MAI NFE103
Année 2013-2014

Les Modèles de Conception

F.-Y. Villemin, CNAM

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/MAI/index.html

Plan

- Conception d'objets réutilisables
- Modèles de conception
- Patrons
- Fabrique abstraite
- Adaptateur
- Observateur
- Procuration (Proxy)
- Conclusion

Conception d'objets réutilisables

Concevoir des logiciel orienté-objet est difficile

Concevoir du logiciel orienté-objet réutilisable, l'est plus encore

Il faut:

- déterminer les objets appropriés
- les décomposer en classes, avec le niveau de détail adéquat
- définir les interfaces des classes et les hiérarchies d'héritage
- établir les relations clés entre elles

Une conception doit être spécifique du problème à résoudre et être assez générale pour répondre aux problèmes et aux exigences futures

Modèles de conception

Christopher Alexander décrit chaque modèle comme :

- un problème qui se manifeste constamment dans notre environnement
- le cœur de la solution de ce problème réutilisable

Un modèle possède quatre éléments essentiels:

1. Le nom de modèle
2. Le problème décrit les situations où le modèle s'applique, expose le sujet à traiter et son contexte
3. La solution décrit les éléments qui constituent la conception, les relations entre eux, leurs part dans la solution, leur coopération
4. Les conséquences sont les effets résultant, de la mise en du modèle et les variantes de compromis que celle-ci entraîne

Modèles de conception

D. Schmidt (1995) : joueur d'échec versus développeur de logiciel

Devenir un maître aux échecs

1. apprendre les règles: nom des pièces, mouvement légal. . .
2. apprendre les principes, les valeur relative des pièces...
3. étudier le jeu des autres maîtres: certaines parties contiennent des modèles qu'il faut comprendre, mémoriser.
4. il y a des milliers de modèles de parties

Devenir un maître dans la conception logiciel

1. apprendre les règles: les algorithmes, les structures de données, les langages...
2. apprendre les principes de programmation orientée objet...
3. étudier aussi les modèles (designs) des autres maîtres, les apprendre, les mémoriser...
4. il y a des centaines de modèle de conception

Modèles de conception

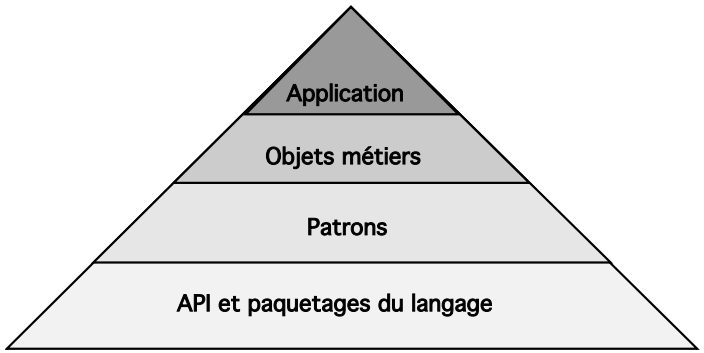
Un objet métier ("framework") est un ensemble de classes qui coopèrent et permettent des conceptions réutilisables dans des catégories spécifiques de logiciels

Exemple:

- fabrication des éditeurs graphiques dans des domaines différents, comme le dessin artistique, la composition musicale, ou la CAO mécanique
- conception de compilateurs pour divers langages de programmation et machines cibles
- création d'applications de modélisations financières
- création d'applications de traitements de dossiers médicaux, de soins...

Modèles de conception

Hiérarchie des objets réutilisables (plus bas, plus général):



Modèles de conception

Un objet métier ("framework") correspond au meilleur compromis trouvé par les concepteurs experts dans son domaine d'application:

- il impose une architecture à une application
- il définit sa structure globale (partitionnement en classes et en objets)
- il déduit la tâche de contrôle, et donc,
- il déduit les responsabilités essentielles (la façon de collaborer des classes et des objets)
- il a la maîtrise des décisions de conception courantes dans son domaine d'application
- il est plus concret et moins élémentaire que les "patrons"
- il est construit en utilisant des "patrons"

Patrons

Un "patron" ("pattern") correspond à la solution à un problème dans un certain contexte

Les "patrons" sont comme ceux utilisés en couture (exemple: tailler une chemise), ils répondent à un problème

L'utilisation du patron devra être modifié en fonction du contexte dans lequel il est utilisé (taille de la personne, tissu utilisé, etc ...)

