DESIGN PATTERN : PATRON DE CONCEPTION

I) design pattern et anti-patterns

A) buts

Les patterns datent de 1977. Ils ont été formalisés par Christopher Alexander (***anthropologue***). Ce sont des solutions génériques adaptables à chaque cas particulier. il appliquait les design pattern aux science humains et à l'architecture. Design pattern pour l'informatique arriva en 1995. Le gang of four ont décidé de publier sur des design pattern sur l'informatique.

réutilisable

code maintenance possible

peu de lien entre les classes

le design pattern a un unique nom. il a un problème à résoudre. il a une solution. il y a des conséquences (possible limitations, des complexités etc...). Dans tous les cas, les design pattern doivent être indépendant du langage. Quand on utilise certain langage plus évolué, on peut adapté le design pattern.

B) les anti-patterns

ils sont non utilisables fait par un stagiaire à un problème ponctuel. Cela peut ralentir le système, le rendre non évolutif, quoiqu'il en soit ça coutera plus cher. Certaines erreurs ont des noms et des catégorie:

- réinventer la roue: et elle ne sera pas ronde, soit cabossé (ça marche mais moins bien) ou bien une roue carrée (vous avez résolu un problème qui n'est pas le votre).

- tout ce qui concerne la ré écriture du code: quand il y a un copié collé, c'est un indicateur de problème dans le code. on ne doit pas dupliquer le code.

- l'objet divin: une classe qui fait tout. ou l'inverse on découpe trop

- gaffe au multi thread

- l'ancre de bateau: faire avancer un bateau ancré: on a du code qui n'est plus utilisé mais qui reste dedans. il faut enlever le code plus utilisé. mettre le code non utilisé en commentaire.

- la couvée de lave: on a du code neuf solidifié (non modifiable). il est tellement central qu'il est bloqué. il est plus adaptable.

il faut tous les éviter.

II) Patterns GoF

A) GoF

Erich Gamma, c'est un informaticien suisse qui a co-auteur Junit et du JDT d'éclipse. il travaille à IBM.

Richard Hemm, il travaillait aussi chez IBM et maintenant il travaille au Boston Consulting Group.

Ralph Johnson, un chercheur en informatique qui travaille à l'université de l'illinois, c'est un créateur du langage small talk.

John Vlissidss, chercheur à stanford, puis il a travaillé pour IBM et depuis il est mort.

Ils ont formalisé les design pattern au sain d'un livre sortie en 1995 nommé design pattern: elements of réutilisable(mais en anglais) orient objected software. 23 design pattern au début, il y en a un en plus maintenant, il y a donc 24 design pattern, mis dans 3 catégories. Ces design pattern sont considérés comme orthogonals (ça correspond à des problèmes différents) dans la théorie.

B) les patterns de création

comment on va créer des objets de manière dynamique. on ne sait pas quelle classe appelé, mais à l'exécution on saura quelle classe utilisée. il y a 5 patterns qui répondent à ce problème: les objets dynamiques.

1) Factory: la fabrique

problème: créer un objet d'une classe, mais on demande à l'utilisateur lequel il veut. on ne connait pas la classe à la compilation mais à l'exécution. on va donc créer une classe qui va créer les objets.

gob

elfe

nain

mage

|  |
| --- |
| Fabrique |
| getFPerso(type):perso |

abs(perso)

on peut faire un index

il y a quelques inconvénients: ça reste imparfait, il on a juste déplacer le switch du main à une classe différente.

