Analyse, Conception Objet

Gang of Four (GOF) Design Patterns

O. Boissier, SMA/G2I/ENS Mines Saint-Etienne, Olivier.Boissier@emse.fr, **Avril 2004**

Sommaire

Model View Controler (1)

- ➢ Première version en 1980, … VisualWorks, …, Java AWT, …
- ➢ Le MVC organisé en trois types d'objets :
  - ➢ **M**odèle (application, pas d 'interface utilisateur),
  - ➢ **V**ue (fenêtres sur l'application, maintenant l 'image du modèle),
  - ➢ **C**ontrôleur (réactions de l'interface aux actions de l'utilisateur, changement des états du modèle).
- ➢ flexibilité, réutilisation.

Exemple : MVC

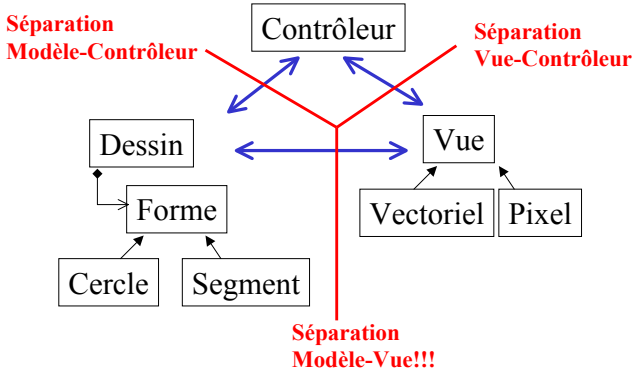
Concevoir l'architecture (classes en UML) d'un logiciel de dessin géométrique supportant les cercles, segments, groupes…

Parties à clarifier :

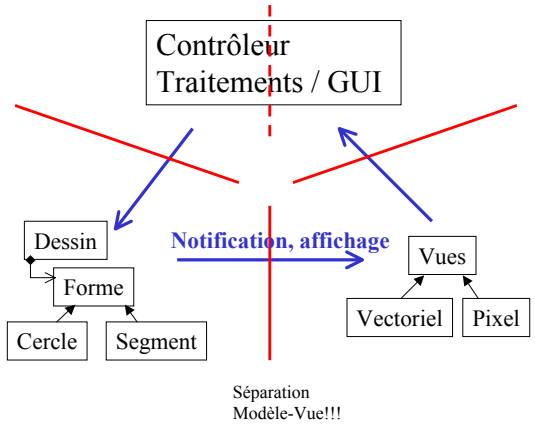
1. Structure interne / Dessins des formes
2. Changements synchronisés
3. Groupes d'objets (Group / Ungroup)
4. Comportements de la souris, des menus contextuels
5. Conversions en multiples formats…

Exemple 1/5 : MVC
Modèle - Vue - Contrôleur

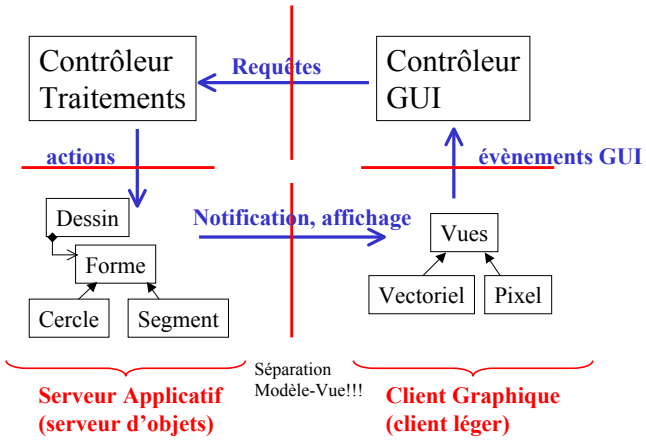
- Fichiers / Représentations Internes / Vues / Interactions utilisateurs



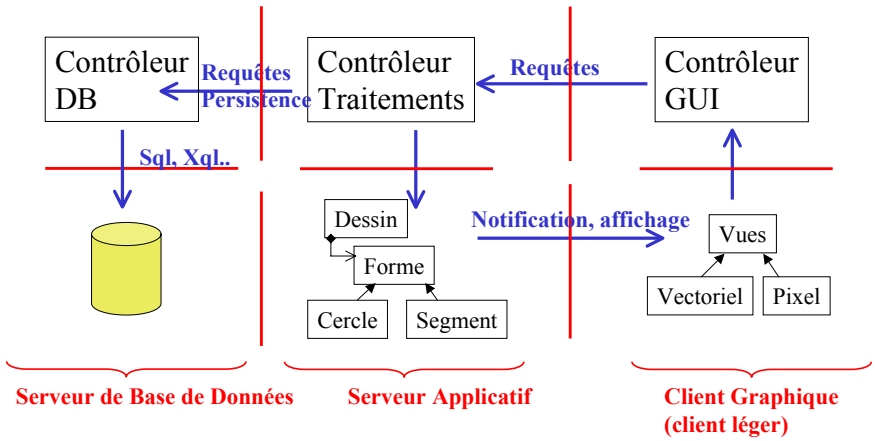
Exemple1/5 : MVC (Suite)
Contrôleur Traitements / GUI



Exemple1/5 : MVC (Suite)
Architecture 2 tiers

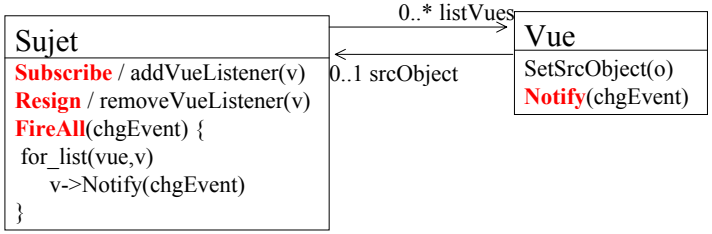


Exemple 1/5 : MVC (Suite)
Architecture 3 tiers



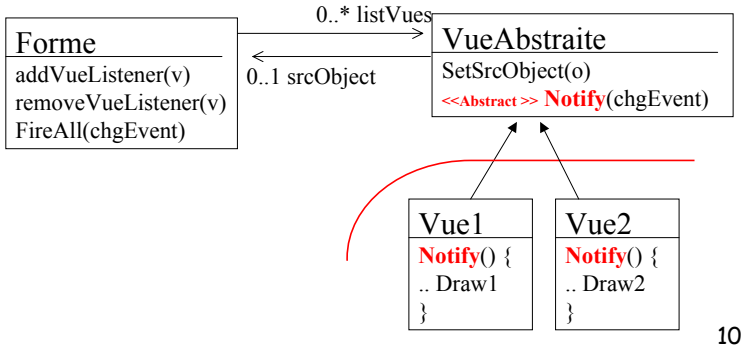
Exemple 2/5 : Publish & Subscribe

- Notifications de changement



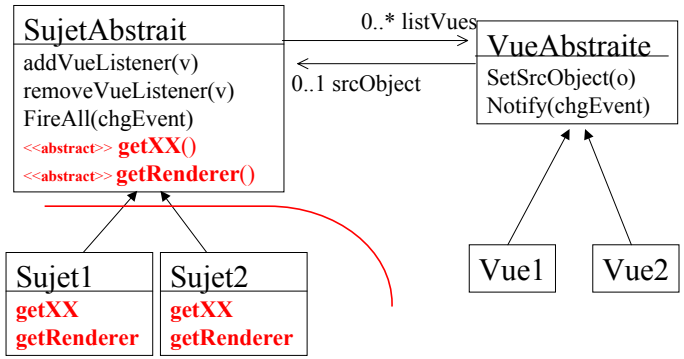
Exemple 2/5: Publish & Subscribe (Bis)