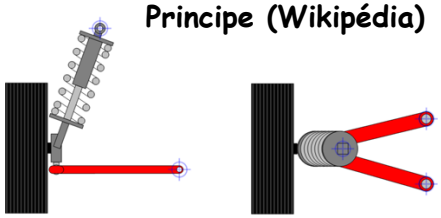
Les patrons permettent une réutilisation des connaissances en apportant une certaine formalisation concernant les explications pour une solution à un problème

⇒ transmission de la connaissance facilitée

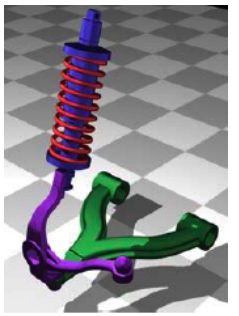
Patrons

La suspension Mac Pherson :
Inventée par Earle S. MacPherson chez General Motors en 1947, adoptée pour la première fois sur la Ford France Vedette en 1949
Aujourd'hui, utilisée par la quasi-totalité des automobiles (sauf quelques véhicules à très hautes performances) pour les trains avant

Principe (Wikipédia)



Exemple de réalisation sur un train avant



Patrons

Le Formalisme des patrons de Christopher Alexander :

Si vous vous trouvez dans un contexte

- par exemple : exemple
- avec ces problèmes
- impliquant ces forces

Alors pour les raisons suivantes

- appliquer les formes de modélisation et/ou les règles
- pour construire une solution
- amenant dans un nouveau contexte et vers d'autres patrons

Patrons

Patrons de conceptions :

- s'occupent de l'organisation d'un programme
- sont indépendants du langage utilisé

Des APIs ont proposées pour certains langages, les idées présentes dans ces patrons sont applicables à d'autres environnements

Patrons de codages :

- sont spécifiques à un langage particulier
- permettent de résoudre des difficultés techniques

Patrons

Format de E. Gamma, R. Helm, R. Johnson et J. Vlissides:

- but du patron
- motivations du patron
- situations dans lesquelles le patron peut s'appliquer
- structure du patron
- solution proposée (participants, collaborations)
- façon dont le patron répond aux objectifs (détails)
- conséquences d'utilisation du patron
- exemples
- patrons qui sont liés à celui qui vient d'être décrit

Patrons

La forme la plus générale pour décrire un patron est:

- Le nom du patron : il doit être bien choisi
- Le problème que le patron essaye de résoudre : Quand on sait quel problème le patron résout, on sait quand l'appliquer
- Le contexte : Un patron résout un problème dans un contexte donné et n'a pas de sens dans un autre contexte
- Les forces impliquées, les efforts à effectuer ou les compromis
- La solution : cette partie décrit la structure, le comportement de la solution
- Des exemples
- Le contexte résultant : ce qui reste à résoudre. Il doit aussi montrer de quel façon le contexte a été changé par le patron
- Le "Design Rationale" explique d'où le patron provient, pourquoi il marche et aussi pourquoi les experts l'utilisent

Catalogue des Patrons

Les patrons de conceptions sont classés suivant le domaine où ils s'appliquent et leur rôle :

		ROLE		
		Créateur	Structurel	Comportement
DOMAINE	CLASSE	Fabrication	Adaptateur(classe)	Interprete Patron de méthode
	OBJET	Fabrique abstraite Monteur Prototype Singleton	Adaptateur(objet) Pont Décorateur Facade Composite Poids mouche Procuration	Chaine de responsabilité Commande Itérateur Médiateur Mémento Observateur Etat Stratégie Visiteur

Patrons

Fabrique Abstraite (Abstract Factory) : Fournit une interface, pour créer des familles d'objets apparentés ou dépendants, sans avoir à spécifier leurs classes concrètes

Adaptateur (Adapter) : Convertit l'interface d'une classe en une interface distincte, conforme à l'attente de l'utilisateur. L'adaptateur permet à des classes de travailler ensemble, qui n'auraient pu le faire autrement pour cause d'interfaces incompatibles

Pont (Bridge) : Découple une abstraction de son implémentation associée, afin que les deux puissent être modifiés indépendamment

Monteur (Builder) : Dans un objet complexe, dissocie sa construction de sa représentation, de sorte que, le même procédé de construction puisse engendrer des représentations différentes

