifier les points de variation ou d'instabilité prévisibles;
  - ✓Affecter les responsabilités pour créer une interface (au sens large) stable autour d'eux (indirection).

46

Protection des variations (3/8)

- Mise en oeuvre:
  - ✓ Cf. Patterns (Polymorphisme, Fabrication pure, Indirection).
- Exemples de mecanismes de PV:
  - ✓ Encapsulation des donnees, brokers, machines virtuelles...
- Exercice:
  - ✓ Point de variation = stockage de Pret dans systemes differents;
  - ✓Utiliser Indirection + Polymorphisme

47

Protection des variations (4/8)

- Ne pas se tromper de combat:
  - ✓Prendre en compte obligatoirement les **points de variation**
    - Nécessaires car identifiés dans le système existant ou dans les besoins
  - ✓Gérer sagement les **points d'évolution**
    - Point de variation futurs, "spéculatifs": à identifier (ne figure pas dans les besoins)
    - Pas obligatoirement à implémenter
    - Le cout de prévision et de protection des points d'évolution peut dépasser celui d'une reconception
  - ✓ → Ne pas passer trop de temps à préparer des protections qui ne servirons jamais

48

Protection des variations (5/8)

- Diffèrent niveaux de sagesse:
  - ✓le novice conçoit fragile;
  - ✓le meilleur programmeur conçoit tout de façon souple et en généralisant systématiquement;
  - ✓l'expert sait évaluer les combats à mener.
- Avantages:
  - ✓Masquage de l'information;
  - ✓Diminution du couplage;
  - ✓Diminution de l'impact ou du cout du changement.

49

Protection des variations (6/8)

Ne pas parler aux inconnus (1/3):

- **Cas particulier de Protection des variations:**
  - ✓ protection contre les variations liées aux évolutions de structure des classes du système.
- **Problème:**
  - ✓ Comment éviter de connaitre la structure d'autres objets indirectement?
  - ✓ Si un client utilise un service ou obtient de l'information d'un objet indirect (inconnu), comment le faire sans couplage?
- **Solution:**
  - ✓ Eviter de connaitre la structure d'autres objets indirectement
    - se limiter aux objets connus directement.
  - ✓ Affecter la responsabilité de collaborer avec un objet indirect à un objet que le client connait directement pour que le client n'ait pas besoin de connaitre ce dernier

50

Protection des variations (7/8)

Ne pas parler aux inconnus (2/3):

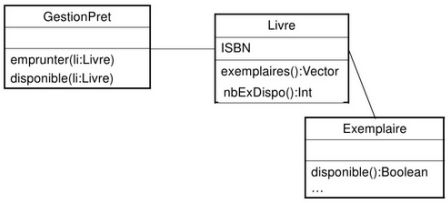
- **Cas général à éviter:** a.getB().getC().getD().methodeDeD();
  - ✓ A est dépendant de B, C et D
    - si l'une des méthodes de la chaine disparait, A devient inutilisable
    - tout changement de methodeDeD() peut avoir une conséquence sur A
- **Préconisation ("Loi de Demeter"):**
  - ✓ Depuis une méthode, n'envoyer des messages qu'aux objets suivants
    - l'objet *this*(self)
    - un paramètre de la méthode courante
    - un attribut de *this*
    - un élément d'une collection qui est un attribut de *this*
    - un objet créer à l'interieiur de la méthode
- **Implication**
- Ajout d'opération dans les objets directs pour servir d'opérations intermédiaires

51

Protection des variations (8/8)

Ne pas parler aux inconnus (3/3):

- Comment implémenter disponible() dans GestionPret?



52