CHAPITRE 1 – INTRODUCTION

1. Caractéristiques d'un modèle orienté objet

1) Éléments majeurs
   - Abstraction
   - Encapsulation
   - Héritage
   - Polymorphisme

2) Intervenants
   - Utilisateurs
   - Clients
   - Analystes
   - Concepteurs
   - Programmeurs

3) Points de vue architecturaux
   - Logique : Décomposition orientée objets
   - Processus : Décomposition de l'exécution
   - Implantation : Décomposition statique des modules et sous-systèmes
   - Déploiement : Relation matériel-logiciel
   - Scénario : Comportement du système pour les utilisateurs

CHAPITRE 2 – FONCTIONNALITÉ & CAS D'UTILISATION

1. Cas d'utilisation

• UNE DESCRIPTION RELATIVEMENT LONGUE

• ALLANT DU DÉBUT À LA FIN D'UN PROCESSUS

• COMPORTANT PLUSIEURS ÉTAPES OU TRANSACTIONS.

1) Cas d'utilisation
   - Capturer le comportement désiré du système
   - Spécifier ce que le système fait, mais pas comment il le fait

❖ Document narratif qui décrit la séquence d'évènements dans laquelle un acteur utilise un système pour accomplir un processus

2) Format de haut niveau

Cas d'utilisation	
Acteur	
Type	
Description	

3) Cas d'utilisation étendu

Cas d'utilisation	
Système	
Niveau	
Acteur primaire	
Partie prenante	
Précondition	
Garanties en cas de succès	

4) Scénario en colonne

CLIENT	SERVEUR
1) blablablabla	
2) HOHOHOHO	
	3) HEHEHEHHEHE
	4) HATE THIS COURSE

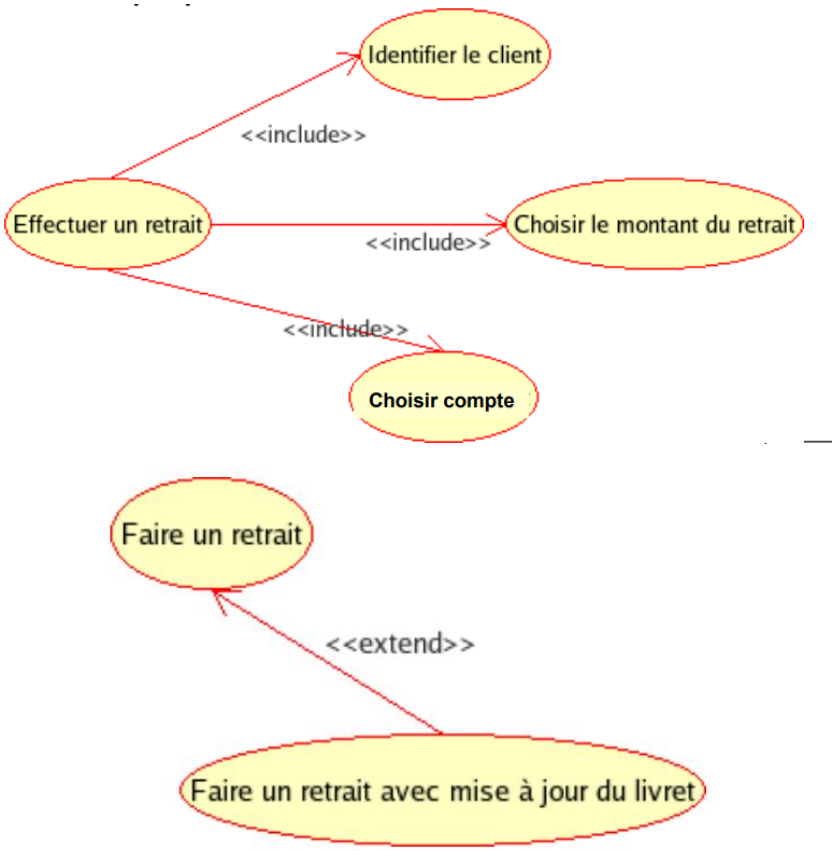
2. Diagramme de contexte

- Permet de visualiser les cas d'utilisations primaires
- Définit les limites du système modélisés, les principaux acteurs du modèle et les cas d'utilisation primaires

1) Relations entre les cas d'utilisations

   i) «include » : Plus commune & plus utile, pour sous-fonctions qui peuvent être reutilisés

   ii) « extends » : Ajouter des étapes à un cas existants sans modifier le cas original



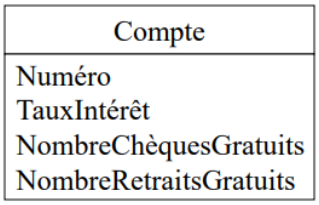
CHAPITRE 3 – CONSTRUNCTION D'UN MODÈLE CONCEPTIONNEL

- ANALYSE ORIENTÉE OBJET
- L'ANALYSE ORIENTÉE OBJET EST LA DÉCOMPOSITION DU PROBLÈME EN CONCEPTS INDIVIDUELS
- LE MODÈLE CONCEPTUEL EST UNE REPRÉSENTATION DES ÉLÉMENTS DU MONDE RÉEL, PAS DE COMPOSANTES LOGICIELLES
  - IL PERMET DE CLARIFIER LA TERMINOLOGIE & LE VOCABULAIRE

1. Modèle conceptuel

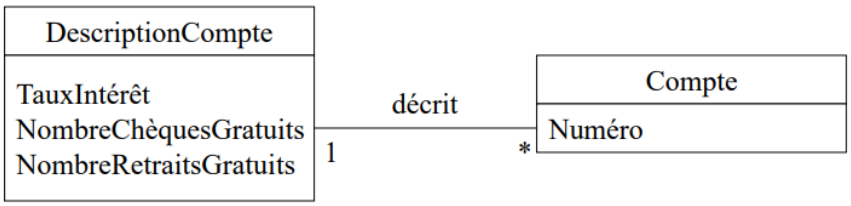
1) Pour construire un modèle conceptuel

   i) Faire la liste des concepts candidats
   ii) Les rassembler dans un diagramme de conceptuel
   iii) Ajouter les associations nécessaires pour conserver les relations qui méritent d'être mémorisée
   iv) Ajouter les attributs nécessaires pour remplir les requis informationnels



Moins bon

Meilleur



2. Concepts, classes & types

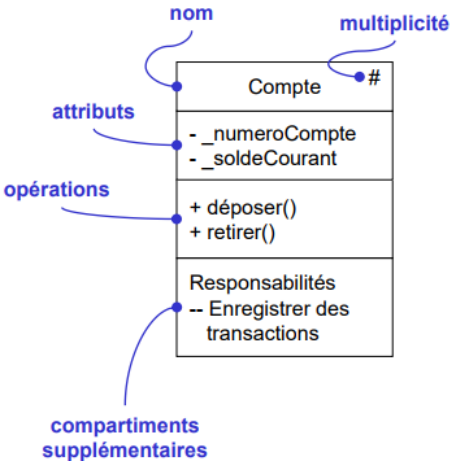
– Un concept: la référence à un élément du monde réel, issu du domaine du problème. Le terme concept n'est pas utilisé ni défini en UML

– Une classe: l'implémentation logicielle d'un concept, avec des attributs et des méthodes. Le terme classe est défini en UML

– Un type: similaire à une classe, mais sans les méthodes. Un type, en UML, est indépendant du langage.