Patrons

Chaîne de responsabilité (Chain of responsibility) : Permet d'éviter de coupler l'expéditeur d'une requête à son destinataire, en donnant la possibilité à plusieurs objets de prendre en charge la requête. Pour ce faire, il chaîne les objets récepteurs, et fait passer la requête tout au long de cette chaîne usqu'à ce qu'un des objets la prenne en charge

Commande (Command) : Encapsule une requête comme un objet, ce qui permet de faire un paramétrage des clients avec différentes requêtes, files d'attente, ou historiques de requêtes, et d'assurer le traitement des opérations réversibles

Composite (Composite) : Organise les objets en structure arborescente représentant la hiérarchie de bas en haut. Le Composite permet aux utilisateurs de traiter des objets individuels, et des ensembles organisés de ces objet,s de la même façon

Patrons

Décorateur (Decorator) : Attache des responsabilités supplémentaires à un objet de façon dynamique. Il permet une solution alternative pratique pour l'extension des fonctionnalités, à celle de dérivation de classes

Façade (Facade) : Fournit une interface unifiée pour un ensemble d'interfaces d'un sous-système. Façade définit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser

Fabrication (Factory method) : Définit une interface pour la création d'un objet, tout en laissant à des sous-classes le choix de la classe à instancier. Une fabrication permet de déférer à des sous-classes les instanciations d'une classe

Poids Mouche (Flyweight) : Assure en mode partagé le support efficace d'un grand nombre d'objets à fine granularité

Patrons

Interprète (Interpreter) : Dans un langage donné, il définit une représentation de sa grammaire, ainsi qu'un interprète qui utilise cette représentation pour analyser la syntaxe du langage

Itérateur (Iterator) : Fournit un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous-jacente

Médiateur (Mediator) : Définit un objet qui encapsule les modalités d'interaction de divers objets. Le médiateur favorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres

Patrons

Mémento (Memento) : Sans violer l'encapsulation, acquiert et délivre à l'extérieur une information sur l'état interne d'un objet, afin que celui-ci puisse être rétabli ultérieurement dans cet état

Observateur (Observer) : Définit une corrélation entre objets du type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent, en soient notifiés et mis à jour automatiquement

Prototype (Prototype) : Spécifie les espèces d'objets à créer, en utilisant une instance de type prototype, et crée de nouveaux objets par copies de ce prototype

Procuration (Proxy) : Fournit un subrogé ou un remplaçant d'un autre objet, pour en contrôler l'accès

Singleton (Singleton) : Garantit qu'une classe n'a qu'une seule instance, et fournit à celle-ci, un point d'accès de type global

Patrons

Etat (State) : Permet à un objet de modifier son comportement lorsque son état interne change. L'objet paraîtra changer de classe

Stratégie (Strategy) : Définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Une stratégie permet de modifier un algorithme indépendamment de ses clients

Patron de méthode (Template Method) : Définit le squelette de l'algorithme d'une opération, en déléguant le traitement de certaines étapes à des sous-classes. Le patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme

Visiteur (Visitor) : Représente une opération à effectuer sur les éléments d'une structure d'objet. Le visiteur permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère

Fabrique Abstraite

Objet - Créateur

Intention : La fabrique abstraite fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes

Alias : Kit

Motivation : Boite à outils d'interfaces utilisateurs (“widgets”: fenêtres, ascenseurs de défilement et boutons) pour décors ("look-and-feel") standards (MacOS X, Microsoft Windows, KDE, Gnome...)

Décors différents ⇒ aspects et comportements spécifiques des “widgets”

Application portable d'un décor standard à l'autre: pas de widgets “codés en dur” pour un décor donné

Fabrique Abstraite

⇒ création d'une classe de base abstraite FabriqueWidgets qui définit une interface pour créer chaque variété de widget

Pour chaque variété de widgets ⇒ créer une classe abstraite dont les sous-classes concrètes implantent les versions de ces widgets conformément aux standards des décors supportés

