Design Patterns

Plan

- Orienté-Objet & Design Patterns
- Généralités sur les Design Patterns
- Étude de Cas
- Utilisation & méthode d’apprentissage
- Conclusion

Objectifs / Positionnement

- Pré requis
  - connaissance Orientée-Objet
  - Langage OO : C++ / Java..
  - Concepts de Librairies
- Buts
  - Concepts abstraits
  - Vocabulaire des concepts (complémentaire d'UML)
  - Nouvelle vision du monde du logiciel
- Non – Buts
  - Pas liés à un langage précis
  - Pas un livre d'apprentissage, pas de recettes !

L'Héritage en Orienté-Objets

- 3 Façons de réutiliser les Objets
  - Héritages (d'interface / de code)
  - Composition
  - Templates (généricité de types..)
- **Héritage de code** : souvent utilisé **à tords**
- L'**héritage d'interface** : Light Motifs des Design Patterns

Limitation d'une approche naïve de l'Orienté-Objets

- Recensement « Merisien » des objets
  - Données, pas Interfaces !
  - Objets fonctionnels seulement, Pas informatiques!
- Héritage de code
  - forte corrélation classes / sous-classes..
- Traitements mélangés entre classes
  - Grande difficulté de compréhension
  - insuffisance des diagrammes de classes de UML

Buts : Rôles des objets

- Limitation des dépendances / connaissances entre objets
- Introduction de **dépendances dynamiques tardives** (« late binding »)
- Par opposition : suppression des **dépendances à la compilation**..
- Rôles des objets systématiquement épurés, et définis par des interfaces

1 rôle => 1 interface + délégation à 1 objet

Possibilité de changement ouverte

Définition : Pattern

- Un patron décrit à la fois un **problème qui se produit très** fréquemment dans l'environnement et l'architecture de la **solution à ce problème de telle façon que l'on puisse utiliser** cette solution des milliers de fois sans jamais l'**adapter deux fois** de la même manière.

C. Alexander

➔ Décrire avec succès des types de **solutions récurrentes à des problèmes communs dans des** types de situations

23 Patterns / 3 classifications

Des objets où, comment, pourquoi faire ?..

⇒ Identification et rôles des objets et des relations

- 1) Modèles **Créateurs**
  - Créer un objet / Accéder à un objet
- 2) Modèles **Structuraux**
  - Combiner les objets en structures
- 3) Modèles de **Comportement**
  - Utiliser les objets pour implanter des fonctionnalités

Retour sur les 23 Patterns

- Les 23 Patterns se trouvent **partout**
- **Sous formes réduites, déguisées, renommées…**

⇒ **Lire des programmes … Savoir les reconnaître et comprendre l'architecture**

⇒ **Ecrire : savoir en mettre partout (!!), en respectant les concepts**

Design Patterns du GoF

(Gamma, Helm, Johnson, Vlissides)

		Catégorie		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Factory Method	Adapter	Interpreter
				Template Method
	Objet	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Description des 23 Patterns ? / Réflexion de chacun !!

- Découverte
  - Bon sens, mais c'est bien sûr..
- 1ère Lecture
  - Catalogue Universitaire ?
- 1ère pratique
  - Je connais!.. Je vais réessayer pareil…
  - Oups.. Je dois relire quelques détails..
- 2ème lecture
  - C'est très fort
- 2ème pratique
  - On les vois partout ! On en met partout !

Présentation d’un Design Pattern

- **Nom du pattern**

– utilisé pour décrire le pattern, ses solutions et les conséquences en un mot ou deux

- **Problème**

– description des conditions d ’applications. Explication du problème et de son contexte

- **Solution**

– description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème ; utilisation de diagrammes de classes, de séquences, …
vision statique ET dynamique de la solution

- **Conséquences**

– description des résultats (effets induits) de l ’application du pattern sur le système (effets positifs ET négatifs)

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

Singleton (1)