1) Classe

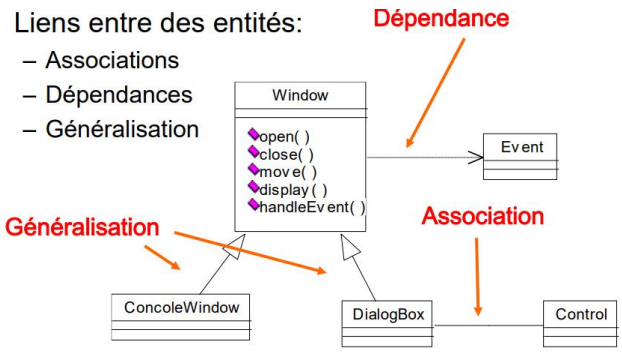
Une classe est la description d'une collection d'objets qui partagent les mêmes attributs, les mêmes opérations, les mêmes relations et la même sémantique.



2) Association

- Relation entre des concepts qui indique une connexion sémantique
- Une association peut être décorée :

- Liens entre des entités:
  - Associations
  - Dépendances
  - Généralisation

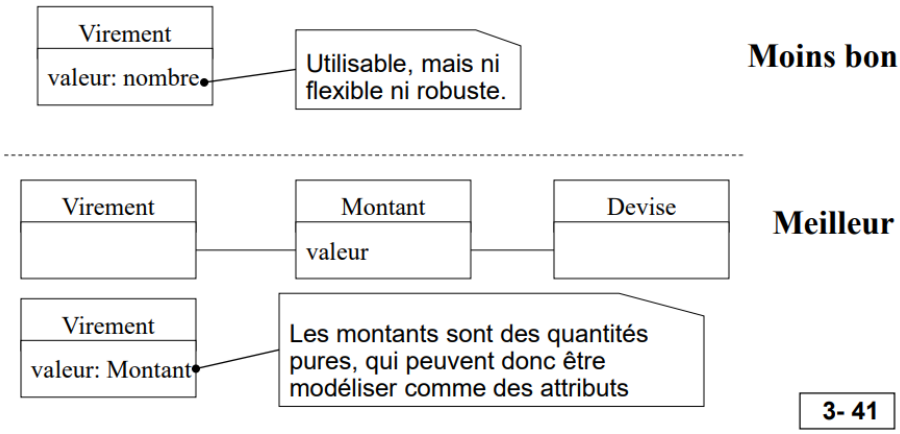


– Nom : décrit la nature de la relation.
– Rôle : nomme la façon dont une classe participe à la relation.
– Multiplicité : indique le nombre d'objets participants.
– Direction : indique le sens de parcours de l'association.

3) Attributs

- TOUTES LES PREMIERES LETTRESSDE MOTS DOIVENT DÉBUTER PAR UNE MAJUSCULE AUF LE PREMIER

❖ Modélisation de quantité et d'unités : Bien que le montant d'une transaction, par exemple, puisse être représenté avec un nombre, il est plus robuste et flexible de le modéliser à l'aide d'un concept séparé pour pouvoir tenir compte des unités:



CHAPITRE 4 – DIAGRAMMES D'INTERACTION

- Interaction : Une description du comportement incluant un ensemble de messages échangés entre un ensemble d'objets à l'intérieur d'un contexte afin d'accomplir un but.

Appeler ——————▶

Invoque une opération d'un objet. Un objet peut s'envoyer un message ce qui correspond à l'invocation locale d'une opération.

Retourner ◀----------

Retourner une valeur à l'appelant.

Envoyer ——————\

Envoyer un signal à un objet.

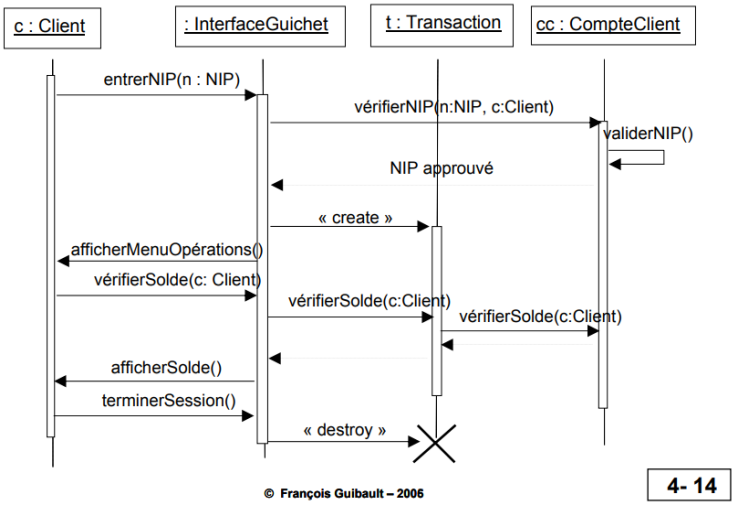
Créer « create » ——————▶

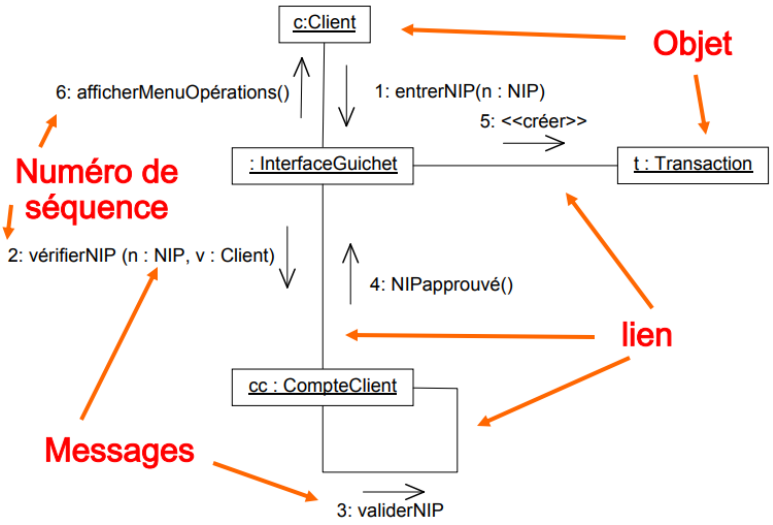
Créer un objet.

Détruire « destroy » ——————▶

Détruire un objet. Un objet peut se suicider en s'envoyant un message de destruction.

1. Diagramme de séquence



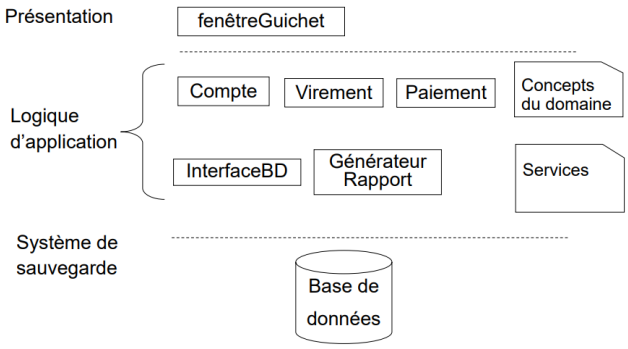

c:Client
Objet
6: afficherMenuOpérations()
1: entrerNIP(n : NIP)
5: <<créer>>
Numéro de séquence
: InterfaceGuichet
t : Transaction
2: vérifierNIP (n : NIP, v : Client)
4: NIPapprouvé()
lien
cc : CompteClient
Messages
3: validerNIP

CHAPITRE 5 – ARCHITECTURE

1. Modèle Architectural

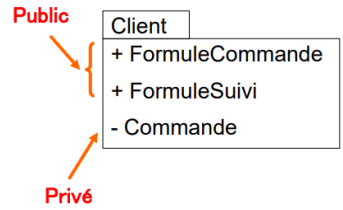
- L'architecture logicielle d'un système est sa décomposition en un certain nombre de sous-systèmes.