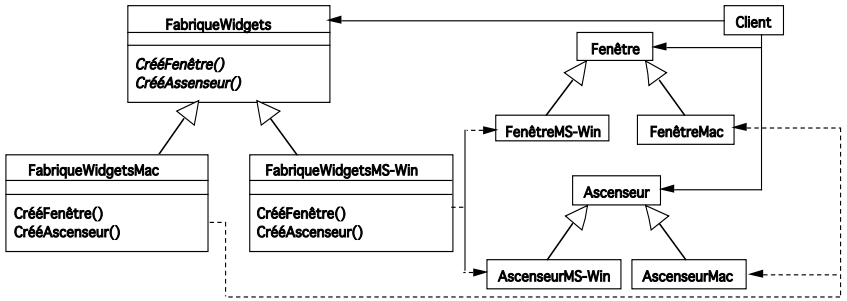
L'interface de FabriqueWidgets possède pour chaque classe abstraite de widgets, une opération qui renvoie un nouvel objet widget

Les clients appellent ces opérations pour obtenir des instances de widgets, mais sans savoir quelles classes concrètes ils utilisent.
Ainsi, le client reste-t-il indépendant du décor en vigueur

Une FabriqueWidgets renforce également l'interdépendance des classes widgets concrètes

Fabrique Abstraite

Exemple (boite à outils d'interfaces utilisateurs) “FabriqueWidgets”:



Fabrique Abstraite

Indications d'utilisation : Le modèle Fabrique Abstraite est recommandé lorsque:

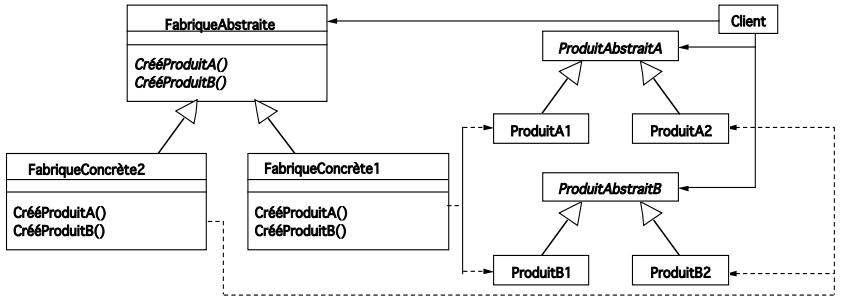
- Un système doit être indépendant de la façon dont ses produits ont été créés, combinés, et représentés.
- Un système doit être constitué à partir d'une famille de produits, parmi plusieurs.
- On souhaite renforcer le caractère de communauté d'une famille d'objet produits conçus pour être utilisés ensemble.

Collaborations : Une FabriqueAbstraite délègue la création des objets produits à sa sous-classe FabriqueConcrète:

Pour créer des objets produits différents, les clients doivent utiliser une fabrique concrète différente

Fabrique Abstraite

Structure du modèle de Fabrique Abstraite:



Fabrique Abstraite

Conséquences : Avantages et contingences du modèle de Fabrique Abstraite

1. Il isole les classes concrètes.
2. Il facilite la substitution de familles de produits.
3. Il favorise le maintient de la cohérence entre les objets.
4. Gérer de nouveaux types de produits est difficile: car l'interface de FabriqueAbstraite détermine l'ensemble des produits qui peuvent être créés ⇒ étendre l'interface de la fabrique donc modifier la classe FabriqueAbstraite et toutes ses sous-classes.

Adaptateur

Classe, Objet - Structure

Intention : Convertit l'interface d'une classe en une autre conforme à l'attente du client et permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles

Alias : Empaqueteur

Motivation : Une classe boîte à outils conçue pour la réutilisation, n'est en fait pas réutilisable, simplement parce que son interface n'est pas adaptée à celle, spécifique du domaine, imposé par l'application

Exemple : un éditeur de dessin permettant à l'utilisateur de tracer et de combiner des éléments graphiques (lignes, polygones, textes, etc.) sous forme d'images et de diagrammes. Le concept de base d'un éditeur de dessin est l'objet graphique dont la forme peut être éditée et qui peut se dessiner lui-même

Adaptateur

L'interface pour les objets graphiques est définie par la classe abstraite “Forme”. L'éditeur définit une sous-classe de Forme pour chaque espèce d'objet graphique : “FormeLigne” pour les lignes, “FormePolygone” pour les polygones...