- **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() {
      ...
   }

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
   ...
}

class Program {
   public void aMethod() {
      Singleton X = Singleton.getInstance();
   }
}
```

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)

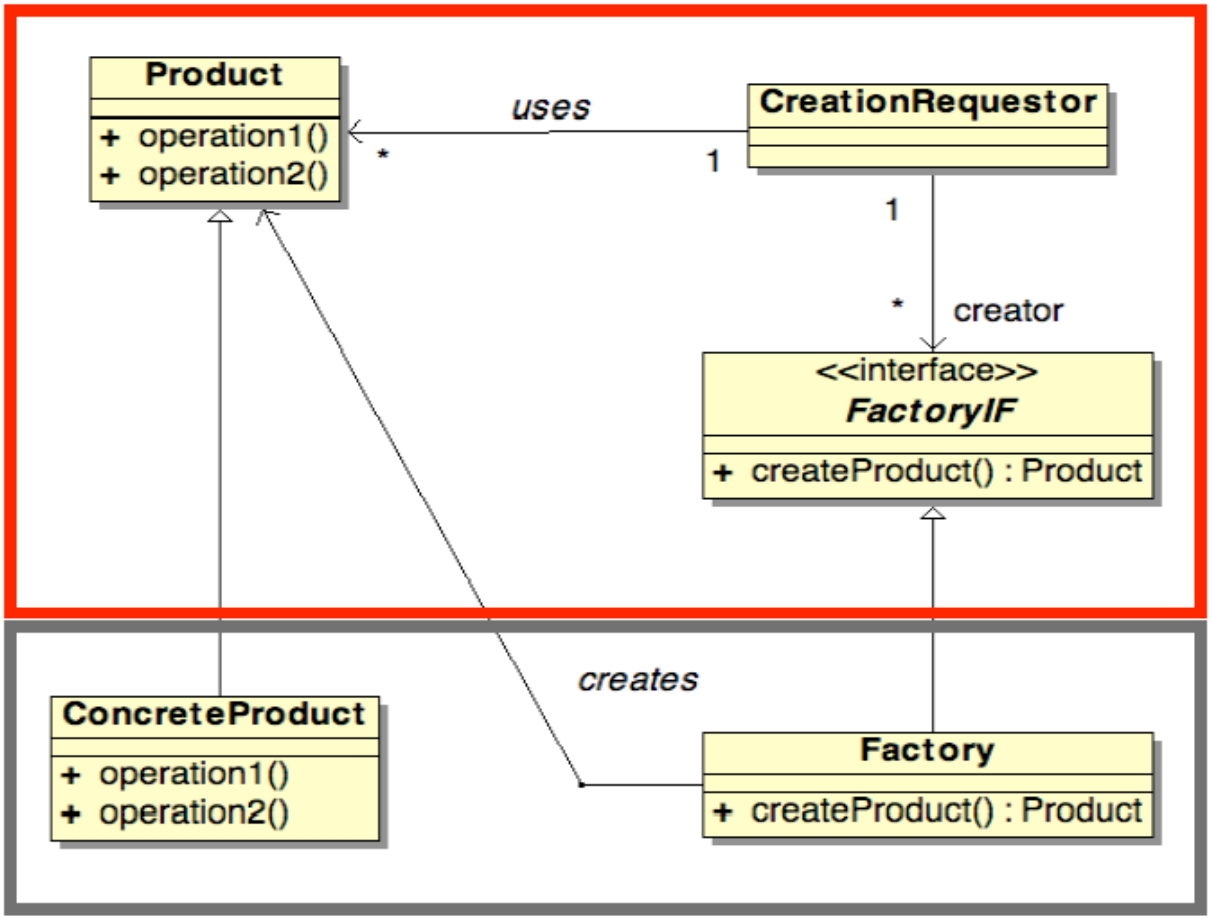
Solution

Product – defines the interface of objects the factory method creates

ConcreteProduct – implements Product interface

FactoryIF – declares the factory method which returns the object of type Product

Factory – overrides the factory method to return an instance of a ConcreteProduct



Abstract Factory (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

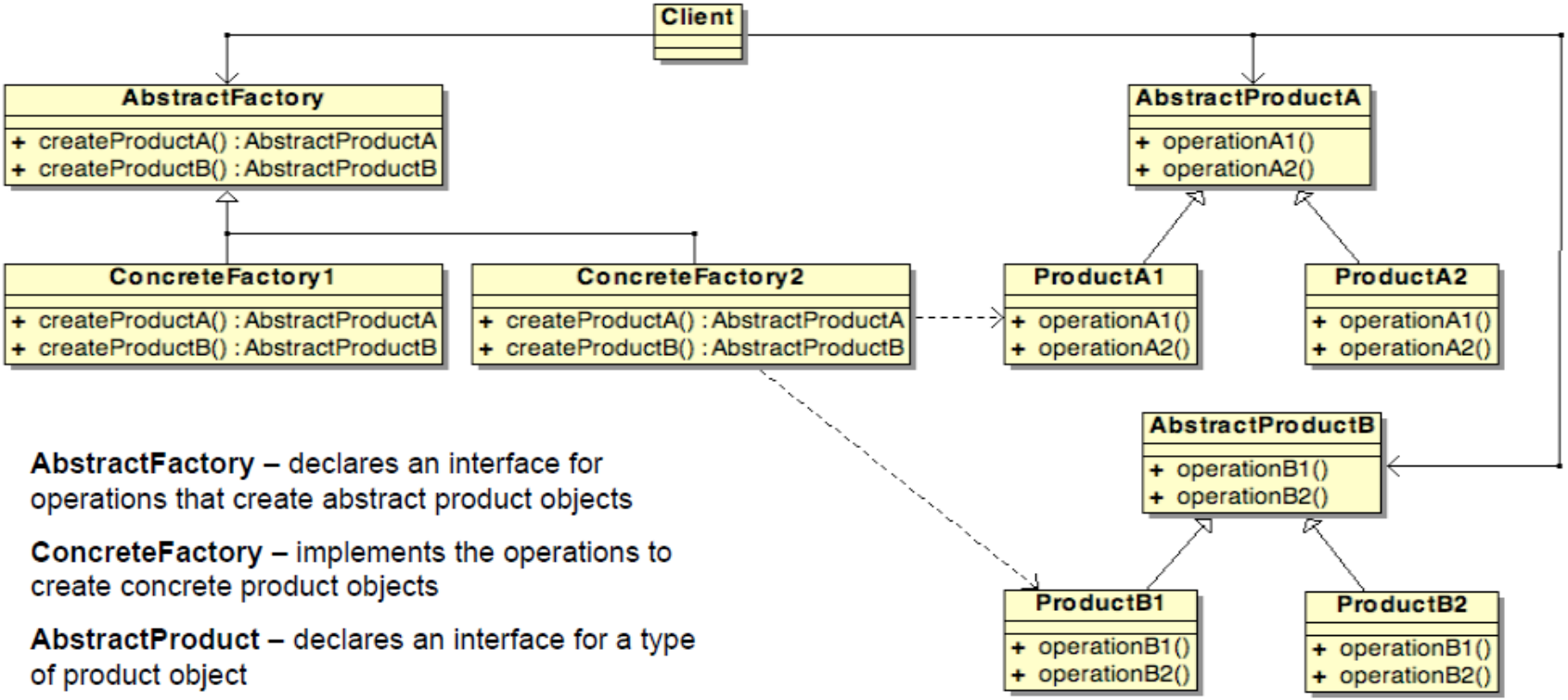
+ Le processus de création est clairement isolé dans une classe

– la mise en place de nouveaux produits dans l'AbstractFactory n'est pas aisée

- **Exemple**

– java.awt.Toolkit

Abstract Factory (2)



AbstractFactory – declares an interface for operations that create abstract product objects

ConcreteFactory – implements the operations to create concrete product objects

