IN413 (Design Patterns en java) : Contrôle final

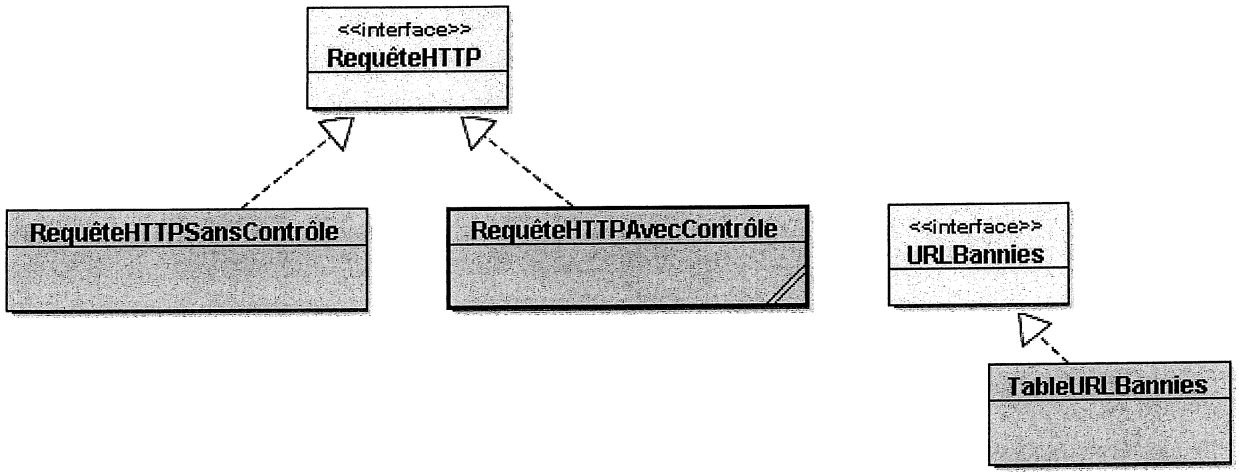
ESIEE Engineering, Jean-Michel DOUIN & Denis BUREAU, 4 novembre 2009

2 heures, SANS DOCUMENTS, SANS CALCULATRICE.

Question 1 (/ 7 pts) : Un Filtrage des URL : le patron Proxy

Au sein d'une institution, certaines URL ont été bannies ; un mandataire se charge de contrôler les accès.

L'architecture ci-dessous reflète l'usage du patron Proxy, ici appliqué au contrôle des requêtes :



L'interface associée aux requêtes :

```
2
3 public interface RequêteHTTP{
4
5   public String effectuer(String url);
6
7 }
8
```

La table des URL bannies implémente cette interface :

```
2 public interface URLBannies{
3
4   public void ajouter(String url);
5
6   public boolean estBannie(String url);
7
8
9 }
```

Q1-1) Proposez une implémentation complète de la classe *RequêteHTTPAvecContrôle* .

Q1-2) Proposez une implémentation d'une table des URLs à bannir : la classe *TableURLBannies* ; vous pourrez utiliser la classe *java.util.TreeSet<E>* , laquelle implémente l'interface *java.util.Set<E>* qui est en annexe.

Question 2 (/ 6 pts) : Une notification HTTP à chaque ajout : le patron Observateur

A chaque ajout d'une URL indésirable dans la table *TableURLBannies*, une notification doit être envoyée à tous les observateurs inscrits.

Q2-1) Modifiez l' implémentation de cette classe afin d'engendrer cette notification ; vous utiliserez la classe *java.util.Observable* qui est en annexe.

Q2-2) Proposez un observateur, lequel implémente *java.util.Observer,* qui à chaque notification émet une requête HTTP à cette url : http://www.esiee.fr/in413/examen/ ; vous utiliserez l'interface *RequêteHTTP* de la question 1 afin d'effectuer cette requête.

Question 3 (/ 3 pts) :

Q3-1) Rappelez la fonctionnalité attendue du Patron *Itérateur* ?

Q3-2) Proposez une méthode permettant de retirer une URL de la table des URLs bannies en utilisant un itérateur.