- Indépendance des Vues pour l'Objet



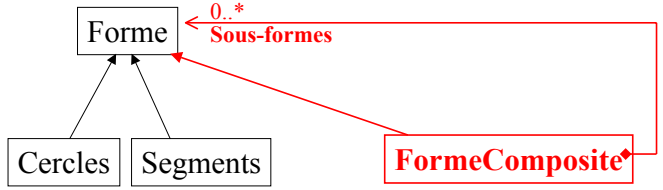
Exemple 2/5: Publish & Subscribe (Ter)

Indépendance des Objets pour les Vues (cf. MVC)



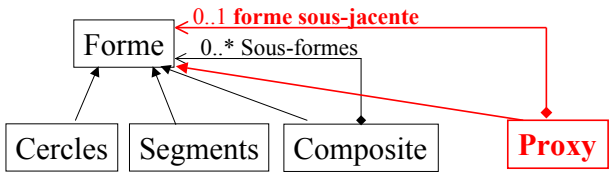
Exemple 3/5 : Composite..

- Group / Ungroup



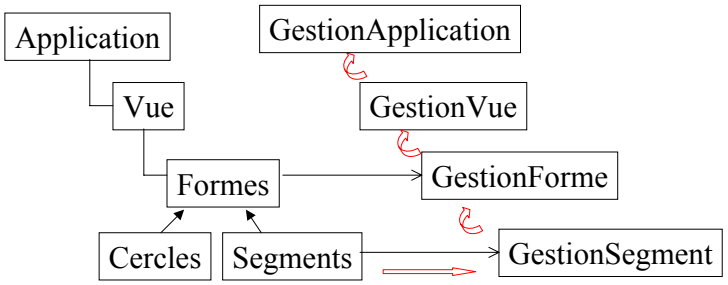
Exemple 3/5 : Composite, Proxy..

- Formes par procuration
(Rotation, Iconifiée, En cours de chargement, etc..)



Exemple 4/5 : Délégation, Chaîne de Responsabilité..

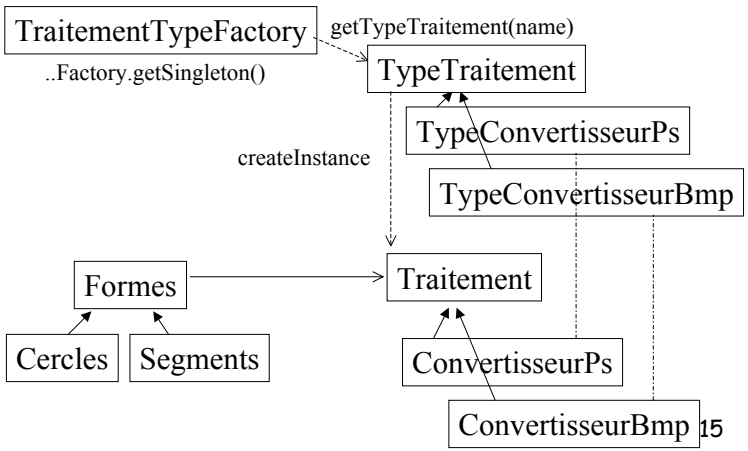
- Gestion de la souris, des évènements graphiques…



Menu Contextuel

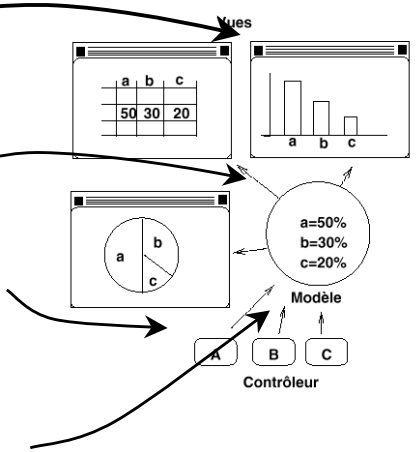
Exemple 5/5 : Stratégie, Visiteur, Factory, Singleton…

- Conversions Multiples, etc..



Model View Controler (2)

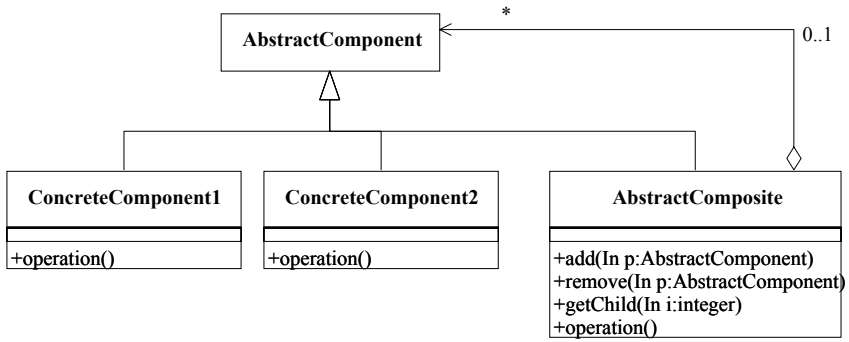
- View-View :
  - Composite, Decorator
- View-Model :
  - Observer
- View-Controler :
  - Strategy, Factory Method
- Controler-Model :
  - Command



Composite (1)

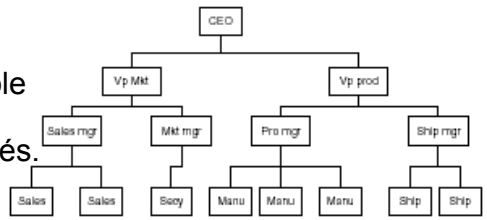
- Problème : on veut établir des structures arborescentes entre des objets et les traiter uniformément
- Solution : cf slide suivant
- Conséquences :
  + hiérarchies de classes dans lesquelles l'ajout de nouveaux composants est simple,
  + simplification du client qui n'a pas à se préoccuper de l'objet accédé,
  - MAIS il est difficile de restreindre et de vérifier le type des composants.
- Exemple :
  - java.awt.Component
  - java.awt.Container

Composite (2)



Composite (3)
Problème "Employés"

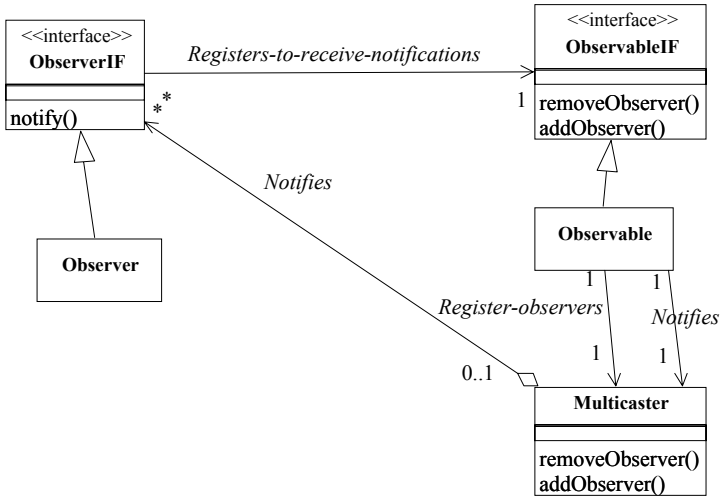
- Chacun des membres de la compagnie reçoit un salaire.
- A tout moment, il doit être possible de demander le coût d'un employé.
- Le coût d'un employé est calculé par :
  - Le coût d'un individu est son salaire.
  - Le coût d'un responsable est son salaire plus celui de ses subordonnés.