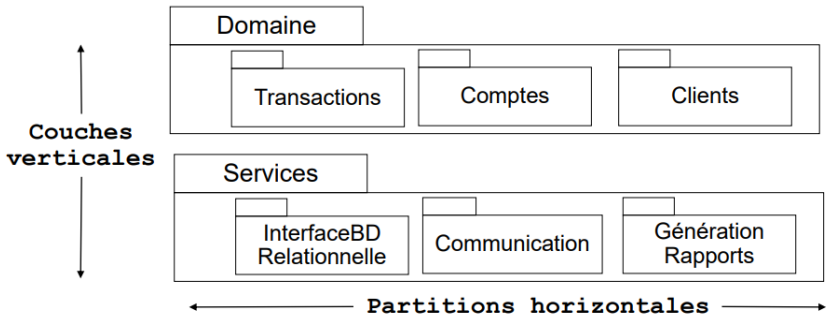
- Architecture multi Niveaux : Présentation, Logique d'application, Système de sauvegarde



2. Diagramme de paquetages

- Utile pour regrouper les éléments communs en paquetages afin de pouvoir les comprendre
- Chaque paquetage doit avoir un nom unique
- Chaque élément d'un paquetage doit avoir un nom unique

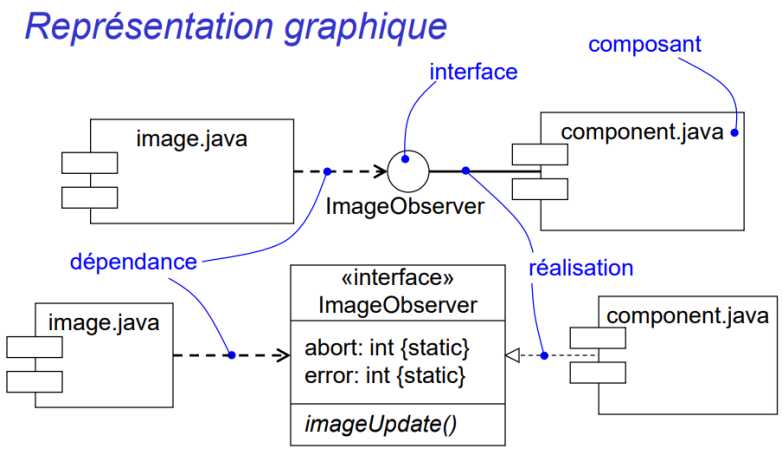




- On regroupe des éléments fournissant un service commun dans un paquetage.

3. Composants & Déploiement

1) Diagramme de composant



- Un diagramme de composants présente une vue statique de l'implémentation d'un système.
- Un composant est une partie physique et remplaçable d'un système qui se conforme et réalise un ensemble d'interfaces
- Un composant est l'implémentation physique de classes ou de paquetages.
- Une interface est une collection de spécifications d'opérations qui définissent le service rendu par une classe ou un composant.

Types de composants

UML distingue trois types de composants :

- *Composant de déploiement*
  Ce sont les composants nécessaires et suffisants pour construire un système exécutable.
- *Composant de réalisation*
  Ce sont les composants résultant du travail de développement (code source, fichier, ...).
- *Composante d'exécution*
  Ce sont les composants qui sont créés lors de l'exécution d'un système.

Stéréotypes standards

UML définit cinq stéréotypes pour les composantes :

- `exécutable`
  spécifie une composante exécutable sur un nœud
- `librairie`
  spécifie un objet de type librairie statique ou dynamique
- `table`
  spécifie une table d'une base de données
- `fichier`
  spécifie un fichier qui contient du code source ou des données
- `document`
  spécifie une composante qui représente un document

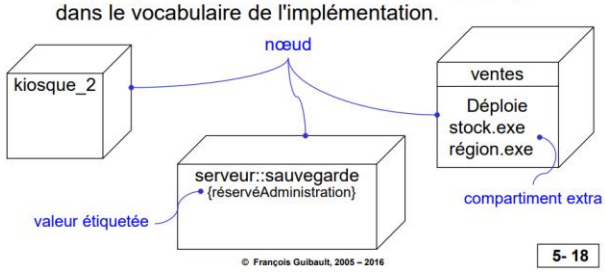
2) Diagramme de déploiement

Diagramme de déploiement

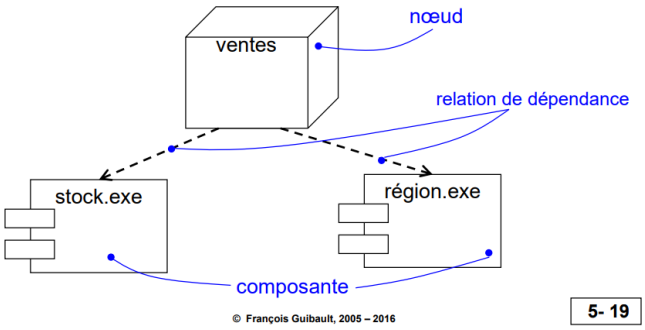
- Un *diagramme de déploiement* présente la configuration physique des ordinateurs et périphériques ainsi que les composants qui s'y exécutent,
- Un *nœud* est un élément physique qui existe au moment de l'exécution et qui représente une ressource ayant des possibilités d'exécution.

Nom et représentation des nœuds

- Chaque nœud doit avoir un nom qui le distingue des autres nœud - Unicité du nom complet (noms des packages englobant + le nom du nœud),
- En pratique les noms de nœuds sont des noms pris dans le vocabulaire de l'implémentation.



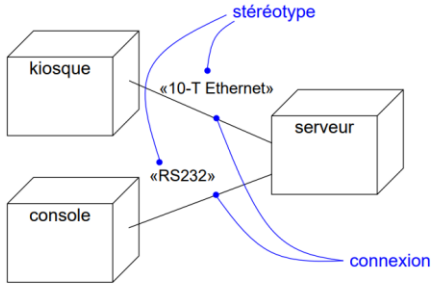
5- 18



5- 19

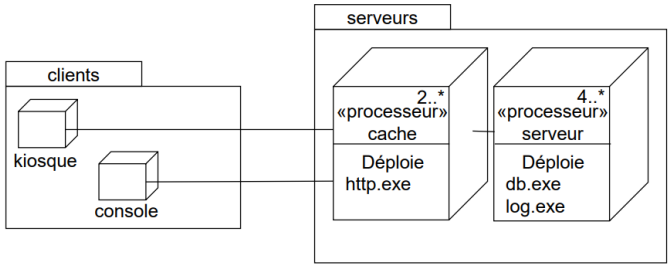
Connexion

- Les associations entre nœuds représentent les connections physiques tels une connexion Ethernet, un câble série, ou encore un bus commun.



5- 20

- Pour modéliser l'implémentation de système client-serveur.



CHAPITRE 6 – PATRONS

		but		
		créationel	**structural**	**comportemental**
portée	**classe**	Factory method	Adapter	Interpreter
				Template Method
	objet	Abstract Factory	Adapter	Chain of Responsability
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

1. Composite et Proxy

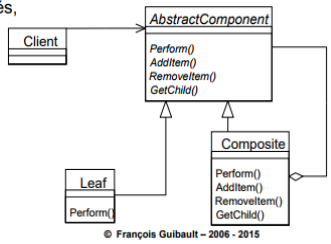
1) Composite

- Intention

  Traiter les objets individuels et les objets multiples, composés récursivement, de façon uniforme.