2) la fabrique abstraite (abstract factory)

imaginons un framework multiplateforme de fenêtrage (ex: GNOME). il faut une fabrique qui va fabriqué les fabriques pour chaque plateforme

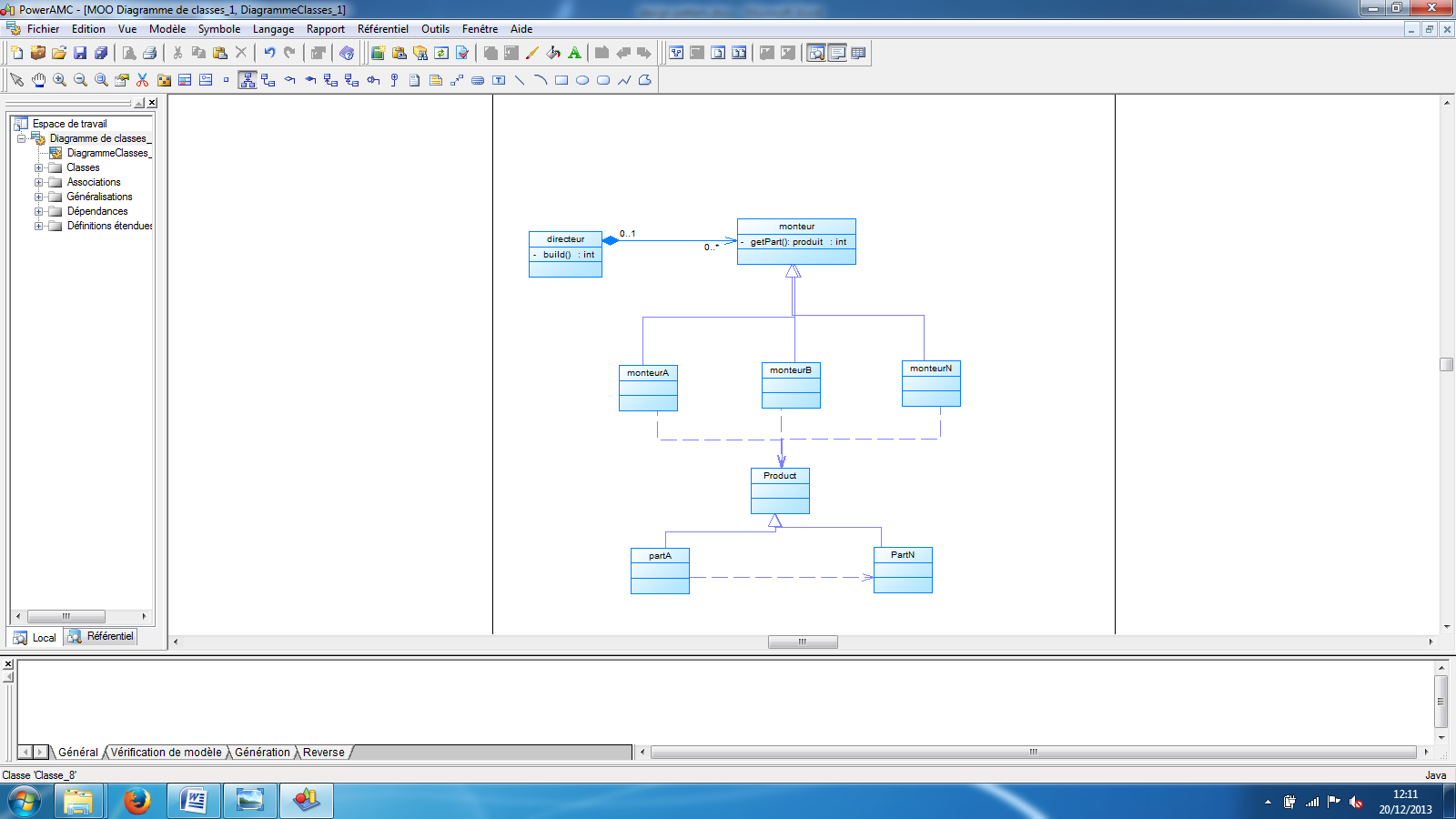
//INSERER LE DESSIN PHOTO

inconvénient: il y a beaucoup de liaison

3) Monteur (Builder)

on a besoin de créer des objets complexes composés de plusieurs parties. on fonction de la demande on aura pas le meme produit à la sortie mais on utilisera la même chaine de construction. on aura un patron personnalisable.

On va utiliser un monteur et on va faire un directeur qui va dirigé la chaine de montage.



inconvénient: le directeur va très vite être énorme. Au niveau des performances, ça sera moins efficace qu'un tout divin.