Question 4 (/ 3 pts) :

Q4-1) Quelles sont les différences entre le patron *Itérateur* et le patron *Visiteur* ?

Q4-2) Quels sont les principaux avantages escomptés avec l'usage du patron *Visiteur* pour la gestion de la table des URLs bannies ?

Qualité globale de la programmation (/ 2 pts) :

- Très bonne → 2 points (notation sur 21)
- Normale → 1 point (notation sur 20)
- Mauvaise → 0 point (notation sur 19)

Annexes

java.util. Interface Set<E>

Type Parameters: E - the type of elements maintained by this set
All Superinterfaces: Collection<E>, Iterable<E>
All Known Implementing Classes: ..., , HashSet, TreeSet, ...

public interface **Set<E>** extends Collection<E>

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

java.util. Interface Observer

public interface **Observer**

A class can implement the Observer interface when it wants to be informed of changes in observable objects.

Method Summary	
void	**update**(Observable o, Object arg) This method is called whenever the observed object is changed.

java.util. Class Observable

java.lang.Object
    java.util.Observable

Constructor Summary	
Observable()	Construct an Observable with zero Observers.

Method Summary	
void	**addObserver**(Observer o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	**clearChanged**() Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false.
int	**countObservers**() Returns the number of observers of this Observable object.
void	**deleteObserver**(Observer o) Deletes an observer from the set of observers of this object.
void	**deleteObservers**() Clears the observer list so that this object no longer has any observers.
boolean	**hasChanged**() Tests if this object has changed.
void	**notifyObservers**() If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
void	**notifyObservers**(Object arg) If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
protected void	**setChanged**() Marks this Observable object as having been changed; the hasChanged method will now return true.

java.util. Interface Iterator<E>

public interface **Iterator<E>**

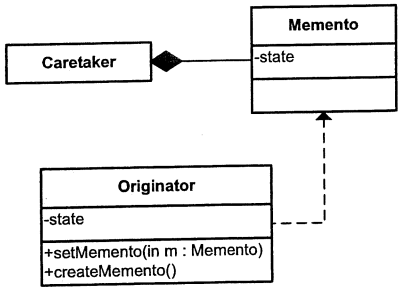
Method Summary	
boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

- C Abstract Factory
- S Adapter
- S Bridge
- C Builder
- B Chain of Responsibility
- B Command
- S Composite
- S Decorator
- S Facade
- C Factory Method
- S Flyweight
- B Interpreter
- B Iterator
- B Mediator
- B Memento
- C Prototype
- S Proxy
- B Observer
- C Singleton
- B State
- B Strategy
- B Template Method
- B Visitor

Memento

Type: Behavioral

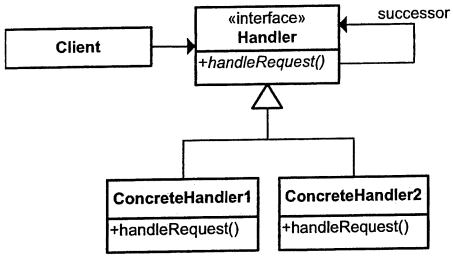
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

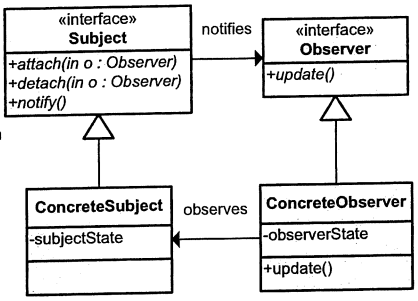
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

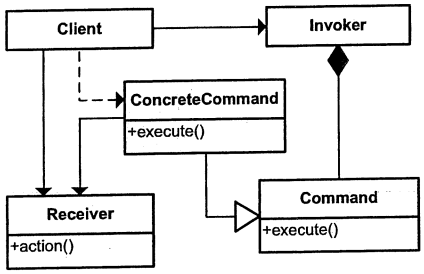
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