- Applicabilité
  - les objets doivent être composés récursivement,
  - les objets dans la structure peuvent être traités uniformément,
  - les clients peuvent ignorer les différences entre les objets individuels et composés,

- Structure

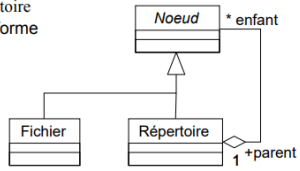


11- 16

- Une structure en arbre suggère un patron `Composite`
  - On cherche un patron structural
  - On veut maximiser l'uniformité et la flexibilité

- Comment appliquer le patron
  - Choisir les participants: `Component`, `Leaf` et `Composite`
  - Choisir les opérations à traiter de façon uniforme

- Établir la correspondance entre les participants au patron `composite` et les classes du système de fichier:
  - Leaf, pour les objets qui n'ont pas d'enfants
    - Fichier, l'objet fichier
  - Composite, pour les objets qui ont des enfants
    - Répertoire, l'objet répertoire
  - Component, l'interface uniforme
    - Noeud



2) Proxy

Patron Proxy

- Intention

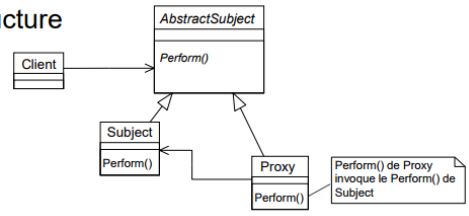
  Fournir un remplaçant ou une doublure pour un autre objet afin de contrôler l'accès à ce dernier.

- Applicabilité

  Ce patron est applicable dès que le besoin d'une référence plus versatile ou plus sophistiquée qu'un simple pointeur se fait sentir.

  L'interface du Proxy doit correspondre à l'interface du sujet.

- Structure



2. Visitor, Template Method and Singleton

1) Patron VIsitor

Patron `Visitor`

- Intention

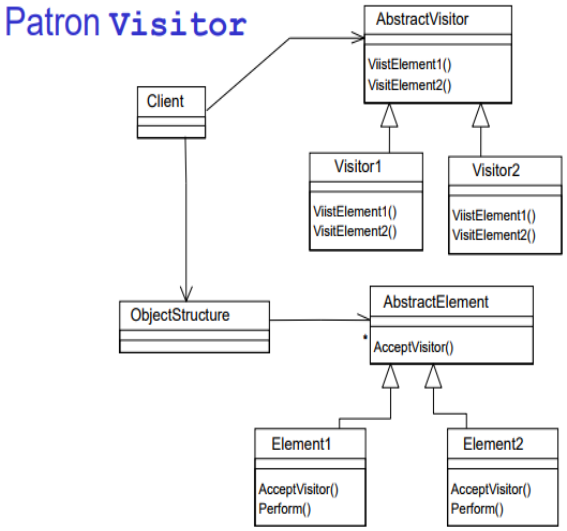
  Représenter une opération qui doit être appliquée sur les éléments d'une structure d'objets. Un Visitor permet de définir une nouvelle opération sans modification aux classes des objets sur lesquels l'opération va agir.

- Applicabilité

  Lorsqu'une structure d'objets contient plusieurs classes avec des interfaces différentes.

  Lorsque plusieurs opérations distinctes et sans liens entre elles doivent agir sur des objets conservés dans une structure.

  Lorsque les classes définissant la structure des objets changent rarement, mais que l'on veut fréquemment définir de nouvelles opérations sur cette structure.



Patron `Visitor`

- Conséquences

  \+ Flexibilité: les Visitors et la structure d'objets sont indépendants

  \+ Fonctionnalité localisée: tout le code associé à une fonctionnalité se retrouve à un seul endroit bien identifié.

  \- Coût de communication supplémentaire entre les Visitors et la structure d'objets.

- Implantation

  – double invocation (double dispatch),

  – interface générale aux éléments de la structure d'objets.

Patron Visitor : évaluation

- Avantages:

  –La dispersion du code est évitée:

  - Tout le code pour faire un `cat` se trouve dans la classe VisiteurCAT

  –Ajouter de nouvelles fonctionnalités ne requiert aucun changement à la hiérarchie de classes du système de fichiers

  - On ajoute plutôt des nouvelles sous-classes de la classe VisiteurNoeud,
  - Les fonctions membres de ces sous-classes **agissent comme des fonctions virtuelles des classes du système de fichiers**.

- Désavantages:

  – Difficile de faire des changements à la hiérarchie des éléments

  - **Toutes** les classes de Visitor doivent être modifiées. Donc tous les clients de l'une ou l'autre de ces classes doivent être recompilés.
  - **Toutes** les implantations des classes Éléments doivent être recompilées,
  - **Tous** les clients des classes Éléments affectés par le changement doivent être recompilés,

  –Tous les types de Visitor doivent avoir un sens pour tous les types d'Éléments:

  - Comment construire un Visitor qui n'agisse que sur les liens ?

  –Le code des Visitor peut être dupliqué

  - Si plusieurs types de nœuds sont traités de la même façon, il faut quand même écrire une fonction pour chaque type.

2) Patron Template Method

Patron **Template Method**

- Intention

  Définir le squelette d'un algorithme dans une opération, et laisser les sous-classes définir certaines étapes.

- Applicabilité

  Pour implanter les aspects invariants d'un algorithme une seule fois et laisser les sous-classes définir les portions variables.

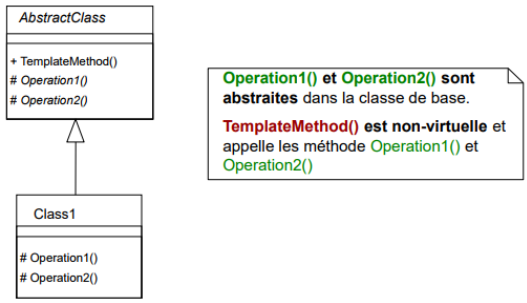
  Pour situer les comportements communs dans une classe afin d'augmenter la réutilisation de code.

  Pour contrôler les extensions des sous-classes.

- Conséquences
  + mène à une inversion de contrôle («Principe Hollywood: ne nous appelez pas, nous vous appellerons »)
  + favorise la réutilisation de code
  + permet d'imposer des règles de surcharge
  - il faut sous-classer pour spécialiser le comportement.
- Implantation
  - méthode Template virtuelle ou non-virtuelle
  - peu ou beaucoup d'opérations primitives
  - convention de noms (préfix do- ou faire-)

Patron **Template Method**

- Structure



3) Patron Singleton

Patron **Singleton**

- Intention

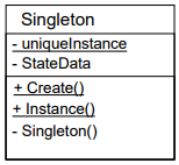
  S'assurer qu'il ne soit possible de créer qu'une seule instance d'une classe, et fournir un point d'accès global à cette instance.

- Applicabilité

  Lorsqu'il doit y avoir exactement une seule instance d'une classe, et que cette instance soit accessible par un mécanisme bien identifié.

  Lorsque la seule instance d'une classe doit être généralisable en sous-classant et que les clients doivent être en mesure d'utiliser l'instance généralisée sans modification à leur code.