AbstractProduct – declares an interface for a type of product object

ConcreteProduct
– defines a product object to be created by the corresponding concrete factory
– implements the AbstractProduct interface

Client – uses interfaces declared by AbstractFactory and AbstractProduct classes

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**

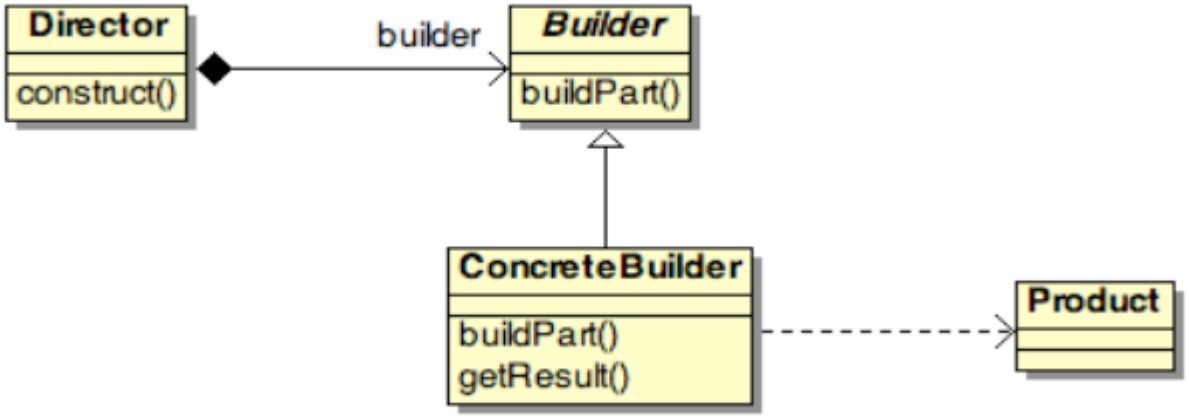
+ Variation possible de la représentation interne d'un produit

- 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)



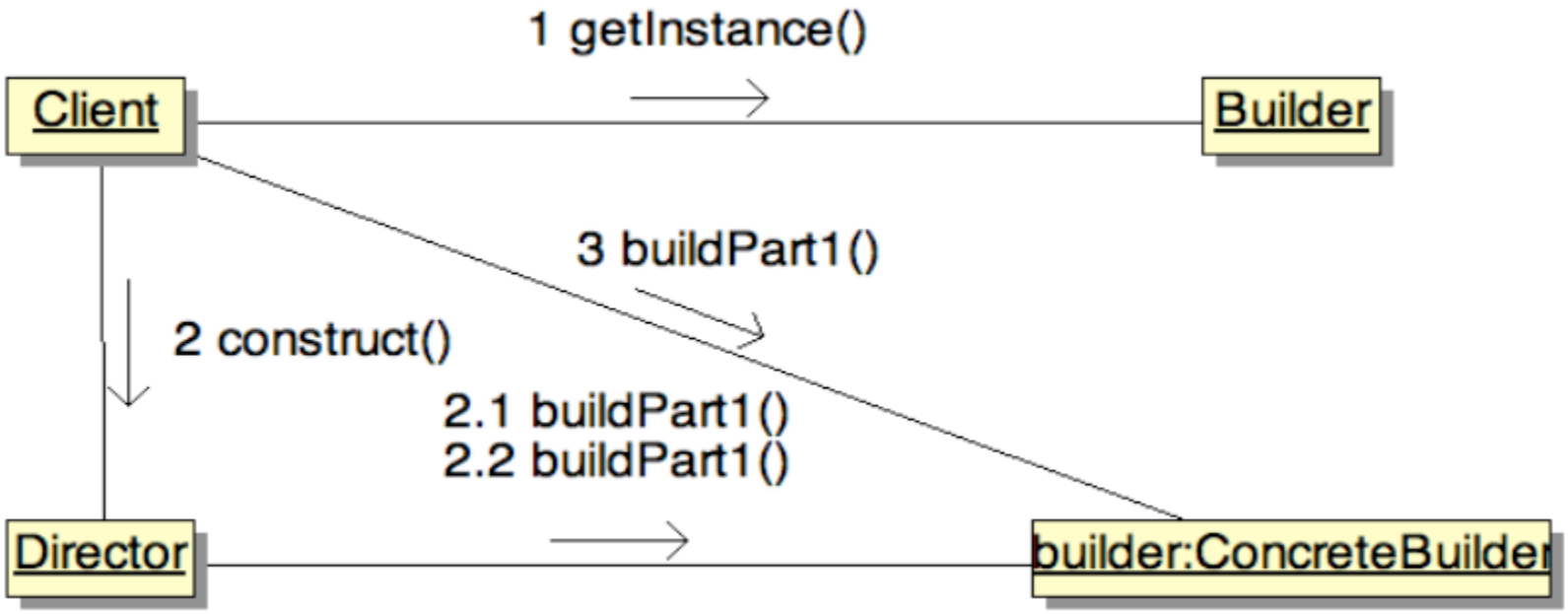
Builder – interface for creating parts of a Product object

ConcreteBuilder – constructs and assembles parts of the product by implementing the Builder interface

Director – constructs an object using Builder Interface

Product – represents the complex object under construction

Builder (3)


1 getInstance()
Client
Builder
3 buildPart1()
2 construct()
2.1 buildPart1()
2.2 buildPart1()
Director
builder:ConcreteBuilder

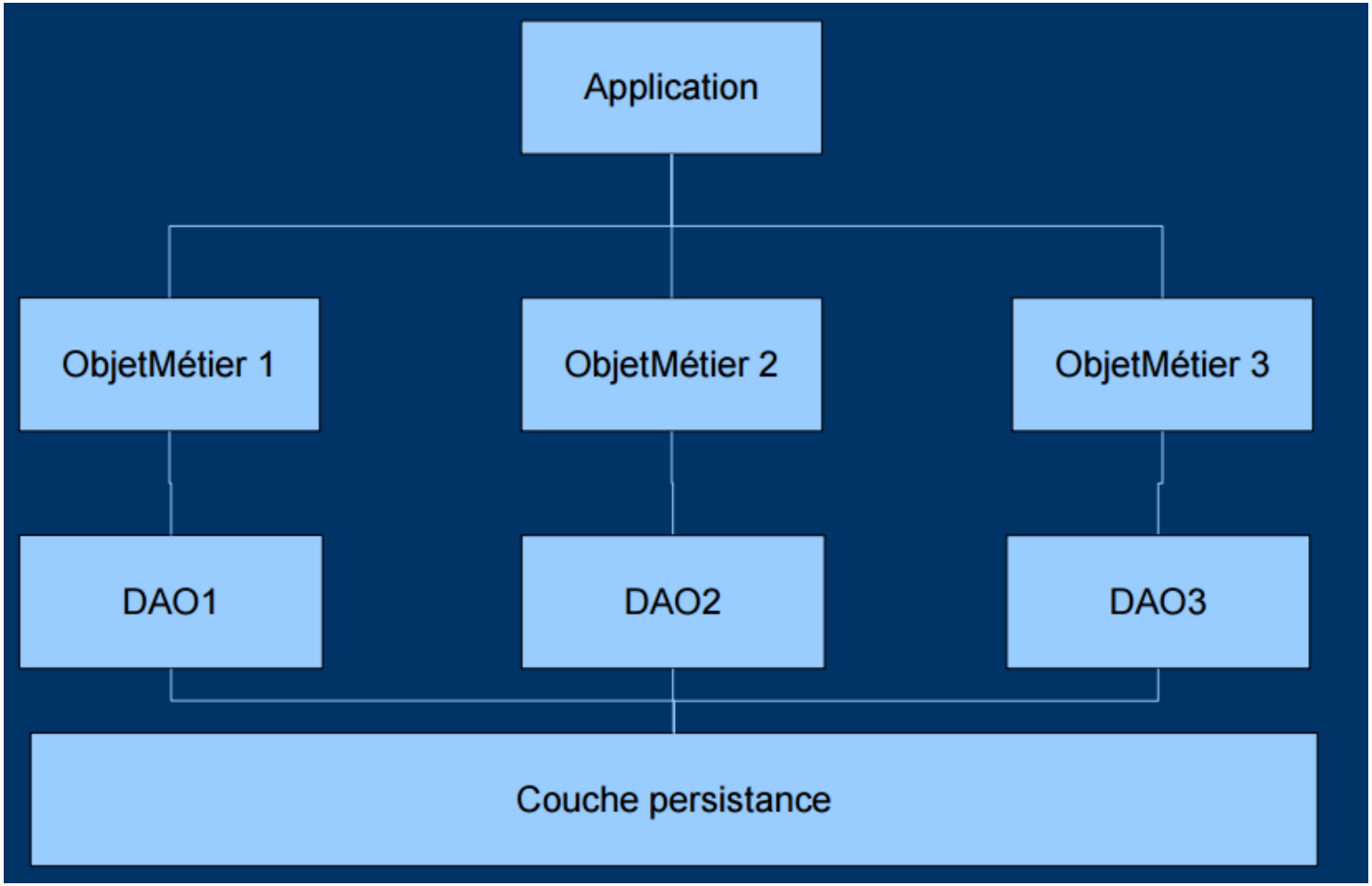
Résumé : DP de création

- 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 accès global à cette instance