4) le singleton

son but est de s'assurer qu'un objet peut etre instancié 1 et 1 seul fois



GetInstance () : if (instance == nul) {

instance = new singleton();

}

return instance;

on ne met pas de singleton pour une classe statique (ex classe statique: Maths() ).

5) Le prototype.

certains objets vont prendre énormément de temps à instancier (ex: une instance qui lit un fichier de 10 000 lignes ou chercher des données en BDD, charger les textures, les polices dans un jeu).

A l'initialisation, on crée un objet (un prototype) qui charge les données, et on va le cloner quand on a besoin d'un nouvel objet. le prototype est générique et s'occupe de charger les données, puis on va cloner ce prototype au besoin.



Problème: il faut s'assure que le clonage soit moins long que la création d'un objet sans clonage. il faut vérifier que c'est utile. Cette méthode est aussi très sensible au contenu du prototype: on change un attribut dans le prototype il faut changer la méthode clone(). il faut faire des compromis. les langages bas niveaux, il faut faire attention aux pointeurs, références, etc..

C) Pattern de structure

ils sont au nombre de 7:

ces design pattern permettent de voir comment les classes vont s'imbriquer entre elle. Ce n'est pas du dynamique.

1- Adaptateur (adapter ou le wrapper)

on passe dune interface a une autre

ex: on fait un logiciel en C avec la sdl et on lit un article sur Qt. on veut traduire les méthode d'une librairie à une autre. il faut donc adapter afin que ça fonctionne.



l'adaptateur doit connaître la cible et l'adapté. on a un couplage fort avec les 2 interfaces.

Une variante est d'utilisé un lien de composition.

2- le pont (bridge)

on prévoit avant plusieurs implémentation possible

permet de séparer l'abstraction et l'implémentation de l'objet. on le réfléchit avant de coder. on veut pouvoir choisir l'implémentation.

Par ex: on fait un logiciel sans savoir quel type de bdd on utilise.



le i doit respecter implémentation, et implémentation doivent avoir la même interface que abstraction. tout est lié.

on est limité à un sous ensemble des fonctionnalités. on est limité au set commun le plus petit des implémentations choisies.

3- objet composite (composite)

c'est l'objet qui contient des objets qui contient des objets

le but est de gérer des objets hiérarchiques (des paniers dans des paniers dans des paniers avec des objets dedans). genre un tiroir avec un rangeur dans le tiroir.



limite: il va complexifier l'ensemble., à n'utiliser que quand la structure n'est pas régulière

4- Le décorateur (decorator)

un seul objet avec des surcouche.

c'est un objet qu'on décore.



quand on peut personnaliser c'est du décorateur, quand on construit c'est de l'objet composite.

La question c'est : est ce contenant/contenu, ou bien un objet qu'on va personnaliser.

limite: ne pas confondre avec le composite.

5- la façade (facade)

pour simplifier on a une classe qui se débrouille pour accéder aux différentes classes

on cache la complexité en la cachant avec une façade.

pas d'héritage entre la façade et le reste.



Exemple: distributeur de billet.

limite: couplage fort

6- le poids plume

quand on a besoin de beaucoup d'objet identiques partagé qui n'ont pas d'état.

fabrique de petit singleton sans état.



limite: l'état: l'objet poids plume ne peut pas avoir d'état.

assez complexe a mettre en oeuvre au départ.

7- le Proxy

on rajoute une surcouche sécurité.

on veut protéger l'application, sans que le client le remarque.

on protège donc la classe d'origine. il vérifie les entrées/sorties.



le client veut la fonction de origine, mais va en faite appeler la méthode m1 de proxy, qui aurait du code avant et/ou après dans la méthode m1.

ça permet de modifier sans modifier la classe dorigine (probleme de droit, sécurité, code source trop complexe)

limite:

ça va ralentir l'exécution, augmenter la taille de mémoire utilisé.

il faut faire attention que la classe origine ne soit pas publique

D) Pattern de comportement.