- Structure



Patron **Singleton**

- Conséquences
  - \+ Réduit la pollution du *namespace* global
  - \+ Permet de contrôler l'instantiation d'une classe (on peut limiter le nombre d'instances à 1, 2, n.)
  - \+ Permet la généralisation par sous-classification (comparativement à une classe équivalente dont toutes les fonctions seraient statiques)
  - \- Implantation peut être légèrement moins efficace qu'une variable globale
- Implantation
  - Opération getInstance() statique
  - Enregistrement de l'instance du Singleton

Patron **Singleton**

- Extrait des conséquences du patron Singleton

  « [Singleton] permet un nombre variable d'instances. Ce patron permet facilement de changer d'idée et de permettre plus d'une instance de la classe Singleton. De plus, on peut utiliser la même approche pour contrôler le nombre d'instances qu'une application utilise. Seule la méthode qui donne accès à l'instance du Singleton (getInstance) doit être modifiée pour changer les règles d'accès. »

- Dans notre cas, on veut **une et une seule instance d'Usager, par usager**.

3. Mediator, Observer, Abstract Factory et Prototype

1) Mediator

Patron **Mediator**

- Intention

  Définir un objet qui encapsule comment un ensemble d'objets interagissent afin de promouvoir un couplage faible et de laisser varier l'interaction entre les objets de façon indépendante.

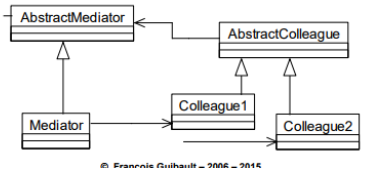
- Applicabilité

  Il existe un comportement de coopération entre des objets, qui ne peut être assigné à un objet individuel,

  Un ensemble d'objets communiquent entre eux de façon bien définie mais complexe,

  L'ordre des opérations peut changer à mesure que le système évolue.

- Structure

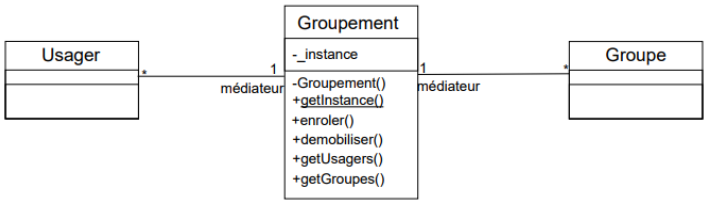


13- 10

Patron **Mediator**

- Conséquences
  - \+ Encapsule les communications
  - \+ Simplifie le protocole entre les objets
  - \+ Évite de forcer un ou plusieurs collègues à assumer les responsabilités de médiation
  - \- Le Mediator peut devenir complexe et monolithique
- Implantation
  - Utilisation de membres statiques plutôt que de classes séparées.
  - Omission de la classe abstraite **`AbstractMediator`**.
  - Nécessité de la classe abstraite **`AbstractColleague`** ?
  - Le Mediator en Singleton.

13- 11



2) Observer

Patron Observer

- Intention

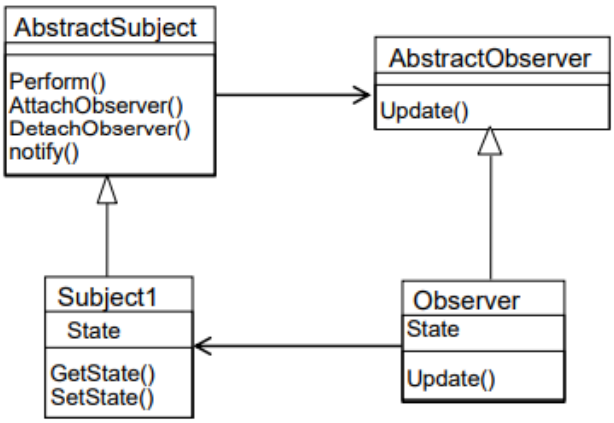
  Définit une relation un à plusieurs entre des objets de façon à ce que lorsqu'un objet change d'état, que tous ses dépendants soient avertis et mis à jour automatiquement.

- Applicabilité

  Lorsqu'une abstraction a deux aspects, l'une dépendant de l'autre.

  Lorsque le fait de changer un objet implique d'en changer d'autres, sans que l'on sache combien d'autres objets doivent être changés.

  Lorsqu'un objet devrait en avertir d'autres sans faire de supposition sur la nature de ces autres objets.



Patron Observer

- **Conséquences**
  - \+ Modularité: Le sujet et les observateurs peuvent varier de façon indépendante.
  - \+ Extensibilité: On peut définir et ajouter autant d'observateurs que nécessaire.
  - \+ Adaptabilité: Différents observateurs fournissent différentes vues du Subject
  - \- Mises à jour inattendues: les Observers ne se connaissent pas
  - \- Coût de la mise à jour: certains Observers peuvent avoir besoin d'indices sur ce qui a changé
- **Implantation**
  - Correspondance Sujet-Observateur… il y a plusieurs façon de l'implanter.
  - Références pendantes.
  - Éviter des protocoles de mise à jour spécifiques à un Observer (modèle « tirer » vs. « pousser »).
  - Enregistrement explicite des modifications d'intérêt.

13- 18

Patron Observer

- Qui invoque la méthode avertir() ?
  - Le **sujet** peut l'invoquer lui-même lorsqu'il est modifié.
    - Rend le processus d'avertissement **automatique**.
    - Par contre, c'est **inefficace** dans les cas où plusieurs **modifications consécutives** sont faites sur le sujet.
  - Le **client** peut être chargé de déclencher lui-même explicitement le processus d'avertissement après avoir fait ses modifications sur un sujet donné.
    - **Efficace pour les modifications consécutives**.
    - Par contre, cela force les clients à démarrer le processus d'avertissement eux-mêmes (**on perd l'automatisme**).

Patron Observer : Résumé

- Le patron **Observer** permet à des sujets d'avertir des entités (observateurs) qu'une autre entité (sujet) a changé, sans que les observateurs n'aient à connaître les sujets en détail.
- À sa plus simple expression, le patron **Observer** permet aux sujets d'enregistrer des observateurs qui seront avertis d'éventuelles modifications au sujet.
- On peut également utiliser un gestionnaire de changements qui permet de centraliser la gestion des associations sujets-observateurs. Ce gestionnaire de changements est en fait un **Médiateur** qui peut être implanté sous forme de **Singleton**.
- Plusieurs bibliothèques et systèmes de classes sont déjà disponibles qui fournissent des mécanismes d'enregistrement et de rappel, dont la bibliothèque `signals2` de Boost.
- Un mécanisme de signaux et de réception de signaux peut être ainsi mis en place pour traiter différents évènements non connus à l'avance.

3) Abstract Factory

Patron `Abstract Factory`

Intention

Procure une interface permettant de créer une famille d'objets connexes ou dépendants sans spécifier leurs classes concrètes.

Applicabilité

Un système devrait être indépendant de la façon dont ses produits sont créés, composés et représentés.

Un système devrait être configuré avec une des multiples familles de produits.

Une famille d'objets de produits connexes est conçue pour que ceux-ci soient utilisés ensemble. On a besoin de renforcer cette contrainte.

On veut fournir une librairie de classes de produits et on veut seulement révéler leurs interfaces et non leur implémentation.

Conséquences

+ Isole les classes concrètes.

  Les clients manipulent les instances à partir des interfaces abstraites. Les noms des classes de produits sont isolés dans l'implantation de l'usine concrète, ils n'apparaissent pas dans le code du client.

+ Facilite les changements de familles de produits.

  Les classes d'une seul usine concrète à la fois sont utilisées. On peut changer facilement d'usine concrète.

+ Encourage la consistance entre les produits.