Observer (1)

- Problème : on veut assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur indépendance.
- Solution : cf slide suivant
- Conséquences :
  + couplage abstrait entre un sujet et un observeur, support pour la communication par diffusion,
  - MAIS des mises à jour inattendues peuvent survenir, avec des coûts importants.
- Exemple :
  - java.util.Observable
  - java.util.Observer

Observer (2)



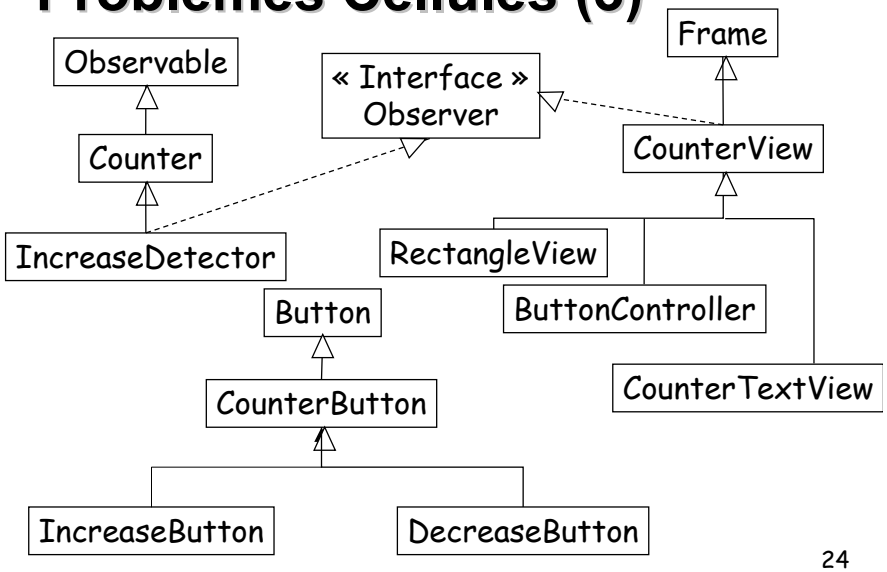
Observer
Problèmes Cellules (1)

- Counter : peut croître ou décroître d'une unité. A chaque fois que le Counter change de valeur, il notifie ses observers du changement.
- IncreaseDetector : affecté à un ou plusieurs Counter, compte le nombre de fois que leurs valeurs croît. Il notifie à ses observers des changements.
- CounterButton : classe abstraite, permettant de changer la valeur d'un Counter
- IncreaseButton (resp. DecreaseButton) fait croître (resp. décroître) la valeur de Counter auquel il est attaché à chaque fois que pressé

Observer
Problèmes Cellules (2)

- CounterView : vue observant un Counter. Utilisée comme Parent pour d'autres vues.
- CounterTextView : affichage de la valeur d'un compteur en Ascii (sous classe de CounterView)
- ButtonControler : fenêtre pour changer la valeur d'un Counter en utilisant les boutons (sous classe de CounterView)
- RectangleView : attaché à deux Counters (un pour la largeur et un autre pour la hauteur) (sous classe de CounterView)

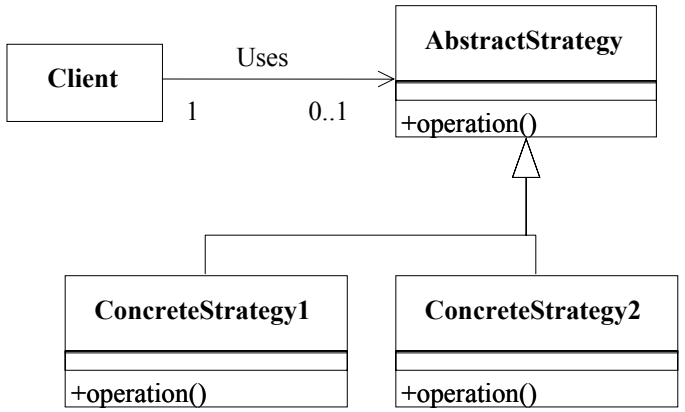
Observer
Problèmes Cellules (3)



Strategy (1)

- Problème : on veut (i) définir une famille d'algorithmes, (ii) encapsuler chacun et les rendre interchangeables tout en assurant que chaque algorithme peut évoluer indépendamment des clients qui l'utilisent.
- Solution : cf. slide suivant
- Conséquences :
  + Expression hiérarchique de familles d'algorithmes, élimination de tests pour sélectionner le bon algorithme, laisse un choix d'implémentation et une sélection dynamique de l'algorithme,
  - Les clients doivent faire attention à la stratégie, surcoût lié à la communication entre Strategy et Context, augmentation du nombre d'objets.

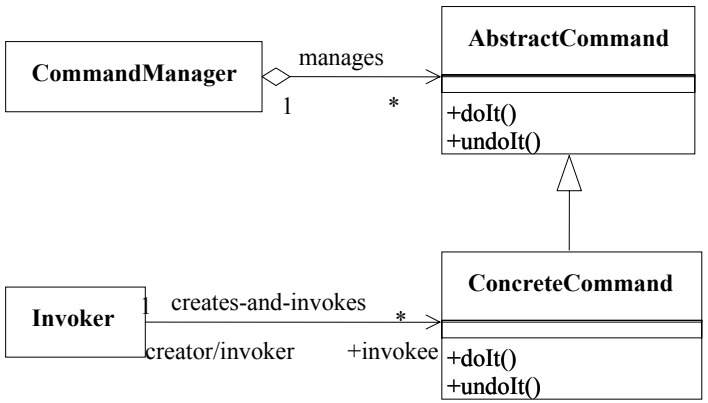
Strategy (2)



Command (1)

- Problème : on veut effectuer des requêtes sur des objets sans avoir à connaître leur structure.
- Solution : cf. slide suivant
- Conséquences :
  + Découplage entre l'objet qui appelle et celui qui exécute.
  + L'ajout de nouvelles commandes est aisée dans la mesure où la modification de classes existantes n'est pas nécessaire.

Command (2)



Model View Controler Problème (1)

- Création d 'une application « Surveillance et conversion de Température » : interface textuelle pour visualiser les valeurs de température avec deux boutons : un pour augmenter la température, et un autre pour décroître la température d'un degré. L'utilisateur peut également inscrire une valeur dans un champ textuel qui est pris en compte lorsqu'il fait Return.
- Faire hériter le Modèle de *java.util.Observable*.
  Donner à la classe des accesseurs pour acquérir l'information sur l'état courant.
  Ecrire les fonctions permettant de modifier l'état. Chacune de ces fonctions doit appeler *setChanged()* et *notifyObservers*() après le changement d'état.
- Créer une ou plusieurs vues.
  Chaque vue implémente l'interface *java.util.Observer* et implémente la méthode *update*.