1) chaîne de responsabilités (chain of responsability)

chaque élément qui peut gérer quelque chose connait son suivant, jusqu'à que tout soit traiter

le pdg demande a son second de faire qque chose, qui va voir le 3eme etc jusqu'au stagiaire qui va faire le boulot mal.

ex: si on clic qque part, l os filera le bébé a word, qui le refilera a la page, qui le refilera au paragraphe qui mettra le curseur a la bonne place.



il faut faire attention a ne pas faire de boucle et que le traitement soit traité.

impossible de changer l'ordre.

2) Commande (command)

on reçoit une commande et on doit la traiter

on a un système qui peut recevoir de l'extérieur des commandes, on aura une classe qui réceptionne les commandes, et qui les transmet a la bonne personne, et qui peut les orchestrer.

le but: un qui reçoit toutes les commandes, les met en attente et les envoie aux bonnes personnes.

c'est la commande qui c'est a qui elle doit s'appliquer:



facile a faire évoluer

limite: il y a un couplage. il y a forcement une classe qui va connaitre les commandes qui existent

3) interpréteur (interpreter)

Permet de décomposé une expression en sous-expression.

le but est différent de l'objet composite



limite : structure complexe. Si l'expression n'est pas correct, si il y a une erreur on a du mal a les traiter.

4) itérateur (iterator)

Quand on a des collections à parcourir, sans savoir comment elles sont codées.

c'est le for each

on parcourt une liste.



chaque collection connait son itérateur.

limite : complexe a mettre en oeuvre, for couplage chaque collection doit connaître son itérateur.

complexe a comprendre, contre-intuitif.

5) Médiateur (mediator)

Dans un programme, il y a beaucoup de classes qui doivent communiquer entre elles.

Afin de simplifier, on va utiliser un médiateur. Ex: la poste, hey coco cette fois ci c'est pas moi.

il est très utile sur un système client/serveur

limite :

couplage fort

faut le penser avant.

point de congestion possible: goulot d'étranglement sur le médiateur.

le médiateur peut avoir à lire le message, donc il est couplé aux classes et aux messages.

6) le Mémento

Dans un programme, on laisse souvent à l'utilisateur d'annuler ce qu'il a fait. On va pas tout stocker, on va stocker juste ce qu'on a besoin pour revenir en arrière.

On aura donc un caretaker, qui va stocker les mémentos sans savoir ce que c'est.

Il va aussi pouvoir supprimer les memento les plus vieux.



limite: performance pas top, la mémoire va être saturée.

le caretaker doit savoir gérer l'historique et savoir faire le tri.

il faut que l'objet sache écrire un mémento et surtout savoir le lire.

7) l'observateur (observer)

on veut simplement suivre un phénomène et agir en fonction de ce phénomène (ex: boussole indique le nord peut importe si on se tourne).

exemple : on\_mouseOver, on\_mouseClick etc...

les classes s'abonnent a un observateur qui les préviendra quand il y a un changement.



on augmente les performances

limite :

il n'y aucun moyen de savoir qui écoute, il peu avoir des notifications dans le vide.

c'est pas facile de savoir ce qu'il faut mettre dans l'évènement. il faut donc réfléchir à ce qu'on met dedans.

8) Etat (state).

imaginons un jeu de voiture. selon l'état de ma voiture on veut obtenir des choses différentes.

on veut pouvoir lire les états sans avoir un couplage fort



limite : perte de performance.

9) Stratégie (strategy)

on veut faire un simulateur de mare a canard. Il y a plusieurs type de canard. Ces canard savent voler nager et faire coincoin. il y a des canards qui font des coincoins différents.



limite: chaque canard doit savoir quelle stratégie ils ont. il y a donc un couplage.

ralentissement du systeme.

10) patron de méthode (template method)



difficile a faire évoluer

on ne peut pas partager les méthodes entre les 2 classes filles

11) visiteur (visitor)

on a plusieurs façon de faire les methodes.



le client doit savoir quel visiteur il appelle.

12)Fonction de Rappel (callback)

ce pattern n'était pas dans la 1ere version du GoF.

il ne possède pas de schéma UML.

tous les langages ne passent pas de méthodes en paramètres.