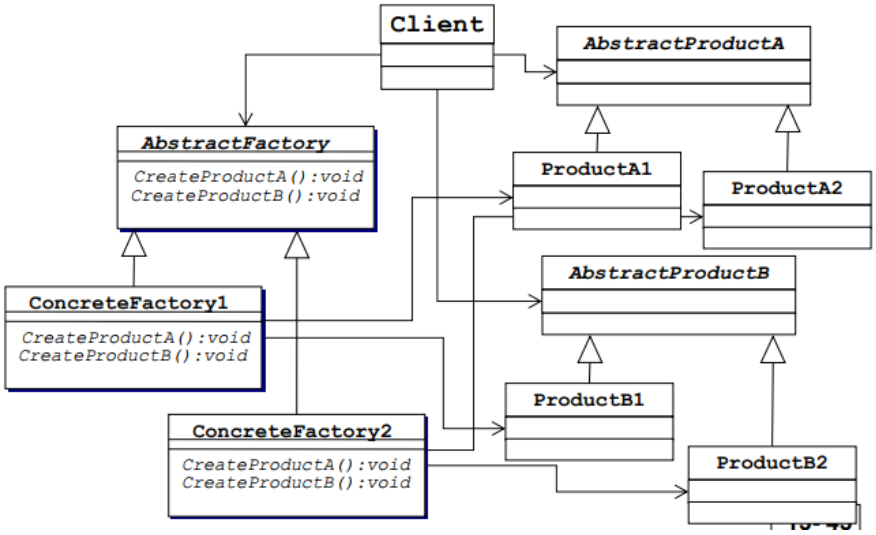
  Regroupe les classes qui doivent fonctionner ensemble.

- Le support de nouveaux types de produits est difficile.

  L'interface de l'usine abstraite détermine l'ensemble de produits pouvant être créés. On doit modifier l'interface et toutes ses sous-classes pour supporter de nouveaux produits.

Implantation

- Les usines peuvent être implantées en Singletons
- Pour la création du produit, on utilise une «Factory Method».
  - Il s'agit d'une méthode abstraite dans la classe de base qui retourne un pointeur à la classe de base du produit. C'est à chaque sous-classe de l'usine abstraite de surcharger la méthode et créer l'instance du produit concret.
- Définir des **usines extensibles**
  - On peut obtenir des usines extensibles en passant en paramètre à la méthode de création un identificateur indiquant le type d'objet à créer.
  - Fonctionne uniquement si tous les objets à créer dérive d'une classe de base identique.
  - On a alors une seule fonction de création.
  - Nécessite des « cast » pour naviguer dans la hiérarchie. C'est donc peu sécuritaire.
  - Mal adapté à un langage statiquement typé comme C++.

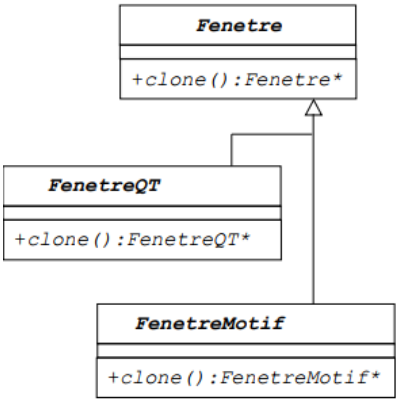


4) Prototype

Le patron **Prototype**

- Les produits possèdent une méthode pour se cloner.
- Le client construit un objet prototype du bon type.
- Puis, pour instancier à nouveau un objet, il utilise la méthode de clonage sur l'objet prototype.
- Élimine le besoin d'avoir recours à des usines concrètes pour chaque famille de produits.

Le patron **Prototype**

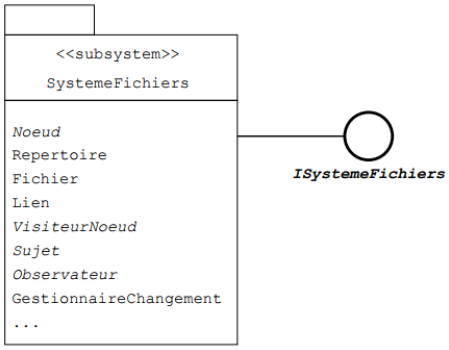



```
Fenetre* fenetrePrototype =
                    new FenetreQT;

void main( void )
{
   [ ... ]
   Fenetre* nvlleFenetre =
               prototype->clone();

   [ ... ]

   Fenetre* autreFenetre =
               prototype->clone();
```

4. Façade et Chain of Responsability

1) Façade

– On voudrait rendre facile l'accès aux fonctionnalités les plus usuelles.
– On voudrait présenter une interface claire et simple au système de fichiers.
– La plupart des clients ne sont pas intéressés à connaître tous les détails de la structure interne du système de fichiers. Ils veulent juste l'utiliser !



Patron **Façade**

Intention

Fournir une interface unifiée à un groupe d'interfaces d'un sous-système. Une façade définit une interface de haut niveau rendant l'utilisation d'un sous-système plus faciles.

Applicabilité

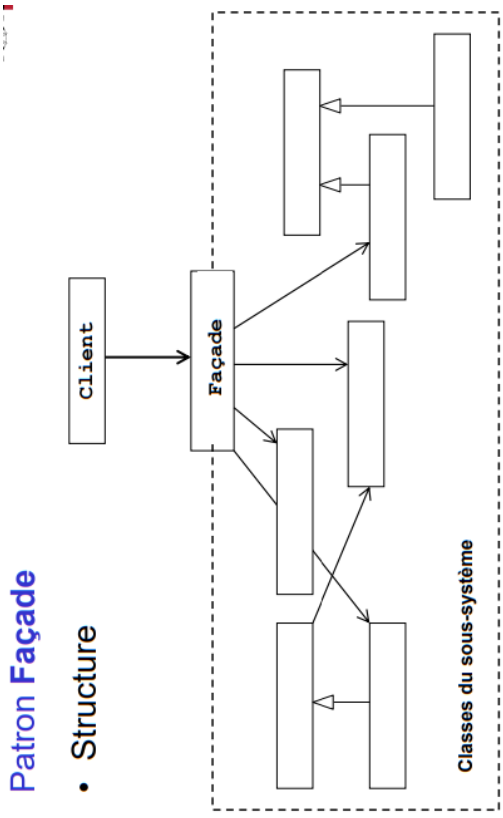
Lorsqu'on veut **fournir une interface simple à un sous-système complexe** (en particulier, l'utilisation des patrons de conception résulte fréquemment en un grand nombre de petites classes rendant le sous-système plus réutilisable, plus configurable, mais parfois plus difficile à utiliser).

On veut **éviter un trop fort couplage** entre les **clients** et les **classes implantant une abstraction**.

On veut implanter une **architecture du système en plusieurs couches**. La classe Façade fournit un point d'entrée à chaque sous-système.

Patron **Façade**

- Implantation
  - Réduire le couplage entre les clients et le sous-système peut amener à faire de la classe façade une classe abstraite. Différentes sous-classes de la façade peuvent alors implanter différentes versions du sous-système.
  - Classes publiques et privées. De même que pour les membres d'une classe, on peut parler de classes publiques et privées dans un sous-système.
    - Dans un langage comme C++, ceci peut être difficile à implanter.
    - Dans un langage comme Java, la notion de classes privées et publiques dans un paquetage existe.



Patron **Façade**

- Structure

2) Chain of Responsability

Intention

Éviter de coupler l'émetteur et le récepteur d'une requête en donnant la possibilité à plus d'un objets de traiter la requête. Les objets récepteurs sont chaînés et la requête traverse la chaîne jusqu'à ce qu'elle soit traitée.