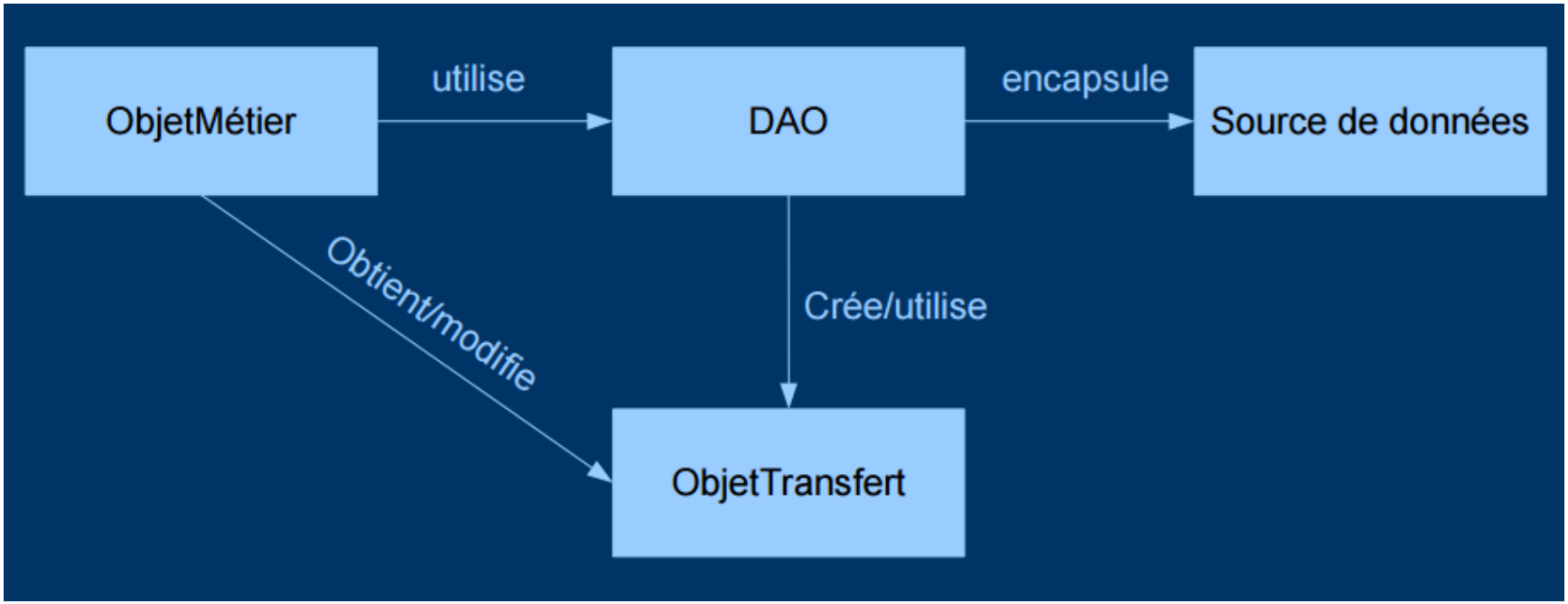
Le design pattern DAO Data Access Object

- Motivations
  - De nombreuses applications ont besoin de gérer des données persistantes
  - La persistance peut être géré par des moyens :
    - Différents selon la configuration de déploiement de l'appli
    - Évoluant dans le temps
- Principe
  - Isoler la couche de persistance du reste de l'appli
- Intérêt
  - L'essentiel de l'appli est indépendant de la persistance
- Référence
  http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

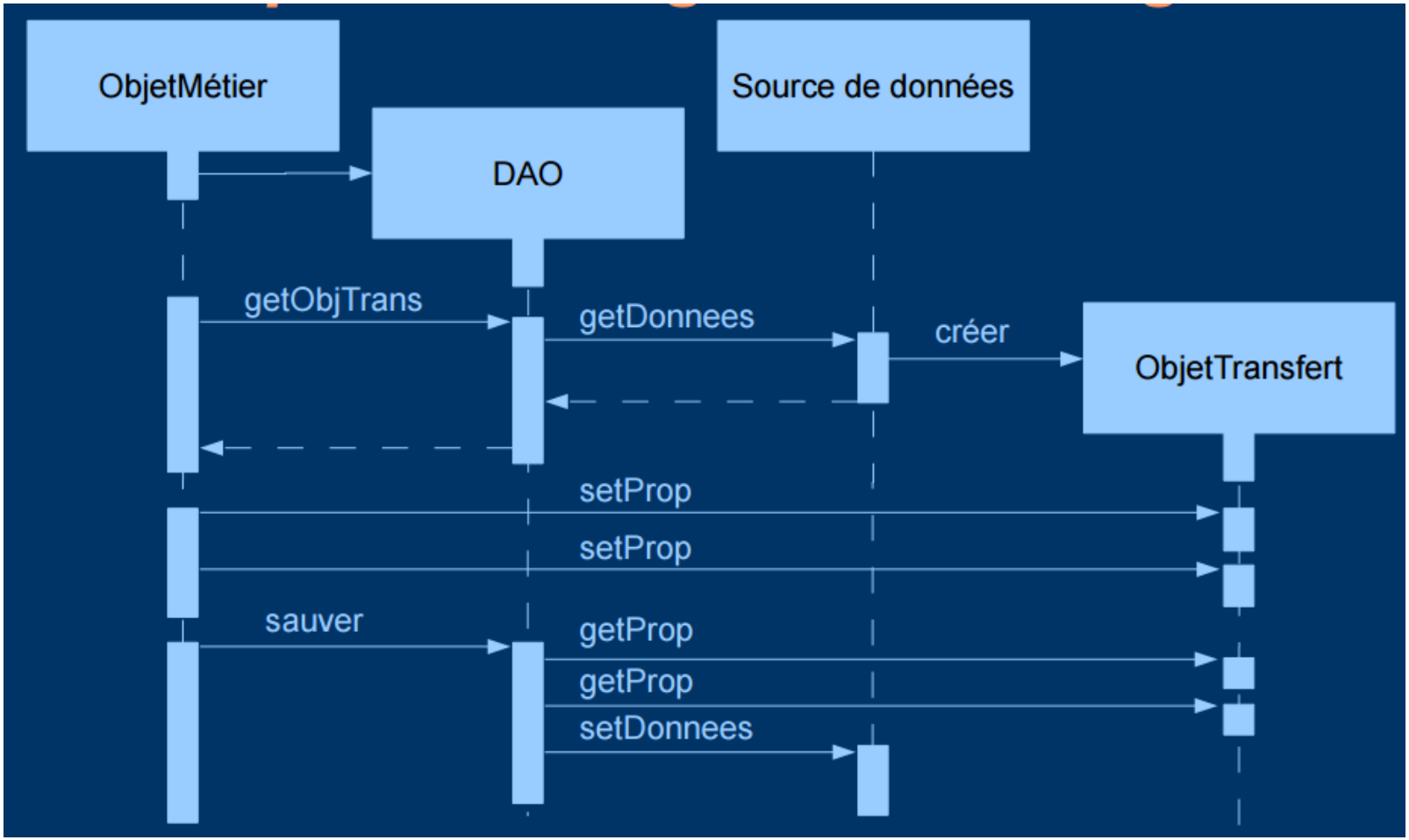
Principe général : Vue abstraite



Solution de base : Structure générale



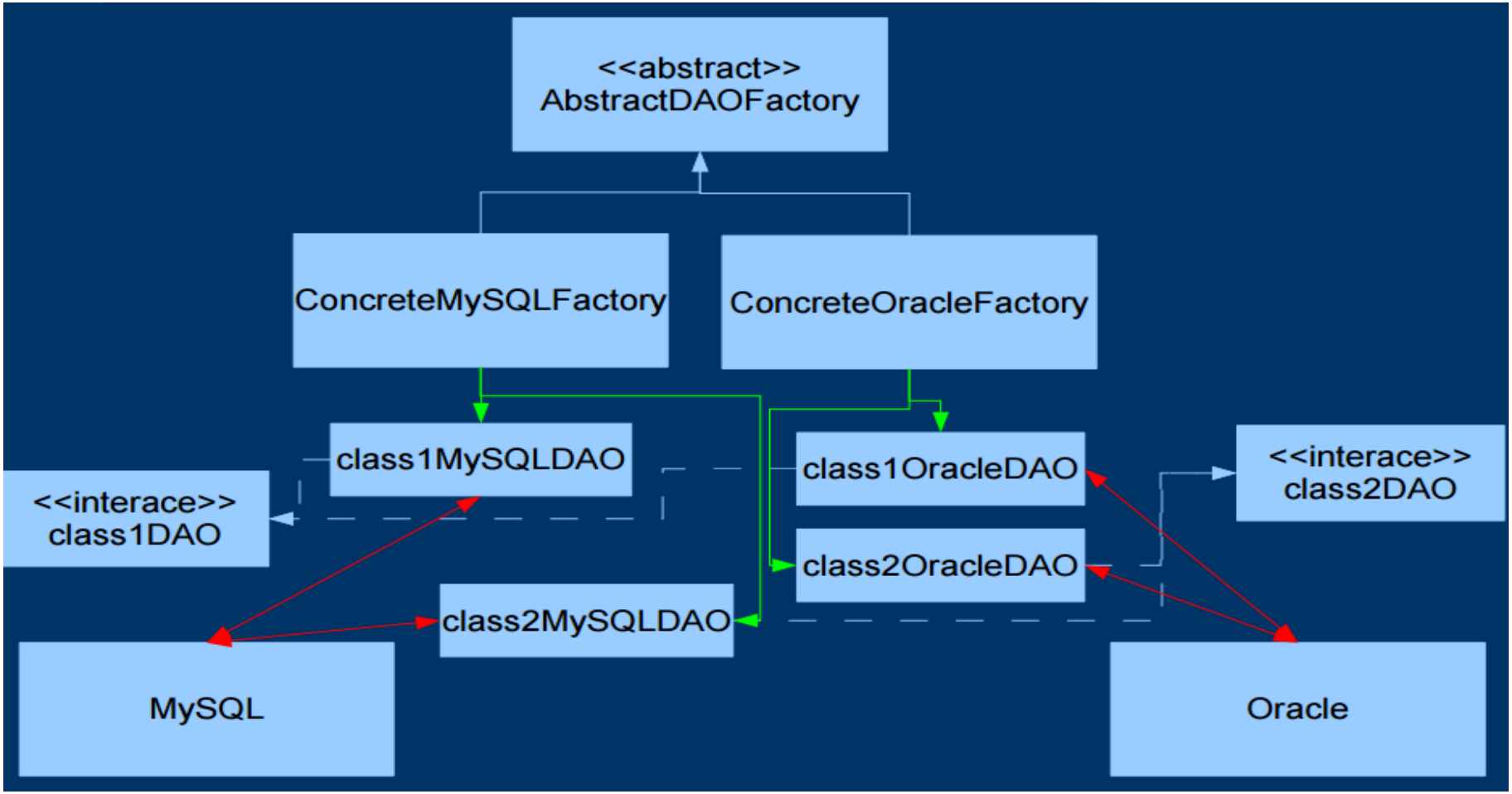
Solution de base : exemple d'échanges de message



Remarques générales sur le modèle

- Un objet DAO par classe métier (pas dit dans le document de référence) et non par objet
  - Objets DAO peuvent implanter le pattern Singleton
- Couche de persistance uniforme pour un projet
  - Objets DAO instanciés par famille
  - Utilisation possible du pattern Abstract Factory
- Objet Transfert pas indispensable

Exemple d'instanciation (1)



Exemple d'instanciation (2)