Model View Controler Problème (2)

```
public class TemperatureModel {
    public double getF(){return temperatureF;}
    public double getC(){return (temperatureF - 32.0) * 5.0 / 9.0;}
    public void setF(double tempF) {
          temperatureF = tempF;
    }
    public void setC(double tempC) {
          temperatureF = tempC*9.0/5.0 + 32.0;
    }
    private double temperatureF = 32.0;
    …..
}
```

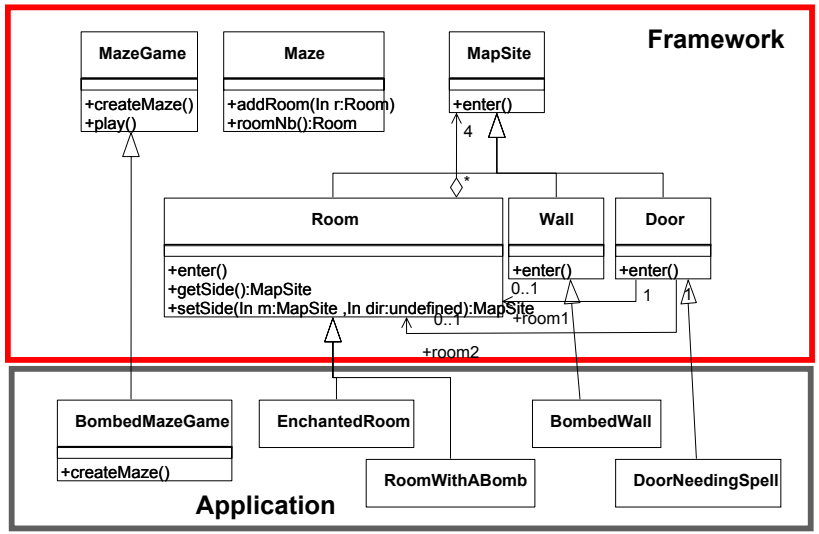

Sommaire

- Model View Controler
- ✓**Design Patterns de création**
- Design Patterns de structure
- Design Patterns de comportement
- Comment appliquer les Design Patterns
- Bibliographie

Design Patterns de création

- Rendre le système indépendant de la manière dont les objets sont créés, composés et représentés.
  - Encapsulation de la connaissance des classes concrètes à utiliser.
  - Cacher la manière dont les instances sont créées et combinées.
- Permettre *dynamiquement* ou *statiquement* de préciser **QUOI** (l'objet), **QUI** (l'acteur), **COMMENT** (la manière) et **QUAND** (le moment) de la création.
- Deux types de motifs :
  1. Motifs de création de classe (utilisation de l'héritage) : Factory
  2. Motifs de création d'objets (délégation de la construction à un autre objet) : AbstractFactory, Builder, Prototype

Maze Game



Maze Game

```
class MazeGame
{
    void Play() {…}
    public Maze createMaze()
    {
        Maze aMaze = new Maze();
        Room r1 = new Room( 1 );
        Room r2 = new Room( 2 );
        Door theDoor = new Door( r1, r2);
        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        r1.setSide( North, new Wall() );
        r1.setSide( East, theDoor );
        r1.setSide( South, new Wall() );
        r1.setSide( West, new Wall() );
        r2.setSide( North, new Wall() );
        r2.setSide( East, new Wall() );
        r2.setSide( South, new Wall() );
        r2.setSide( West, theDoor );
        return aMaze;
    }
}
```

Bombed Maze Game

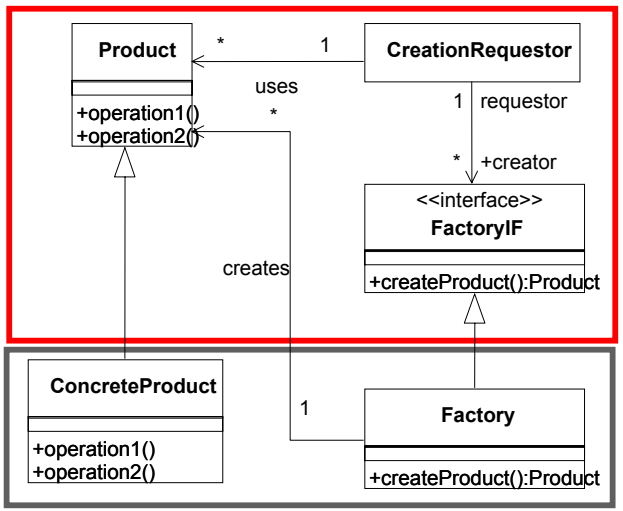
```
class BombedMazeGame extends MazeGame
{
    …
    public Maze createMaze()
    {
        Maze aMaze = new Maze();
        Room r1 = new RoomWithABomb( 1 );
        Room r2 = new RoomWithABomb( 2 );
        Door theDoor = new Door( r1, r2);
        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        r1.setSide( North, new BombedWall() );
        r1.setSide( East, theDoor );
        r1.setSide( South, new BombedWall() );
        r1.setSide( West, new BombedWall() );
        r2.setSide( North, new BombedWall() );
        r2.setSide( East, new BombedWall() );
        r2.setSide( South, new BombedWall() );
        r2.setSide( West, theDoor );
        return aMaze;
    }
}
```


Factory Method (1)

- **Problème** :
  - ce motif est à utiliser dans les situations où existe le besoin de standardiser le modèle architectural pour un ensemble d ’applications, tout en permettant à des applications individuelles de définir elles-mêmes leurs propres objets à créer.
- **Conséquences** :
  - + Elimination du besoin de code spécifique à l’application dans le code du framework (uniquement l’interface du Product)
  - - Multiplication du nombre de classes.

Factory Method (2)

Solution



Maze Game — Bombed Maze

```
class MazeGame
{
    void Play() {…}
    public Maze createMaze()
    {
        Maze aMaze = makeMaze();
        Room r1 = makeRoom( 1 );
        Room r2 = makeRoom( 2 );
        Door theDoor = makeDoor( r1, r2);
        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        r1.setSide( North, makeWall() );
        r1.setSide( East, theDoor );
        r1.setSide( South, makeWall() );
        r1.setSide( West, makeWall() );
        r2.setSide( North, makeWall() );
        r2.setSide( East, makeWall() );
        r2.setSide( South, makeWall() );
        r2.setSide( West, theDoor );
        return aMaze;
    }
}
```

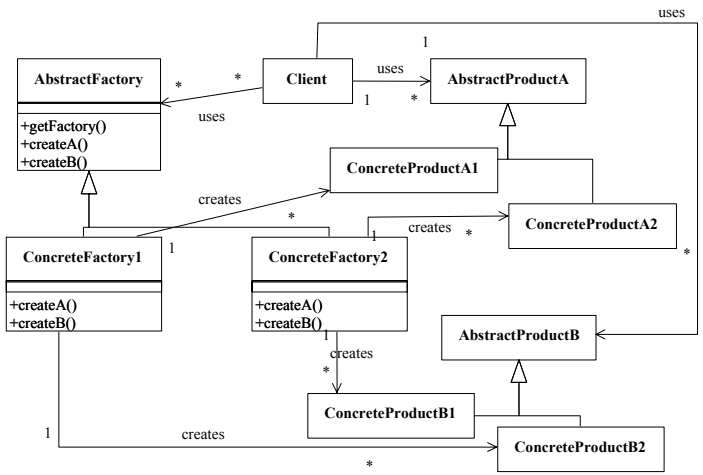
```
class BombedMazeGame extends MazeGame
{
    …
    public Wall makeWall()
    {
      return new BombedWall();
    }

    public Room makeRoom(int i)
    {
      return new RoomWithABomb( i );
    }
}
```


AbstractFactory == Kit (1)

- Problème :
  - ce motif est à utiliser dans les situations où existe le besoin de travailler avec des familles de produits tout en étant indépendant du type de ces produits
  - doit être configuré par une ou plusieurs familles de produits.
- Conséquences :
  - +Séparation des classes concrètes, des classes clients :
    - les noms des classes produits n'apparaissent pas dans le code client.
    - Facilite l'échange de familles de produits.
    - Favorise la cohérence entre les produits.
  - + Le processus de création est clairement isolé dans une classe.
  - - la mise en place de nouveaux produits dans l'AbstractFactory n'est pas aisée.