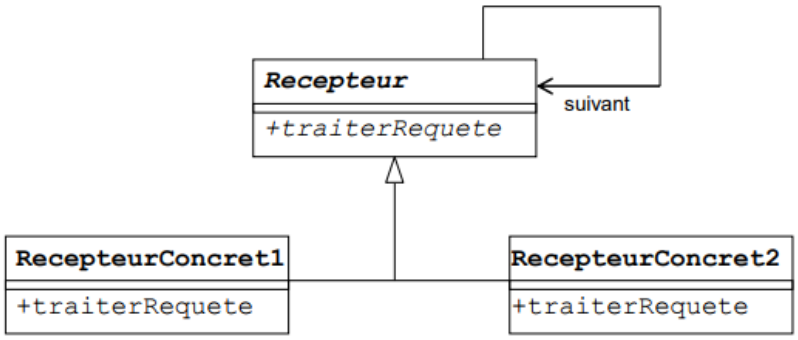
Applicabilité

Plus d'un objets peuvent traiter une requête et que l'objet qui traitera la requête n'est pas connu à priori.

On veut envoyer une requête à plus d'un objets sans spécifier explicitement lequel la traitera.

L'ensemble d'objets qui peut traiter la requête doit être spécifié dynamiquement.

- Structure



Conséquences

+ **Réduit le couplage**. Ce patron évite à l'émetteur de devoir connaître l'objet récepteur. L'émetteur ne connaît que le point d'entrée de la chaîne.
+ **Augmente la flexibilité** avec laquelle les responsabilités peuvent être assignées aux objets. On peut modifier la chaîne en cours d'exécution et changer dynamiquement la façon dont les requêtes seront traitées.
- La réception et **le traitement de la requête n'est pas garantie**. Comme il n'y a pas de récipiendaire explicite, rien ne garantit à l'émetteur que sa requête sera bel et bien traité.

5. Iterator, Strategy, State et Command

Patron Iterator

Intention

Fournir une méthode d'accès séquentielle aux éléments d'un objet agrégat (liste, vecteur, …) sans exposer sa structure interne.

Applicabilité

Pour accéder au contenu d'un objet agrégat sans révéler sa structure interne.

Pour supporter les traversées multiples et simultanées d'un objet agrégat.

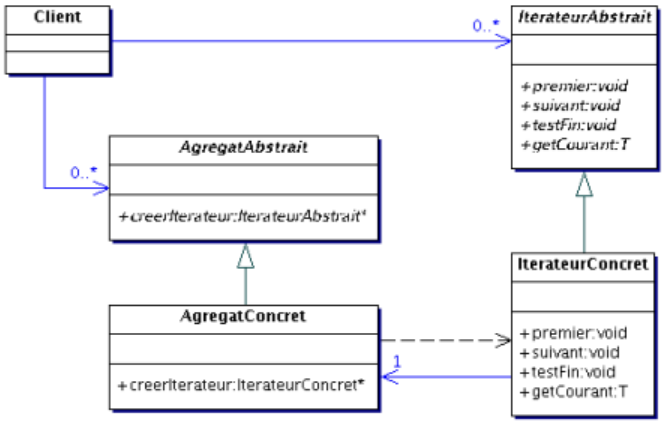
Pour fournir une interface uniforme de traversée pour différents types de structures agrégat.

Motivation

On veut éviter de polluer les objets agrégats avec des fonctions de parcours des éléments. On voudrait éviter d'avoir ceci:

Patron Iterator

Structure



Conséquences

+ Variabilité : L'itérateur permet de supporter plusieurs variations dans le mode de traversée de l'agrégat.
+ Simplification : En définissant un itérateur, on peut simplifier l'interface de l'objet agrégat.
+ Traversées multiples : On peut faire **plusieurs traversées simultanées** sur un même objet agrégat.
+ Réutilisation : On peut écrire des algorithmes génériques valables pour tous les types d'agrégats en passant par les itérateurs (STL).

Patron Strategy

Intention

Encapsuler un algorithme dans une classe de façon à le rendre **interchangeable**. Le patron Strategy permet de faire varier l'algorithme indépendamment du client qui l'utilise.

Applicabilité

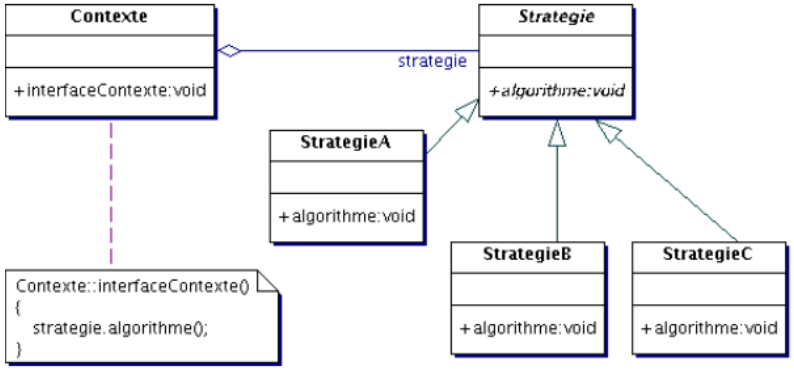
Plusieurs classes ne diffèrent que par leur comportement. On peut alors configurer une classe avec plusieurs comportements indépendants sans multiplier le nombre de sous-classes.

Plusieurs variantes d'un algorithme sont nécessaires.

Un algorithme requiert et stocke plusieurs données. La stratégie permet d'encapsuler ces données.

Une classe a plusieurs comportements qui impliquent de multiples énoncés conditionnels.

Patron Strategy



Patron Strategy

Conséquences

+ Support de familles d'algorithmes reliés.
+ Alternative au sous-classement. Permet d'obtenir des combinaisons d'algorithmes en limitant le sous-classement.
+ Élimination d'énoncés conditionnels.
+ Choix d'implantations libre.
- Les clients doivent être conscients de l'existence des stratégies et comprendre les nuances entre chacune afin de choisir la bonne. Les clients sont donc exposés à l'implantation.
- Coût de communication supplémentaire pour l'appel à la fonction virtuelle de la stratégie.
- Augmente le nombre d'objets dans le système. Certaines stratégies peuvent être implantées dans des objets partagés pour réduire ce problème.

Patron State

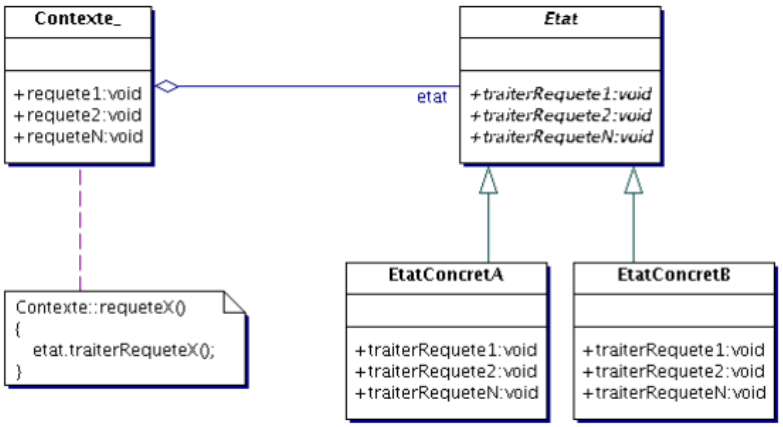
Intention

Permettre à un objet de changer son comportement en fonction de son état. L'objet se comportera comme s'il avait changé de classe.

Applicabilité

Le comportement d'un objet dépend de son état et cet état varie en cours d'exécution.