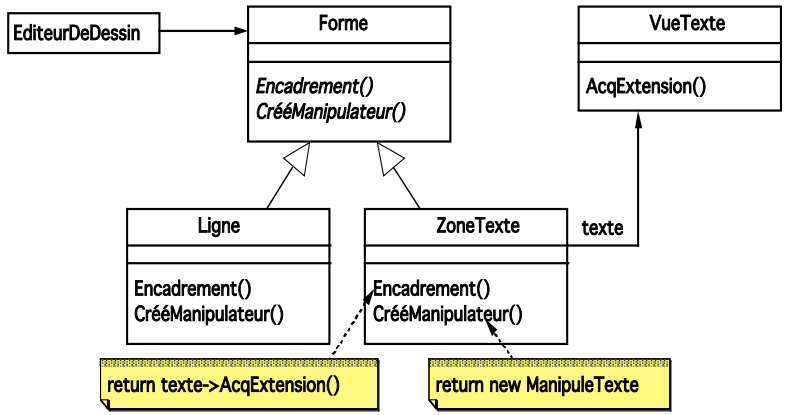
Les classes des formes géométriques élémentaires (FormeLigne, FormePolvgone) sont très simples à coder (caractéristiques de dessin et d'édition limitées)La sous-classe FormeTexte pour l'affichage et l'édition d'un texte est difficile à implanter

Il se trouve des boîtes à outils d'interfaces utilisateurs dans le commerce qui offre une classe “VueTexte” sophistiquée pour l'affichage et l'édition de textes. L'idéal serait de réutiliser VueTexte pour implantater de FormeTexte

Mais on ne peut utiliser les objets VueTexte et FormeTexte de façon interchangeable

Adaptateur

Exemple d'un éditeur de formes:



Adaptateur

Modifier les classes VueTexte afin qu'elles se conforment à l'interface de Forme, ne peut être envisagé si l'on ne dispose pas du code source de la boîte à outils

On peut définir FormeTexte de façon à ce qu'elle adapte l'interface de VueTexte à celle de Forme

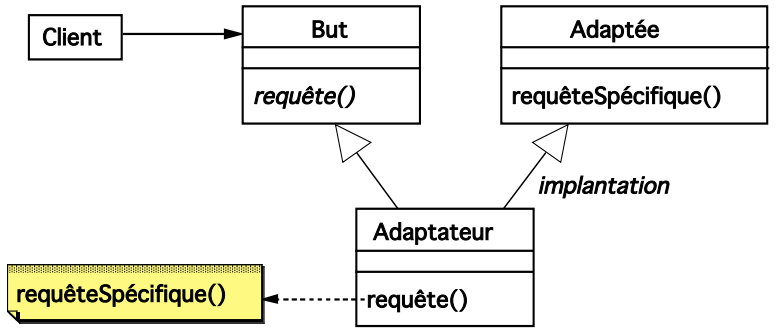
On peut réaliser cela de deux manières différentes :

1. par héritage de l'interface de Forme, et de l'implantation de VueTexte (version classes du modèle Adaptateur)
2. par composition d'une instance de VueTexte dans FormeTexte et en implantant FormeTexte en termes de l'interface de VueTexte (version objets du modèle Adaptateur)

Adaptateur

Version classes du modèle Adaptateur

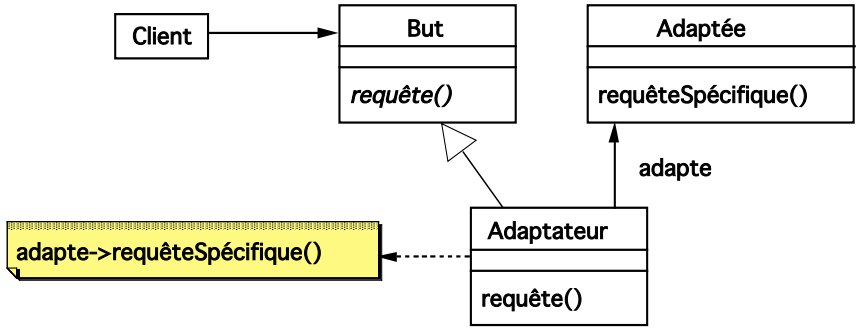
Un adaptateur de classes utilise l'héritage multiple pour adapter une interface à une autre :



Adaptateur

Version objets du modèle Adaptateur

Un adaptateur d'objet repose sur la composition d'objets:



Adaptateur

Indications d'utilisation : Le modèle Adaptateur est utilisé lorsque

- On veut utiliser une classe existante, mais dont l'interface ne coïncide pas avec celle escomptée
- On souhaite créer une classe réutilisable qui collabore avec des classes sans relations avec elle et encore inconnues, c'est-à-dire avec des classes qui n'auront pas nécessairement des interfaces compatibles
- On a besoin d'utiliser plusieurs sous-classes existantes, mais l'adaptation de leur interface par dérivation de chacune d'entre elles est impraticable. Un adaptateur objet peut adapter l'interface de sa classe parente

Adaptateur

Collaborations : Les clients appellent les opérations d'une instance d'Adaptateur. En réponse, l'adaptateur appelle des opérations de l'adapté pour exécuter la requête

Conséquences : Les adaptateurs de classes et d'objets conduisent à des compromis différents

Un adaptateur de classe :

- adapte Adapte à But en s'en remettant à une classe Adaptateur concrète. Un adaptateur de classe ne fonctionne pas si on essaie d'adapter une classe et toutes ses sous-classes
- permet à un adaptateur de redéfinir certain des comportements de l'adapté (Adaptateur sous-classe de Adapte)
- introduit un seul objet, et aucun pointeur additionnel n'est nécessaire pour atteindre Adapte

Adaptateur

Un adaptateur d'objet :

- permet à un simple Adaptateur de travailler avec plusieurs adaptés (Adapte et toutes ses sous-classes). L'Adaptateur peut également ajouter des fonctionnalités à tous les adaptés en une seule fois
- rend plus difficile la surcharge du comportement de Adapte (nécessite la dérivation de cette dernière, et impose que l'Adaptateur fasse référence à la sous-classe plutôt qu'à Adapte)
- Points à prendre en compte lors de l'utilisation du modèle Adaptateur :
  1. Jusqu'à quel point un Adaptateur adapte-t-il ?
  2. Adaptateurs "enfichables". En construisant l'adaptateur d'interface dans une classe, on supprime le préalable que les autres classes voient la même interface.
  3. Utilisation des adaptateurs bidirectionnels (transparence)

Observateur

Objet - Comportemental

Intention : Définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour

Alias : Dépendants, Diffusion - Souscription

Motivation : Un effet couramment induit par le partitionnement d'un système en une collection de classes coopérantes, est l'obligation de maintenir la cohérence des objets en relation. Il n'est pas souhaitable d'obtenir cette cohérence au prix d'un couplage étroit entre les classes, car cela réduirait leur réutilisabilité

Observateur

Les objets de base de ce modèle sont le sujet et l'observateur. Un sujet peut avoir un nombre quelconque d'observateurs sous sa dépendance

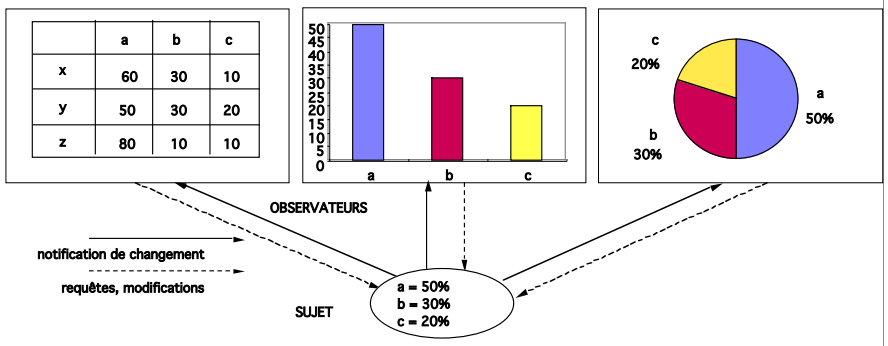
Tous les objet reçoivent une notification chaque fois que le sujet subit une modification de son état

En réponse, chaque observateur interrogera le sujet sur son état afin d'y adapter le sien propre

Ce type d'interaction est également connu sous le nom de Diffusion - Souscription

- Le sujet diffuse les notifications
- Il les expédie, sans avoir à connaître ses observateurs
- Des observateurs, en nombre quelconque, peuvent souscrire pour recevoir les notifications