Dans API Java : java.awt.Toolkit

AbstractFactory == Kit (2)



Maze Game — Bombed Maze

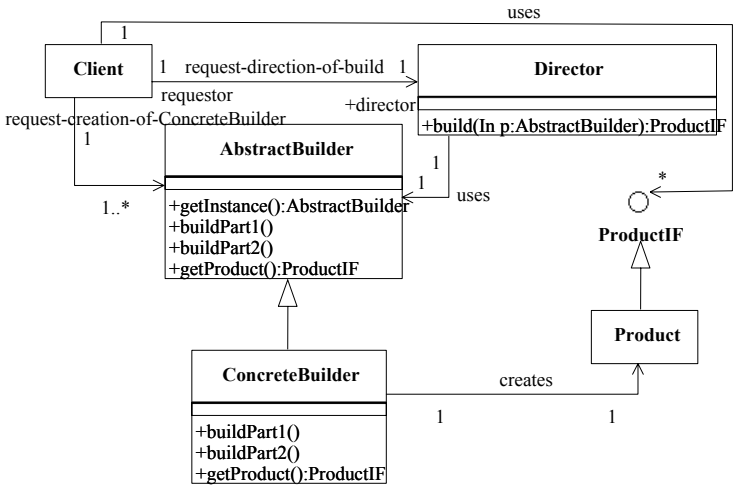
```
class MazeGame
{
    void Play() {…}
    public Maze createMaze(MazeFactory f)
    {
        Maze aMaze = f.makeMaze();
        Room r1 = f.makeRoom( 1 );
        Room r2 = f.makeRoom( 2 );
        Door theDoor = f.makeDoor( r1, r2);
        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        r1.setSide( North, f.makeWall() );
        r1.setSide( East, theDoor );
        r1.setSide( South, makeWall() );
        r1.setSide( West, makeWall() );
        r2.setSide( North, makeWall() );
        r2.setSide( East, makeWall() );
        r2.setSide( South, makeWall() );
        r2.setSide( West, theDoor );
        return aMaze;
    }
}
```

```
class BombedMazeFactory extends
MazeFactory
{
    …
    public Wall makeWall()
    {
      return new BombedWall();
    }

    public Room makeRoom(int i)
    {
      return new RoomWithABomb( i );
    }
}
```

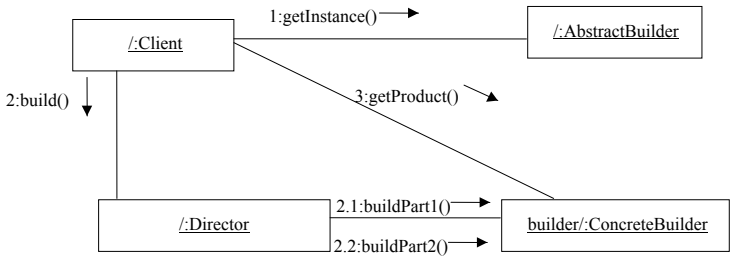
Builder (1)

- **Problème** :
  - ce motif est intéressant à utiliser lorsque l'algorithme de création d'un objet complexe doit être indépendant des constituants de l'objet et de leurs relations, ou lorsque différentes représentations de l'objet construit doivent être possibles.
- **Conséquences** :
  - + Variation possible de la représentation interne d'un produit : l'implémentation des produits et de leurs composants est cachée au Director.
    Ainsi la construction d 'un autre objet revient à définir un nouveau Builder.
  - + Isolation du code de construction et du code de représentation du reste de l'application.
  - + Meilleur contrôle du processus de construction.

Builder (2)



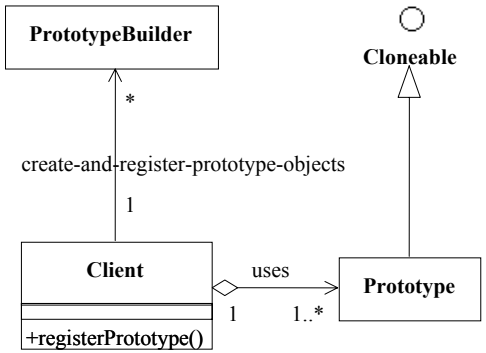
Builder (3)



Prototype (1)

- **Problème** :
  - le système doit être indépendant de la manière dont ses produits sont créés, composés et représentés : les classes à instancier sont spécifiées au moment de l'exécution. La présence de hiérarchies de Factory similaires aux hiérarchies de produits doivent être évitées. Les combinaisons d'instances sont en nombre limité.
- Conséquences :
  - mêmes conséquences que Factory et Builder.
- Dans API Java : java.lang Interface Cloneable

Prototype (2)



Bilan sur le problème du labyrinthe

- Factory : CreateMaze a un objet en paramètre utilisé pour créer les composants du labyrinthe, on peut changer la structure du labyrinthe en passant un autre paramètre
- Builder : CreateMaze a un objet paramètre capable de créer lui même un labyrinthe dans sa totalité, il est possible de changer la structure du labyrinthe en dérivant un nouvel objet.
- FactoryMethod : CreateMAze appelle des fonctions virtuelles au lieu de constructeurs pour créer les composants du labyrinthe, il est alors possible de modifier la création en dérivant une nouvelle classe et en redéfinissant ces fonctions virtuelles.
- Prototype : CreateMaze est paramétré par des composants prototypiques qu'il copie et ajoute au labyrinthe, il est possible de changer la structure du labyrinthe en fournissant d'autres composants.

Singleton

- **Problème** :
  - avoir une seule instance d'une classe et pouvoir l'accéder et la manipuler facilement.
- **Solution** :
  - une seule classe est nécessaire pour écrire ce motif.
- **Conséquences** :
  - l'unicité de l'instance est complètement contrôlée par la classe elle même. Ce motif peut facilement être étendu pour permettre la création d'un nombre donné d'instances.

Singleton (2)

```
// Only one object of this class can be created
class Singleton
{
    private static Singleton _instance = null;
    private Singleton() { fill in the blank }
    public static Singleton getInstance()
    {
        if ( _instance == null ) _instance = new Singleton();
        return _instance;
    }
    public void otherOperations() { blank; }
}
class Program
{
    public void aMethod()
    {
    X = Singleton.getInstance();
    }
}
```

Résumé sur les Design Patterns de création

- **Le Factory Pattern** est utilisé pour choisir et retourner une instance d 'une classe parmi un nombre de classes similaires selon une donnée fournie à la factory.
- **Le Abstract Factory Pattern** est utilisé pour retourner un groupe de classes.
- **Le Builder Pattern** assemble un nombre d 'objets pour construire un nouvel objet, à partir des données qui lui sont présentées. Fréquemment le choix des objets à assembler est réalisé par le biais d 'une Factory.
- **Le Prototype Pattern** copie ou clone une classe existante plutôt que de créer une nouvelle instance lorsque cette opération est coûteuse.
- **Le Singleton Pattern** est un pattern qui assure qu 'il n 'y a qu 'une et une seule instance d 'un objet et qu 'il est possible d 'avoir un accés global à cette instance.