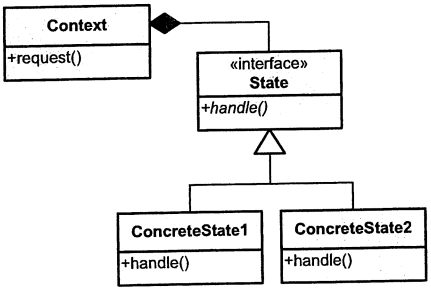
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

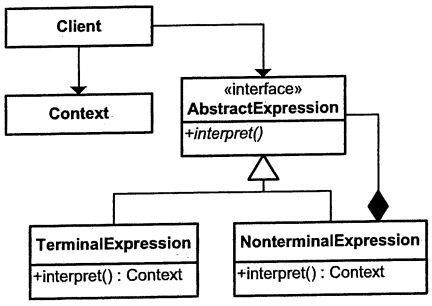
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

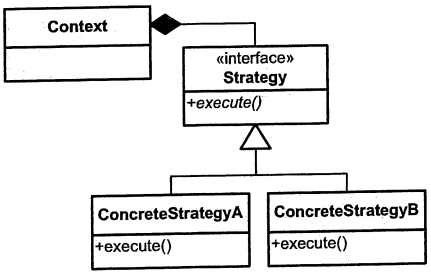
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

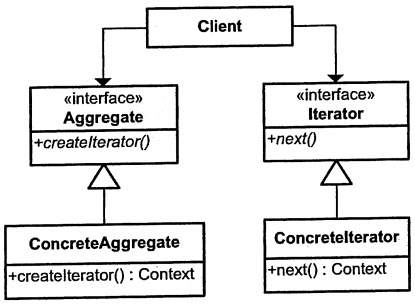
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

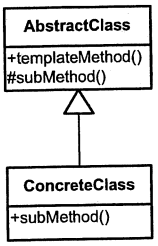
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Template Method

Type: Behavioral

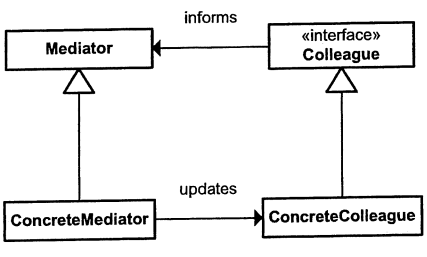
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Mediator

Type: Behavioral

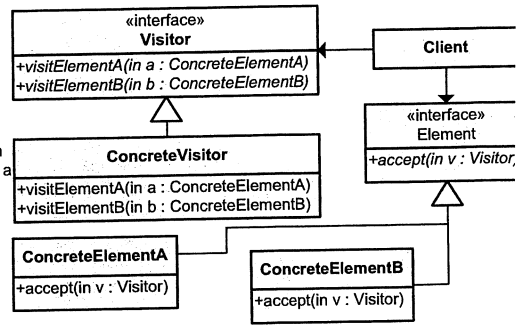
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.



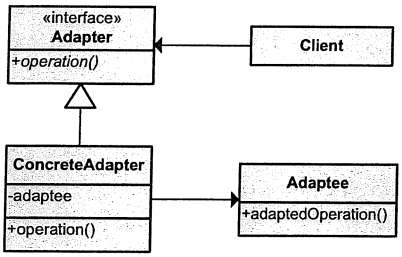
Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.



Adapter

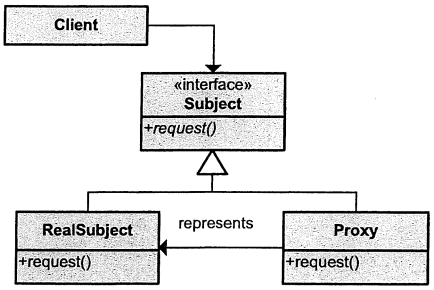
Type: Structural

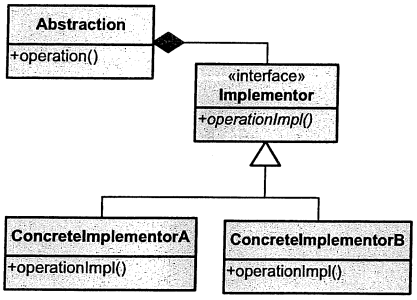
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.