Observateur

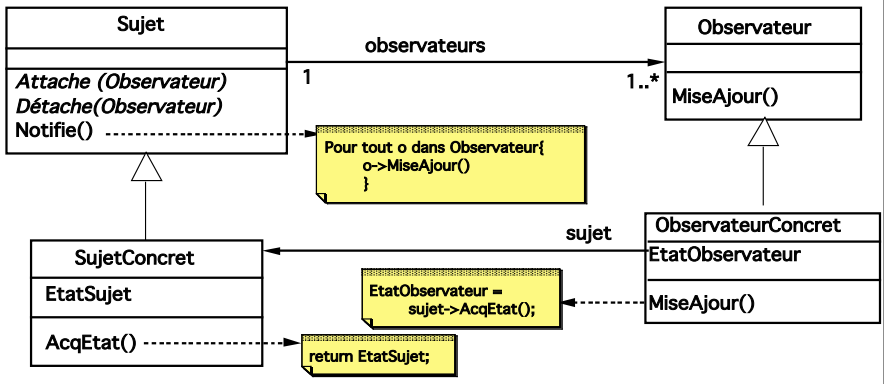


Observateur

Indications d'utilisation :

- Quand un concept a deux représentations, l'une dépendant de l'autre. Encapsuler ces deux représentations dans des objets distincts permet de les réutiliser et de les modifier indépendamment
- Quand la modification d'un objet nécessite de modifier les autres, et que l'on ne sait pas combien sont ces autres
- Quand un objet doit être capable de faire une notification à d'autres objets sans faire d'hypothèses sur la nature de ces objets. En d'autres termes, quand ces objets ne doivent pas être trop fortement couplés

Observateur



Observateur

Le Sujet Concret notifie ses observateurs de tout changement se produisant qui pourrait rendre l'état de ses observateurs incompatible avec le sien propre

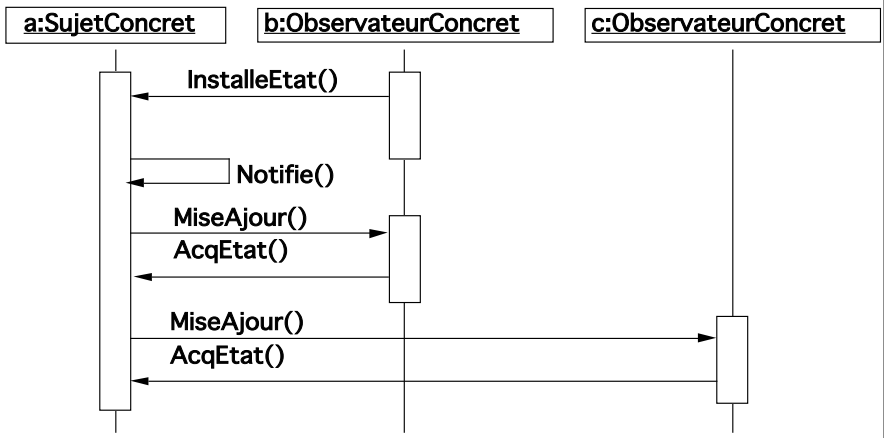
Après avoir été informé d'un changement dans le sujet concret, un objet ObservateurConcret peut faire une demande d'information au sujet

L'ObservateurConcret utilise ces informations pour mettre son état en conformité avec celui du sujet

Le diagramme d'interaction ci-après illustre la collaboration entre un sujet et deux observateurs

L'objet Observateur qui initie la demande de modification, diffère sa mise à jour jusqu'à ce qu'il reçoive la notification en provenance du sujet. La notification n'est pas toujours effectuée par un observateur. Elle peut l'être complètement par le sujet ou par un autre objet

Observateur



Observateur

Conséquences :

Le modèle Observateur permet de modifier les sujets et les observateurs indépendamment. On peut réutiliser les sujets sans réutiliser les observateurs, et réciproquement

On peut ajouter un nouvel observateur sans avoir à modifier les autres et sans avoir à modifier le sujet

Avantages et contingences du modèle Observateur :