Sommaire

Design Patterns de structure

- Abstraction de la manière dont les classes et les objets sont composés pour former des structures plus importantes.
- Deux types de motifs :
  - Motifs de structure de classes avec l'utilisation de l'héritage pour composer des interfaces et/ou des implémentations (ex : Adapter).
  - Motifs de structure d'objets avec la composition d'objets pour réaliser de nouvelles fonctionnalités :
    - ajouter un niveau d'indirection pour accéder à un objet (ex : Adapter d'objet, Bridge, Facade, Proxy),
    - composition récursive pour organiser un nombre quelconque d'objets (ex : Composite).

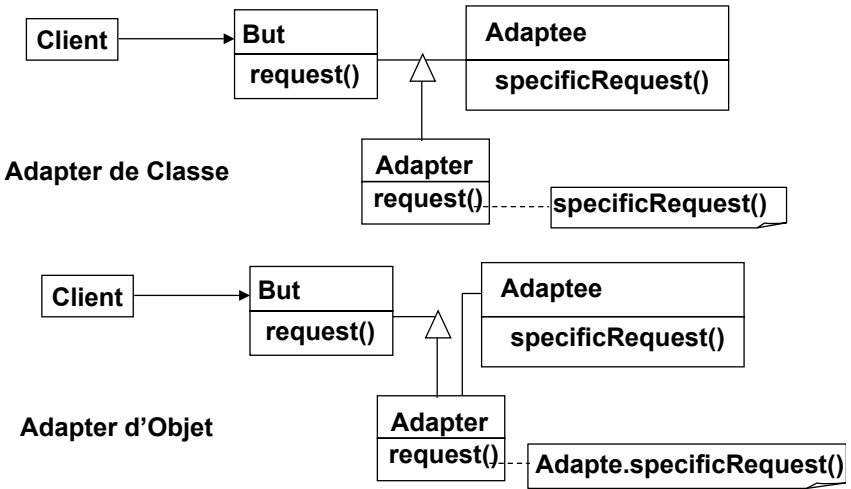
Adapter (1)

- Problème
  - Utilisation d'une classe existante dont l'interface ne nous convient pas (→ convertir l'interface d'une classe en une autre)
  - Utilisation de plusieurs sous-classes dont l'adaptation des interfaces est impossible par dérivation (→ Object Adapter)
- Solution (cf. slide suivant)
- Conséquence
  - Adapter de classe
    + il n'introduit qu'une nouvelle classe, une indirection vers la classe adaptée n'est pas nécessaire.

    MAIS il ne fonctionnera pas dans le cas où la classe adaptée est racine d'une dérivation.
  - Adapter d'objet
    + il peut fonctionner avec plusieurs classes adaptées.

    MAIS il peut difficilement redéfinir des comportements de la classe adaptée.

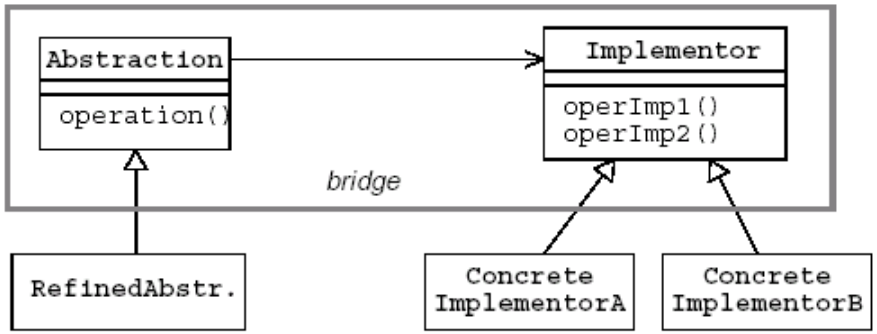
Adapter (2)



Bridge (1)

- Problème : ce motif est à utiliser lorsque l'on veut découpler l'implémentation de l'abstraction de telle sorte que les deux puissent varier indépendamment.
- Solution (cf. slide suivant)
- Conséquences :
  - \+ interfaces et implémentations peuvent être couplées/découplées lors de l'exécution.

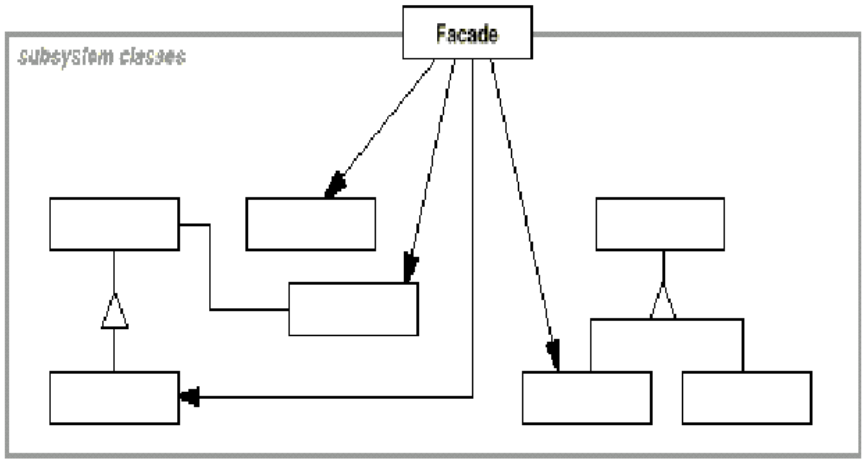
Bridge (2)



Facade (1)

- Problème : ce motif est à utiliser pour faciliter l'accès à un grand nombre de modules en fournissant une couche interface.
- Solution (cf. slide suivant)
- Conséquences :
  - \+ facilite l'utilisation de sous-systèmes
  - \+ favorise un couplage faible entre des classes et l'application
  - \- MAIS des fonctionnalités des classes interfacées peuvent être perdues selon la manière dont est réalisée la Facade.