Les opérations contiennent de multiples énoncés conditionnels de type *switch-case* (souvent basés sur des type énumérés) qui permettent d'avoir un comportement différent selon l'état de l'objet. La patron State permettra d'éliminer ce genre d'énoncés conditionnels et de le remplacer par une solution orientée objet.



Conséquences

+ Localisation des comportements propres aux états. Toutes les données et les comportements propres aux états se situent dans les sous-classes. Il est alors facile d'ajouter de nouveaux états.
+ Rend les transitions entre les états explicites.
+ Permet d'éviter les états inconsistants (le changement d'état est atomique).
+ Élimine les grands énoncés conditionnels.
± Les comportements reliés aux états sont distribués dans des sous-classes, ce qui augmente le nombres de classes et est moins compact. Toutefois, cette avenue est très utile s'il y a beaucoup d'états différents et est largement préférable à l'utilisation de multiples énoncés conditionnels.

Patrons State vs Strategy : quelle est la différence ?

Comparaison entre les deux patrons

- **La structure** des deux patrons est similaire.
  - Dans les deux cas, on délègue la responsabilité de l'implantation à une hiérarchie de classes distinctes (les stratégies et les états).
  - Dans le cas des états cependant, **plusieurs fonctions varient** en fonction de l'état, alors que les stratégies encapsulent chacune **un algorithme**.
- **L'intention** derrière les deux patrons n'est pas la même.
  - Dans les stratégies, l'intention est d'encapsuler un algorithme et de possiblement combiner plusieurs stratégies pour utiliser simultanément et indépendamment différentes versions de plusieurs algorithmes.
  - Dans le cas des états, le comportement de **tout l'objet** doit être modifié lorsque l'état change. Le patron State permet de spécialiser le comportement en fonction de l'état de l'objet et les transitions d'états peuvent être déterminées par les états eux-mêmes.

Patron Command

Intention

Encapsuler une requête dans un objet de façon à permettre de supporter facilement plusieurs types de requêtes, de définir des queues de requêtes et de permettre des opérations « annuler ».

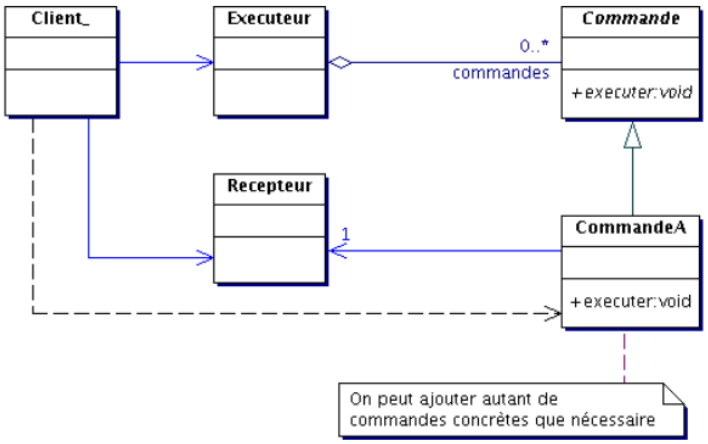
Applicabilité

Paramétrer des objets avec une action à effectuer. Command est une alternative orientée objet aux fonctions de *callback*.

Spécifier une queue de requêtes qui seront exécutées ultérieurement.

Support naturel pour les *undo*/*redo*.

Patron Command



Conséquences

+ Découple l'objet qui invoque la requête de celui qui sait comment la satisfaire.
+ Les commandes sont encapsulées dans des objets qui peuvent être manipulées comme tout objet. L'utilisation d'objets amène également plus de flexibilité.
+ On peut facilement créer de nouvelles commandes.
+ Les commandes peuvent être assemblées en des commandes composites si nécessaires. Le patron Command peut, en effet, être combiné au patron Composite pour représenter des commandes qui sont un ensemble d'autres commandes.

CHAPTER 7 – EXCEPTIONS

Le traitement des exceptions

MONTR

- Une exception permet de transmettre de l'information entre l'endroit où se produit une erreur et l'endroit où elle sera traitée.
- Lors du lancement d'une exception, le cours normal d'exécution est suspendu et la pile des appels est remontée jusqu'à l'endroit où cette erreur peut être traitée.
- Le code intermédiaire entre le lancement d'une exception et son traitement doit être conçu de façon à laisser passer l'exception sans corruption.

Exception:
Interruption de séquence permettant de traiter une erreur.

try:
Énoncé permettant de rassembler un ensemble d'instructions susceptibles de lancer une exception.

throw:
Instruction permettant de lancer une exception. Cette instruction doit être comprise dans un bloc *try*. Un objet est créé pour encapsuler l'exception et il sera transmis à l'endroit traitant cette exception.

catch:
Identifie un bloc d'instructions pouvant recevoir et traiter une exception. C'est à cet endroit que l'exception lancée pourra être traitée.

std::bad_alloc
Lancé par new si l'allocation de mémoire échoue.

std::bad_cast
Lancé par dynamic_cast si la conversion de type échoue.

std::bad_exception
Lancé par unexpected() en cas d'exception inattendue.

std::bad_typeid
Lancé par une expression de type typeid si le pointeur est NULL.

std::ios_base::failure
Classe de base pour les exceptions lancées dans iostream.

std::logic_error
Représente des problèmes dans la logique interne d'un programme qui peuvent normalement être prévenus en écrivant correctement le code de l'application.

Sous-classes de logic_error:
- std::invalid_argument. Indique que les arguments passés à la fonction sont incorrects ou invalides
- std::length_error. Accès à hors de la plage de mémoire allouée. Typiquement, débordement de tableau.
- std::domain_error. Représente des erreurs de domaine au sens mathématique du terme.
- std::out_of_range. Représente un argument dont la valeur n'est pas dans la plage attendue.

std::runtime_error
Représente des erreurs qui ne peuvent être détectées qu'à l'exécution du programme.

Sous-classes de runtime_error
- std::overflow_error. Erreur arithmétique interne de débordement.
- std::underflow_error. Erreur arithmétique interne de dépassement de la capacité inférieure.
- std::range_error. Erreur de plage invalide dans les calculs internes.

États des objets en cas d'exceptions

État initial

L'objet est laissé dans l'état où il était avant l'appel à la fonction.

État correct

L'état de l'objet est cohérent. Tous les attributs sont cohérents entre eux et l'objet est utilisable normalement bien qu'il ne soit plus dans l'état initial.

État incorrect

Certains attributs ne sont plus cohérents entre eux. L'objet n'est **pas utilisable. Par contre, son destructeur est encore utilisable.**

État indéfini

Le destructeur de l'objet n'est plus utilisable. Cela se produit, par **exemple, si un pointeur a été désalloué mais est resté dans un état** indéfini (n'a pas été réinitialisé). Le détruire une deuxième fois **(dans le destructeur) produit un comportement indéfini (mais probablement désagréable).**

1. On doit essayer de laisser l'objet dans l'état initial (l'état où il était avant l'appel à la fonction).
2. Si c'est impossible, il faut essayer de laisser l'objet dans un état **correct (utilisable).**
3. Si c'est impossible, il faut le laisser dans un état défini (on peut au **moins le détruire ou le réinitialiser).**
4. **Il faut également éviter les fuites de ressources.**
5. Il ne faut pas attraper d'exceptions inutilement.
6. Il ne faut pas cacher de l'information contenue dans une exception à d'autres parties du programme.
7. **Il ne faut pas compter sur la capacité à traiter les exceptions des destructeurs.**
8. **Il ne faut pas devenir trop paranoïaque !**