```
Public abstract class AbstractDAOFactory {
        public abstract Class1DAO getClass1DAO();
        public abstract Class21DAO getClass2DAO();
        public static getAbstractDAOFactory(int sgbd) {
                switch(sgbd) {
                        case Mysql:
                                return new ConcreteMySQLFactory();
                                break;
                        case Moracle:
                                return new ConcreteOracleFactory();
                                break;
                }
        }
}

public class ConcreteMySQLFactory extends AbstractDAODactory {
        public Class1DAO getClass1DAO() {
                return Class1MySQLDAO.getInstance();
        public Class2DAO  getClass2DAO() {
                return Class2MySQLDAO.getInstance();
}
```

Exemple d'instanciation (3)

```
Public interface  Class1DAO {
	public Class1 load(int id);
	public void save(Class1 obj);
}

public class Class1MySQLDAO implements Class1DAO {
	private static instance = null;
	private Class1MySQLDAO() {
		// connexion à la base par exemple
	}
	public static Class1DAO getInstance() {
		if (instance == null) {
			instance = new Class1MySQLDAO();
		}
		return instance;
	}
	public Class1 load(int id) {
		// requête d'accès à la base
		return new Class1(params);
	}
	public void save(Class1 obj) {
		// requête d'update/insert suivant les cas
	}
}
```

Etude de cas : 5 Problèmes

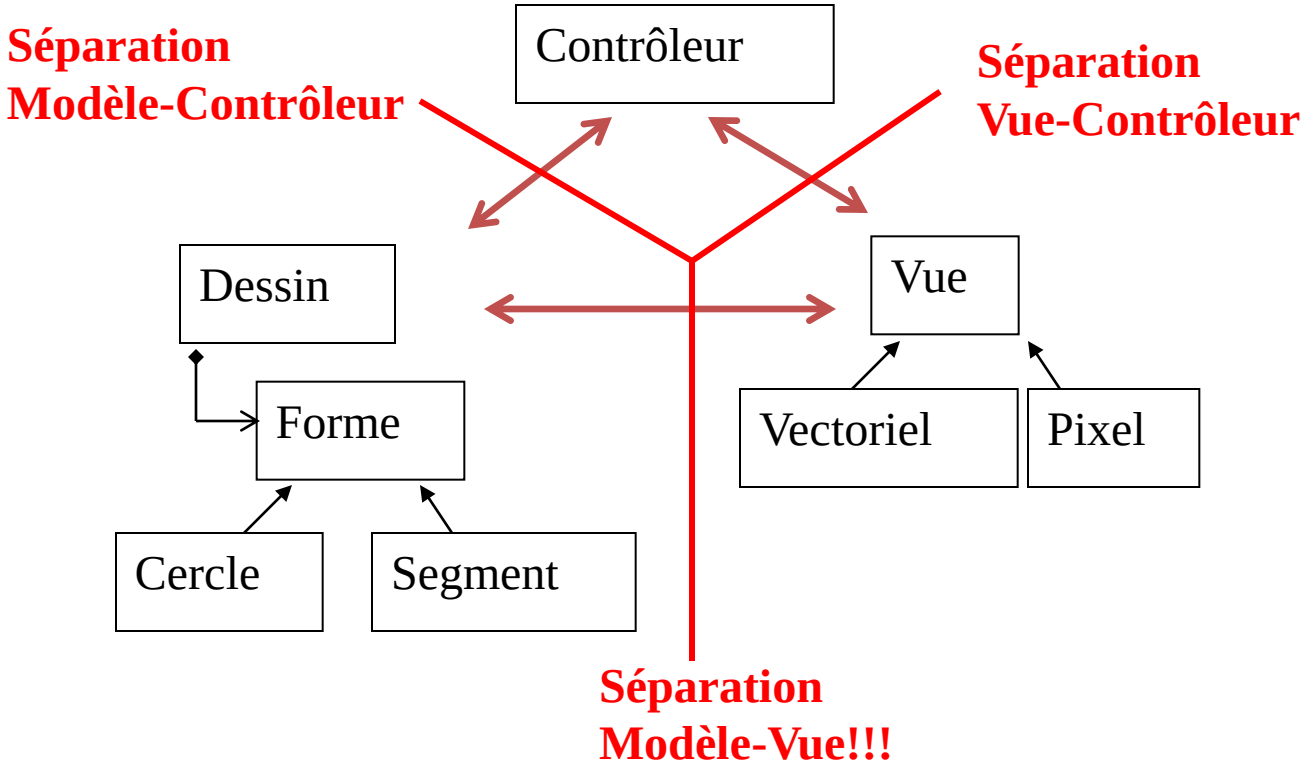