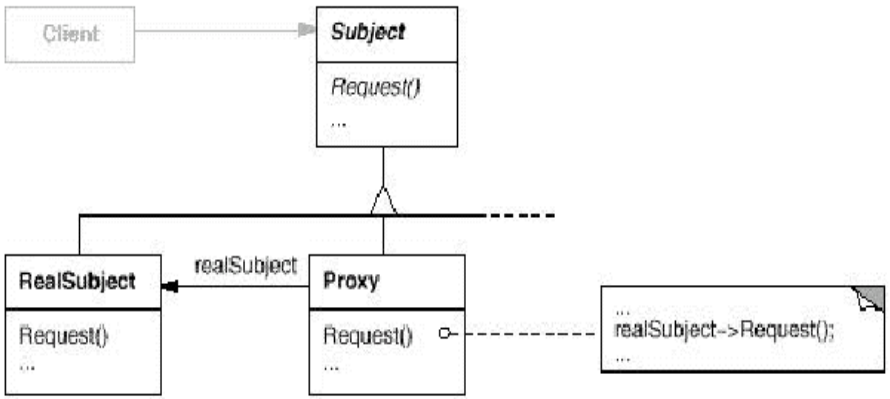
Facade (2)



Proxy (1)

- Problème : ce motif est à utiliser pour agir par procuration pour un objet afin de contrôler les opérations qui lui sont appliquées.
  - Masquer des problèmes d'accès (ex : fichier),
  - Différer l'exécution d'opérations coûteuses,
  - Contrôler les droits d'accès
- Solution (cf. slide suivant)
- Conséquences :
  - \+ ajout d'un niveau d'indirection lors de l'accès d'un objet permettant de cacher le fait que l'objet est dans un autre espace d'adressage, n'est pas créé,…

Proxy (2)



Sommaire

- Model View Controler
- Design Patterns de création
- Design Patterns de structure

✓**Design Patterns de comportement**

- Comment appliquer les Design Patterns
- Bibliographie

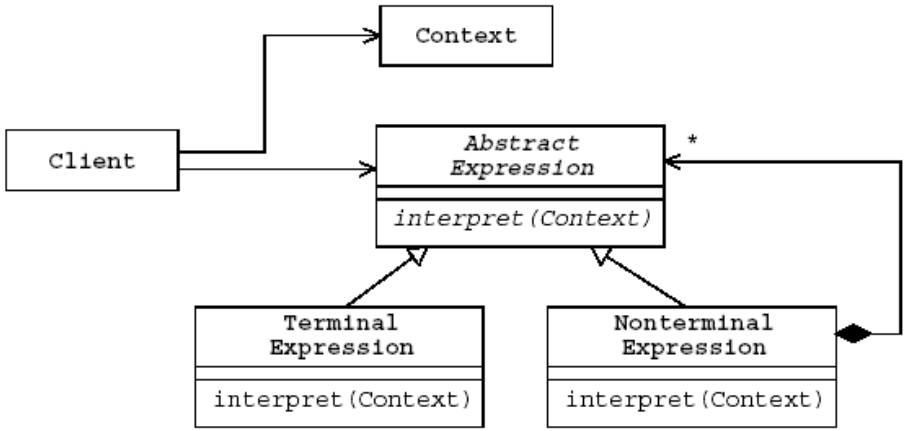
Design Patterns de comportement

- Description de structures d'objets ou de classes avec leurs interactions.
- Deux types de motifs :
  - Motifs de comportement de classes : utilisation de l'héritage pour répartir les comportements entre des classes (ex : Interpreter).
  - Motifs de comportement d'objets avec l'utilisation de l'association entre objets :
    - pour décrire comment des groupes d'objets coopèrent (ex : Mediator),
    - pour définir et maintenir des dépendances entre objets (ex : Observer),
    - pour encapsuler un comportement dans un objet et déléguer les requêtes à d'autres objets (ex : Strategy, State, Command)
    - pour parcourir des structures en appliquant des comportements (ex : Visitor, Iterator).

Interpreter (1)

- Problème : ce motif est à utiliser lorsque l'on veut représenter la grammaire d'un langage et l'interpréter, lorsque :
  - La grammaire est simple,
  - l'efficacité n'est pas critique
- Solution (cf. Slide suivant)
- Conséquences :
  - + facilité de changer et d'étendre la grammaire.
  - + l'implémentation de la grammaire est simple,
  - - MAIS les grammaires complexes sont dures à tenir à jour.

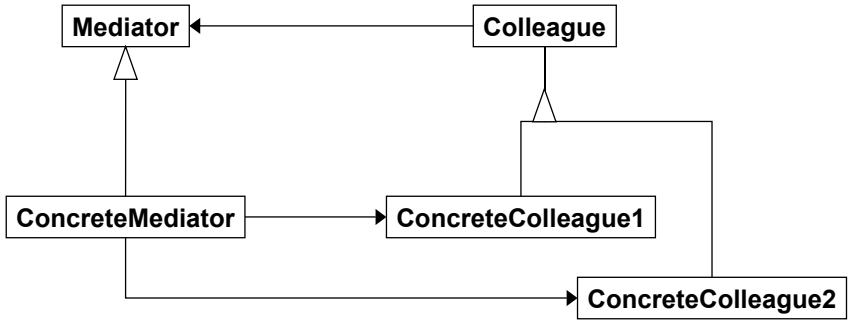
Interpreter (2)



Mediator (1)

- Problème : Assurer l'interaction entre différents objets en assurant leur indépendance :
  - Les interactions entre les objets sont bien définies mais conduisent à des interdépendances difficiles à comprendre, ou
  - La réutilisation d'un objet est difficile de part ses interactions avec plusieurs objets.
- Solution
- Conséquences :
  - + limitation de la compartimentation : le médiator contient le comportement qui serait distribué sinon entre les différents objets, indépendance des ''Colleagues'',
  - + simplification des protocoles (many-to-many → one-to-many),
  - + abstraction des coopérations entre objets,
  - - MAIS centralisation du contrôle, complexité possible du Médiator

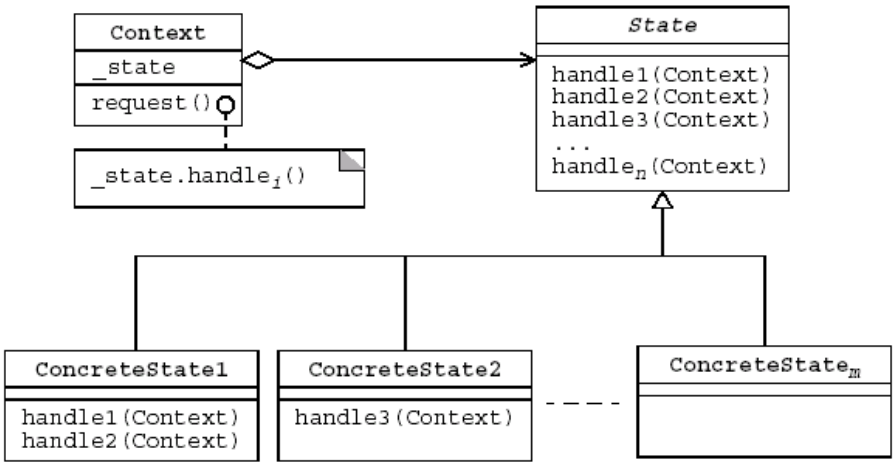
Mediator (2)



State (1)

- Problème : ce motif est à utiliser lorsque l'on veut qu'un objet change de comportement lorsque son état interne change.
- Solution (cf. slide suivant)
- Conséquences :
  - + possibilité d'ajouter ou de retirer des états et des transitions de manière simple,
  - + suppression de traitements conditionnels,
  - + les transitions entre états sont rendues explicites.