Bridge

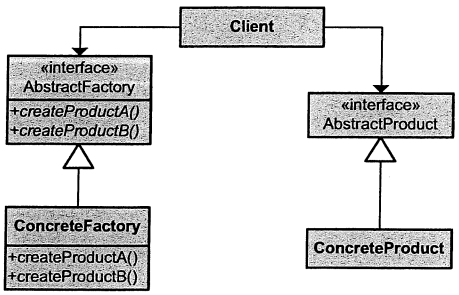
Type: Structural

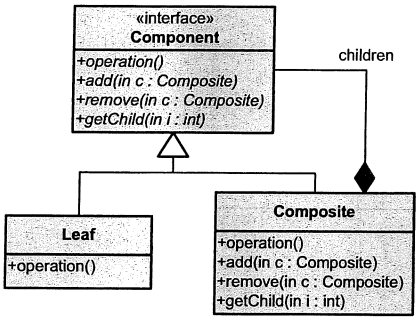
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.





Composite

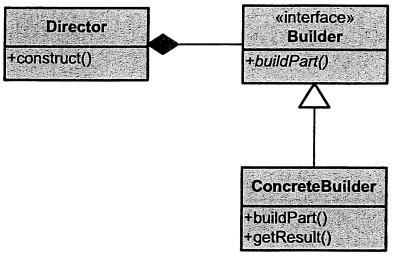
Type: Structural

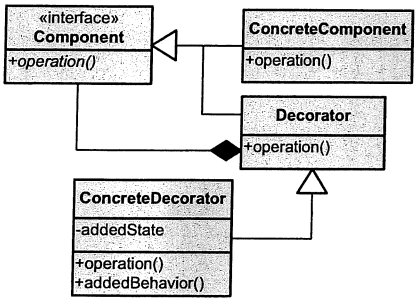
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.





Decorator

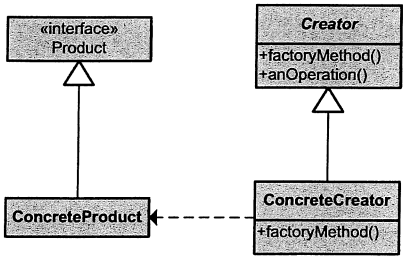
Type: Structural

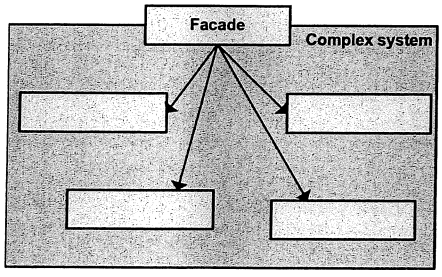
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.





Facade

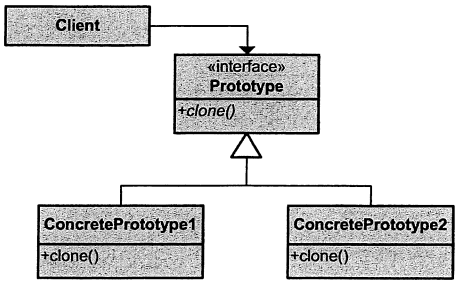
Type: Structural

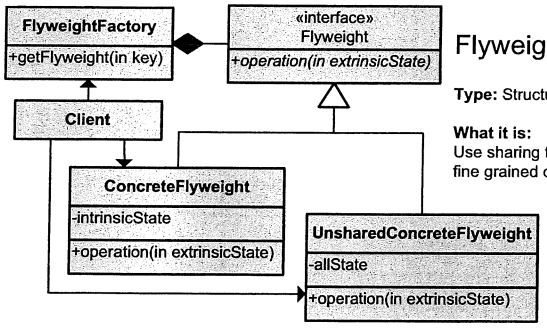
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.





Flyweight

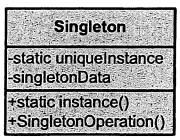
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Gamma, Erich, Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.