1. Isoler le couplage entre Sujet et Observateurs
2. Support de la diffusion
3. Mises à jour inopinées

Procuration

Objet-Structurel

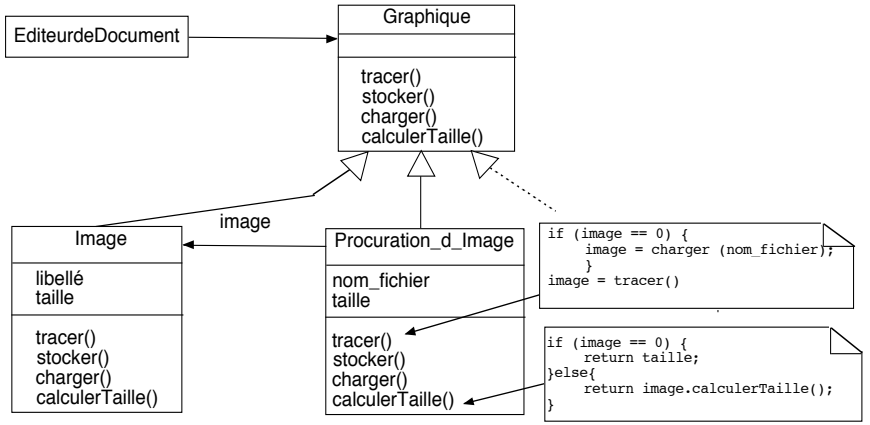
Intention : Fournir à un tiers objet un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

Alias : Subrogé, Proxy.

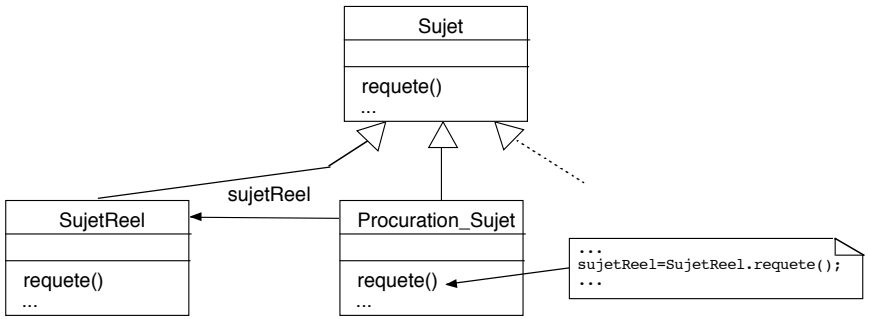
Motivation : Ce qui justifie de contrôler l'accès à un objet, c'est le souci de différer ce qui est coûteux dans sa création et son initialisation, jusqu'au moment de son utilisation effective.

Exemple : Considérons un éditeur de documents capable d'insérer des objets graphiques dans un document. Certains de ces objets tels que les grandes images en mode "raster", peuvent être coûteuses à créer. Comme l'ouverture d'un document doit être rapide, il vaut mieux éviter de créer ces objets coûteux, dés l'ouverture du document

Procuration



Procuration



Procuration

Indications d'utilisation :

L'utilisation de procuration est indiquée quand on a besoin de références à un objet, qui soient plus créatives et plus sophistiquées qu'un simple pointeur

Quelques situations courantes dans lesquelles le modèle de procuration peut être employé :

1. Une procuration à distance fournit un représentant local d'un objet situé dans un espace adresse différent :

   Une **Procuration** = un "**Ambassadeur**"

2. Une procuration virtuelle crée des objets

3. Une procuration de protection contrôle l'accès à l'objet original. Les procurations de protection sont utiles quand les objets doivent satisfaire différents droits d'accès

Procuration

Indications d'utilisation :

4. Une référence intelligente est le remplaçant d'un pointeur brut, qui réalise des opérations supplémentaires, lors de l'accès à l'objet

Exemple :

Décompte du nombre des références faites à un objet réel, de sorte que celui-ci puisse être libéré automatiquement, dés qu'il n'y a plus de références (également appelés "pointeurs intelligents")

Procuration

Collaboration :

Une procuration retransmet les requêtes au SujetReel selon les règles caractéristiques de son type

Conséquences :

Le modèle procuration introduit un certain degré d'indirection dans l'accès à un objet. L'indirection supplémentaire, a de nombreux rôles différents selon le type de procuration :

1. Une procuration distante peut cacher le fait qu'un objet réside dans un autre espace d'adresses
2. Une procuration virtuelle peut effectuer des optimisations telles que la création d'un objet à la demande

Bibliographie

Design Patterns, Elements of reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley Professional computing series, (Français) International Thomson Pub, 1996.

Pattern Languages of Program Design, J. Coplien et D. Schmidt, Addison-Wesley, 1995.

Design Patterns for Object-Oriented Software Development, W. Pree, Addison -Wesley 1998.