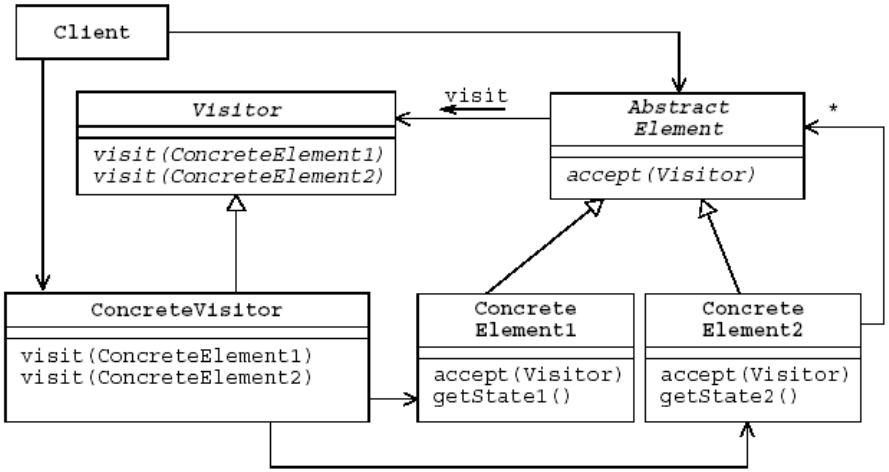
State (2)



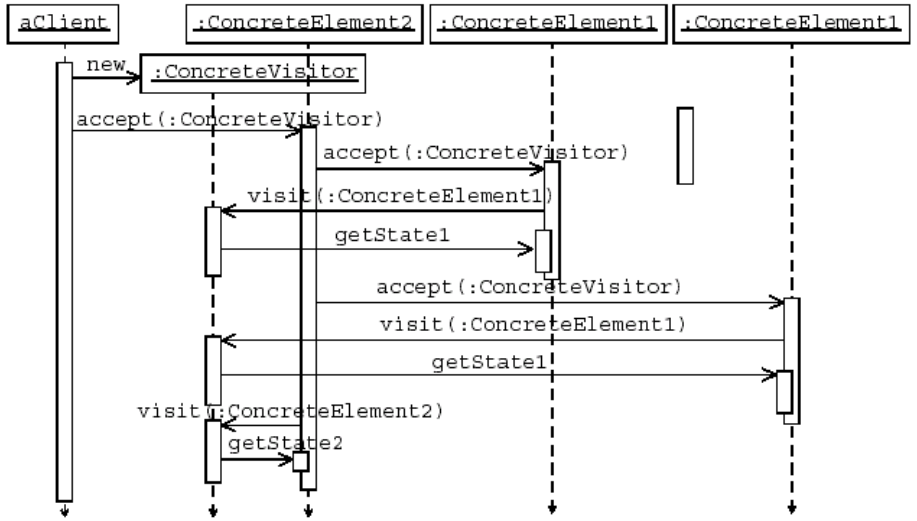
Visitor (1)

- Problème : ce motif est à utiliser lorsque :
  - Des opérations doivent être réalisées dans une structure d'objets comportant des objets avec des interfaces différentes,
  - Plusieurs opérations distinctes doivent être réalisées sur des objets d'une structure,
  - La classe définissant la structure change rarement mais de nouvelles opérations doivent pouvoir être définies souvent sur cette structure.
- Solution (cf. slide suivant)
- Conséquences :
  - + l'ajout de nouvelles opérations est aisé
  - + union de différentes opérations et séparations d'autres,
  - - MAIS l'ajout de nouvelles classes concrètes est freiné.

Visitor (2)



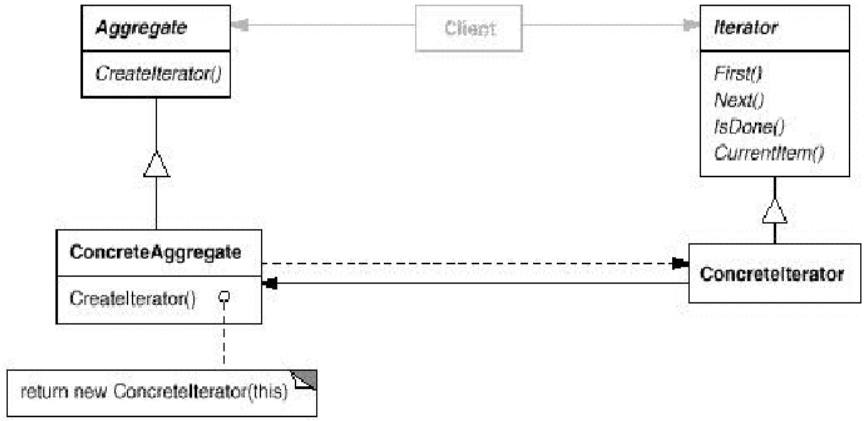
Visitor (3)



Iterator (1)

- Problème : ce motif est à utiliser pour parcourir une collection d'éléments sans accéder à sa structure interne.
- Solution (cf. slide suivant)
- Conséquences
  - + des variations dans le parcours d'une collection sont possibles,
  - + simplification de l'interface de la collection
  - + plusieurs parcours simultanés de la collection sont possibles

Iterator (2)



Sommaire

- Model View Controler
- Design Patterns de création
- Design Patterns de structure
- Design Patterns de comportement

✓**Comment appliquer les Design Patterns**

➢Bibliographie

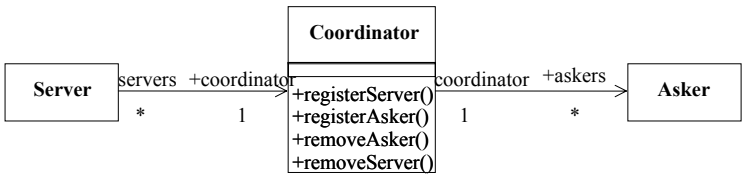
Simulation discrète

- Programme simple de simulation d 'une banque
  - arrivée aléatoire des Clients (Poisson)
  - temps de service des caissiers tirés aléatoirement
- Paramètres d 'entrée :
  - durée de la simulation,
  - nombre de caissiers,
  - efficacité de chaque caissier,
  - intervalle moyen entre deux arrivées.
- Paramètres de sortie :
  - durée réelle de la simulation
  - nombre de clients servis
  - taux d 'occupation de chaque caissier
  - temps moyen d 'attente des clients
- Mécanismes généralisables :
  - mise en correspondance caissier/client
  - simulation d 'événements discrets

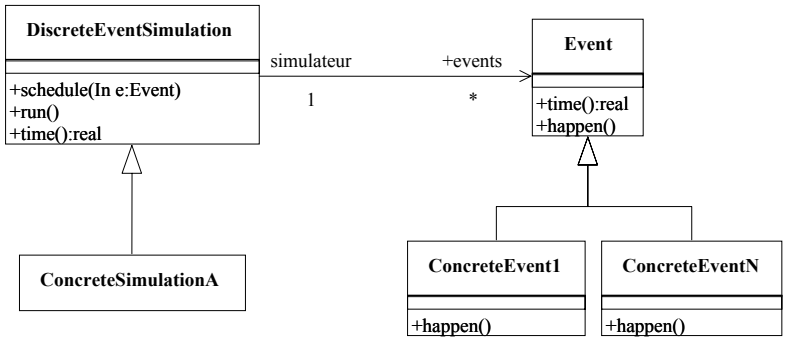
Simulation discrète (suite)

- Coordinateur
  - gère un ensemble de demandeurs et un ensemble de fournisseurs
  - met en correspondance, dès que possible, un nouveau demandeur avec l 'un des fournisseurs recensés
  - un objet demandant un service peut ne pas connaître l 'ensemble des objets pouvant satisfaire sa requête. Le coordinateur se charge de trouver un tel objet, et de les mettre en correspondance.
- Simulation discrète
  - rendre le mécanisme de gestion des événements indépendant de la nature de ces derniers

Broker (suite)



Simulation



Application totale