Concevoir l'architecture (classes en UML) d'un logiciel de dessin géométrique supportant les cercles, segments, groupes…

Parties à clarifier :

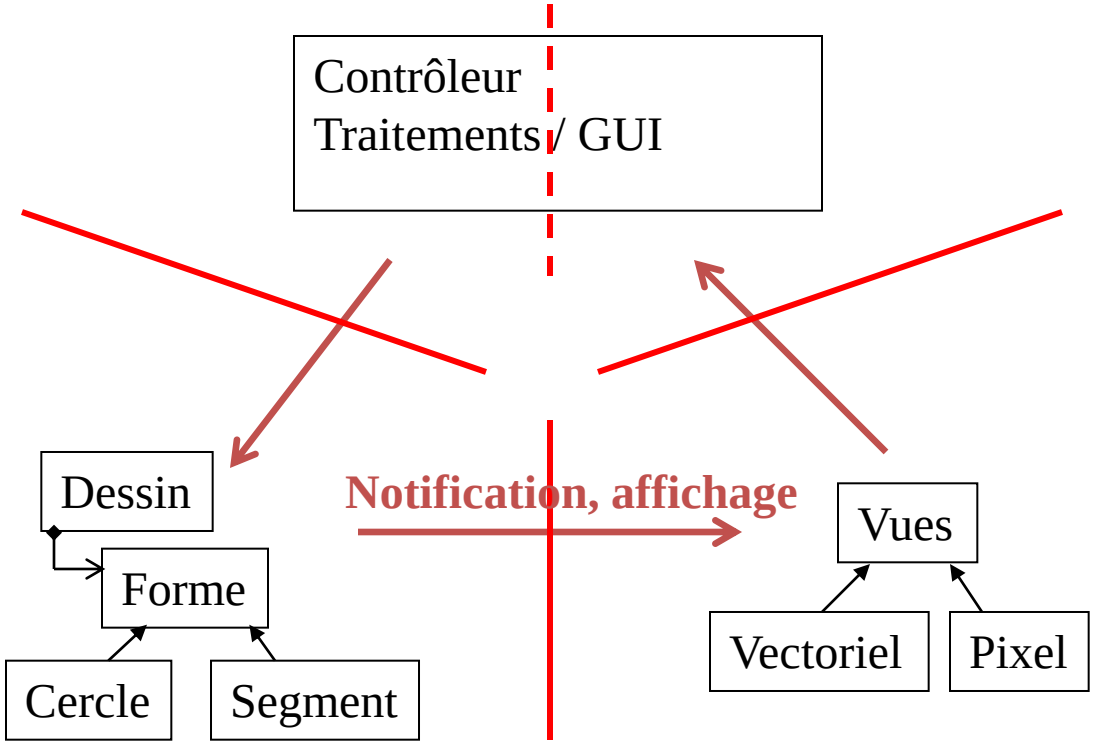
1. Structure interne / Dessins des formes
2. Changements synchronisés
3. Groupes d'objets (Group / Ungroup)
4. Comportements de la souris, des menus contextuels
5. Conversions en multiples formats…

Pb 1/5 : MVC
Modèle - Vue - Contrôleur

- Fichiers / Représentations Internes / Vues / Interactions utilisateurs

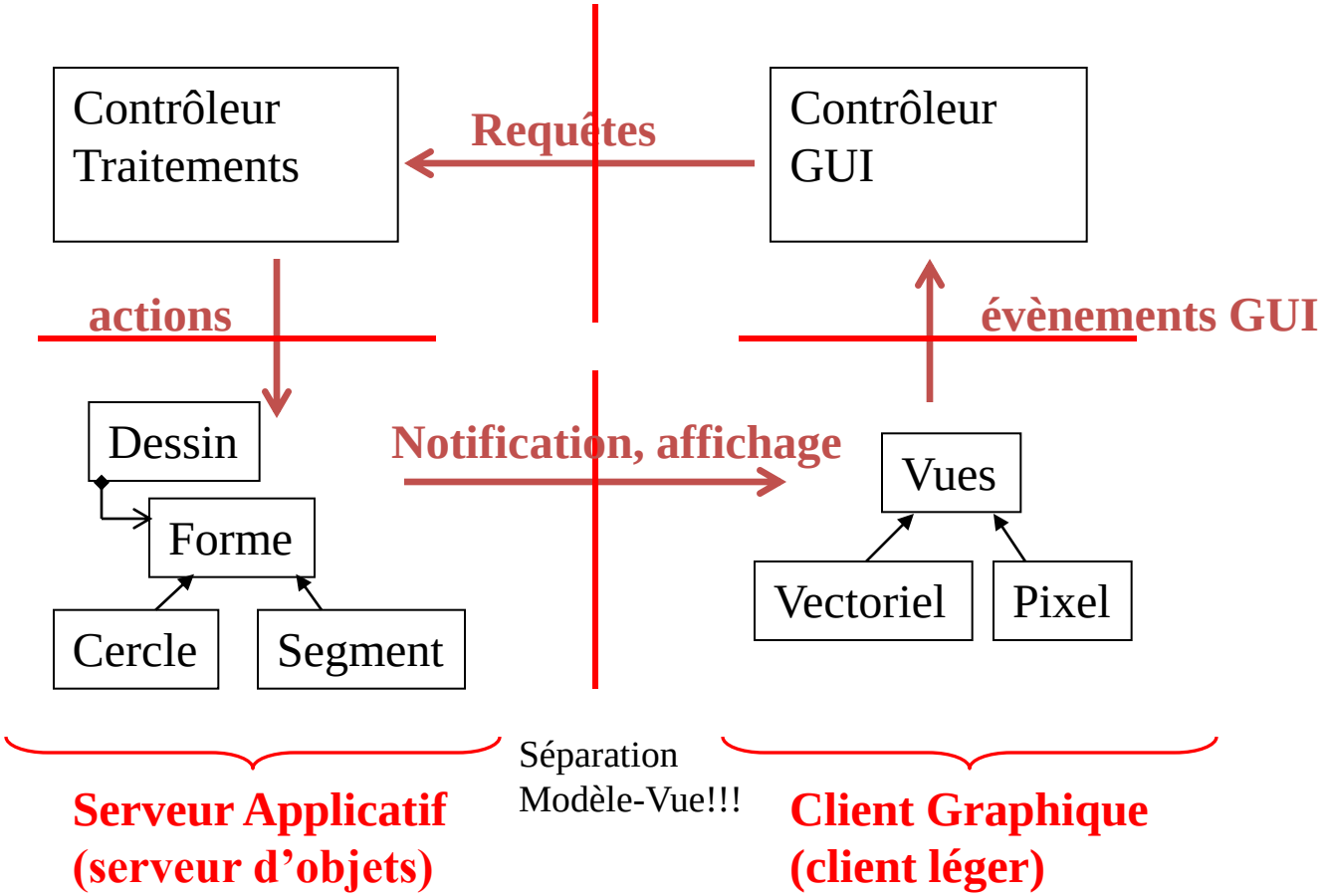


Pb1/5 : MVC (Suite) : Contrôleur Traitements / GUI

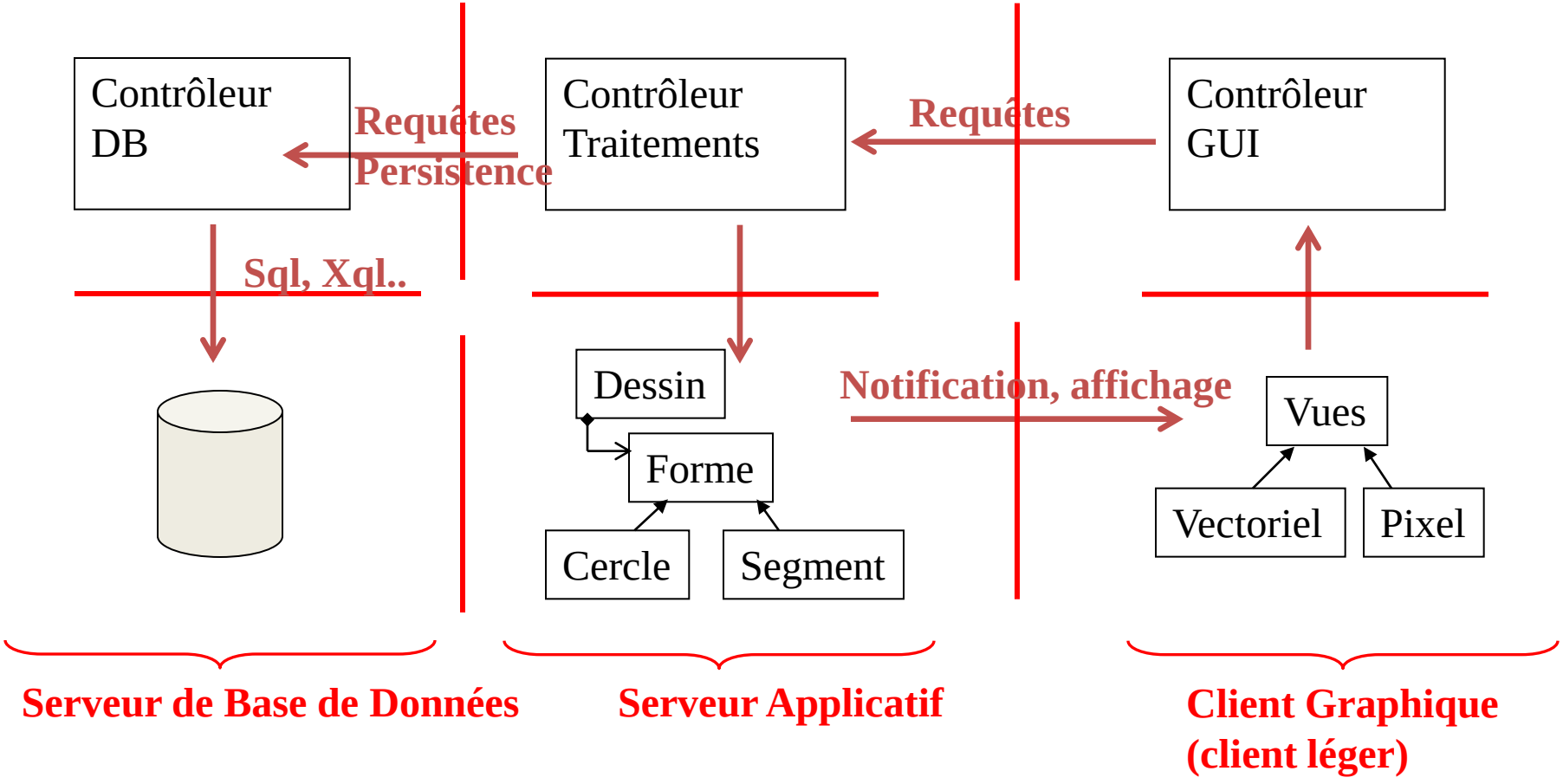


Séparation
Modèle-Vue!!!

Pb1/5 : MVC (Suite) Architecture 2 tiers

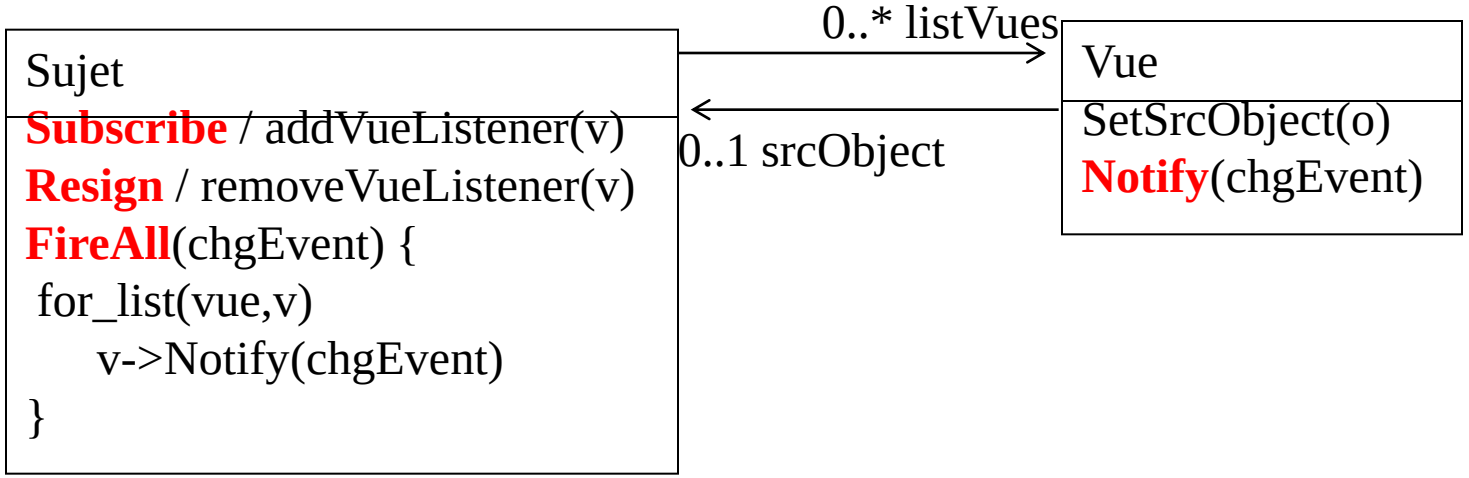


Pb1/5 : MVC (Suite) Architecture 3 tiers



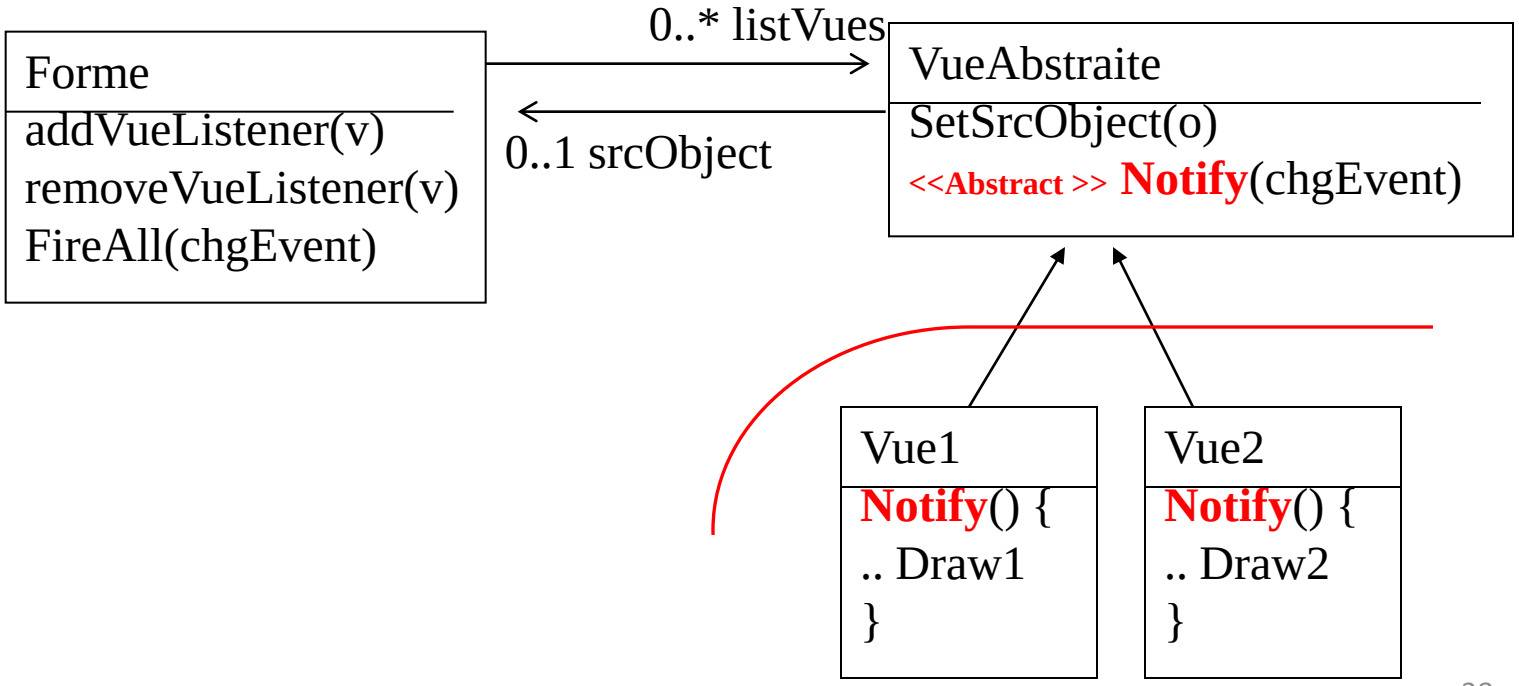
Pb2/5 : Publish & Subscribe

- Notifications de changement



Pb 2/5: Publish & Subscribe (Bis)

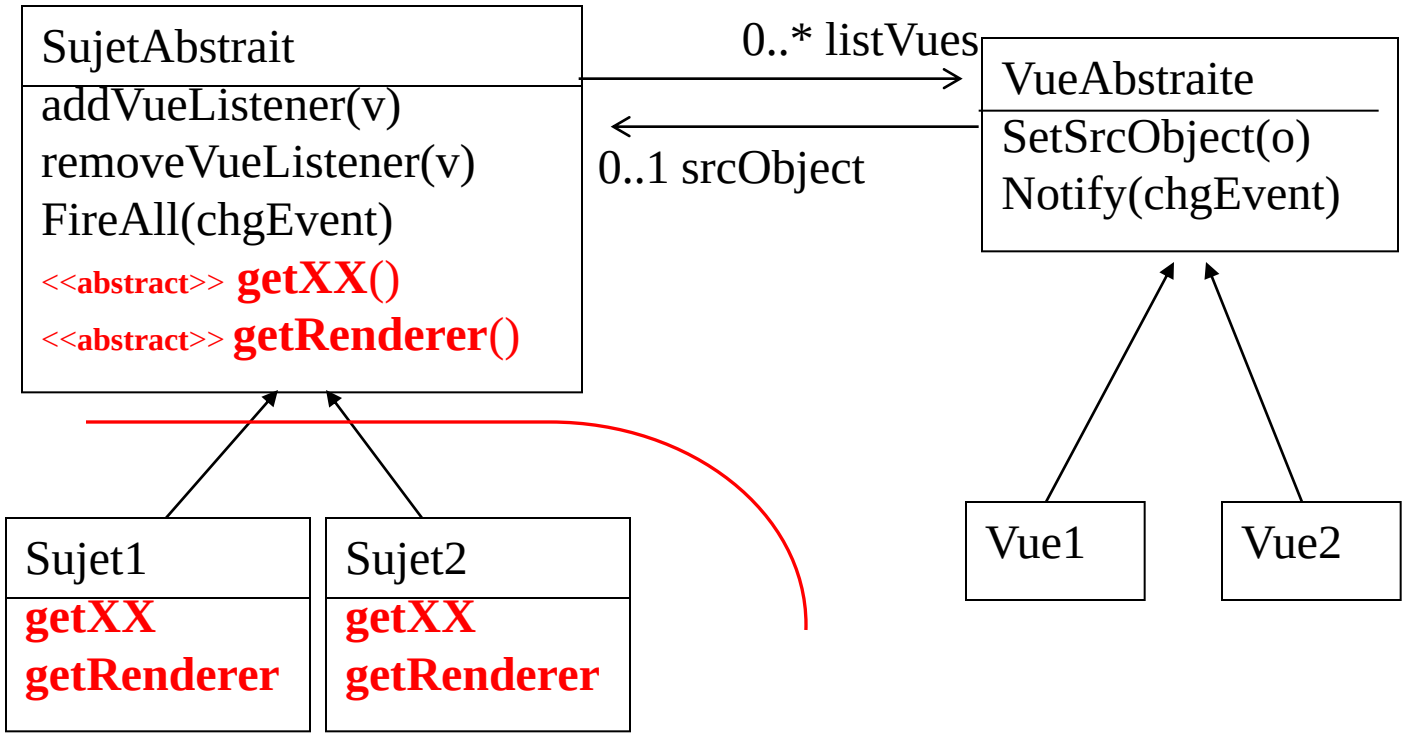
- Indépendance des Vues pour l'Objet



Pb2/5: Publish & Subscribe (Ter)

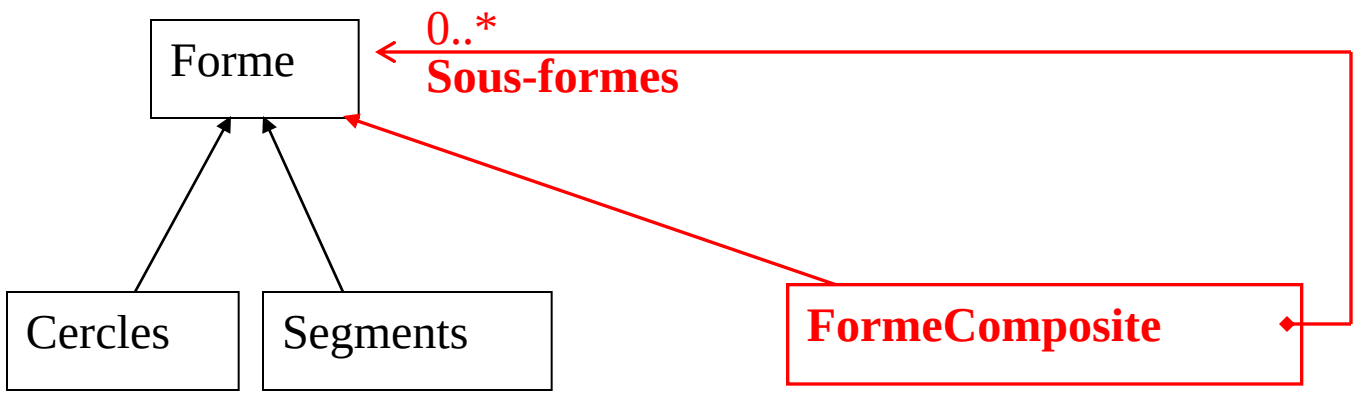
Indépendance des Objets pour les Vues

(cf. MVC)



Pb 3/5 : Composite..

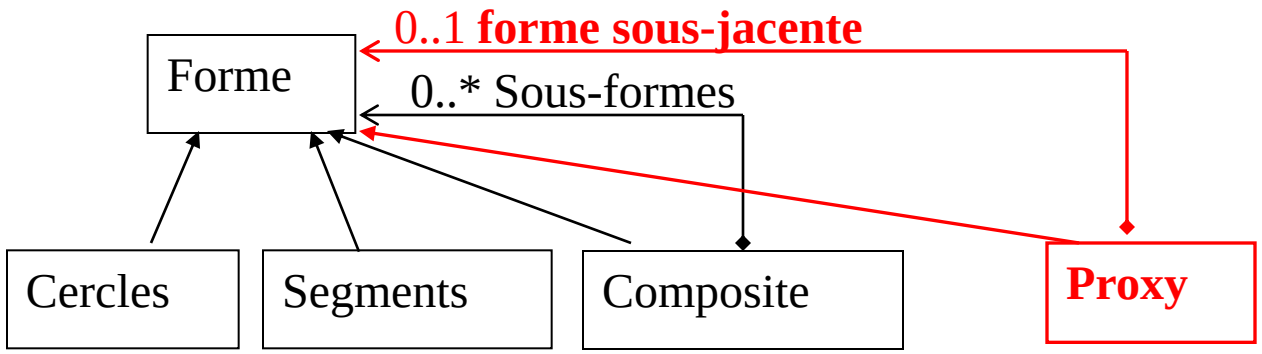
- Group / Ungroup



Pb3/5 : Composite, Proxy..

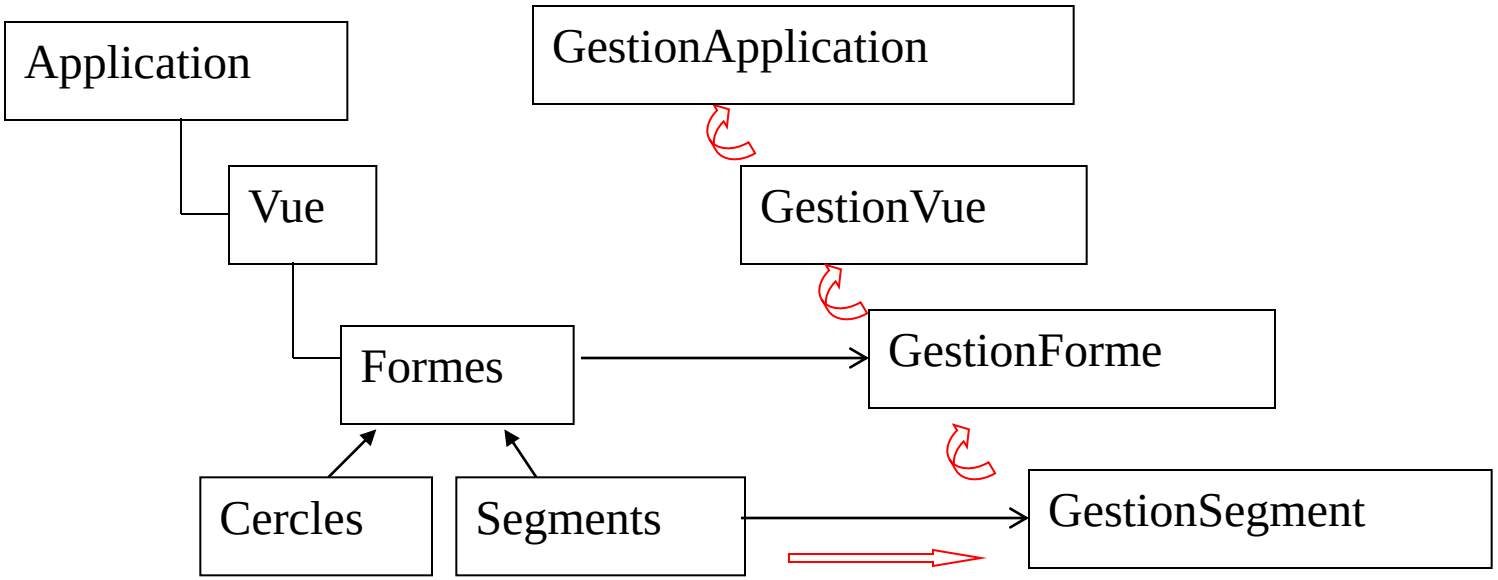
- Formes par procuration

(Rotation, Iconifiée, En cours de chargement, etc..)



Pb4/5 : Délégation, Chaîne de Responsabilité..

- Gestion de la souris, des évènements graphiques…



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

- Conversions Multiples, etc..