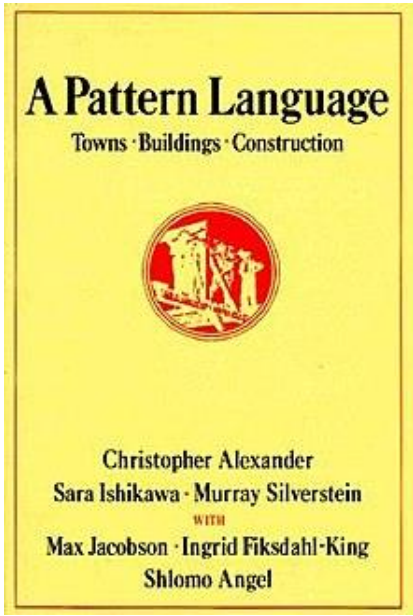
Design patterns
Patrons de conception

Stéphanie CHOLLET

Origines : en architecture

"The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

C. Alexander *et al.*, "A pattern language: Towns, Buildings, Construction", 1977



The book creates a new language, what the authors call a pattern language derived from timeless entities called patterns. As they write on page xxxv of the introduction, "All 253 patterns together form a language." Patterns describe a problem and then offer a solution. In doing so the authors intend to give ordinary people, not only professionals, a way to work with their neighbors to improve a town or neighborhood, design a house for themselves or work with colleagues to design an office, workshop or public building such as a school.

Définition

- Un patron est une solution générale pour un problème connu et récurrent dans un contexte donné

- Différents types de patterns en informatique :
  - Design patterns ou patrons de conception
  - Patrons architecturaux

Patterns = bonne pratique !

Gestion de projet

Organisation de l'application

Organisation du code

Génie Logiciel

- Diagramme de GANTT
- Cycle de vie
- Qualité du logiciel
- Analyse des risques
- Documentation :
  - Plan de développement
  - Cahier des charges
  - Dossier de spécifications
  - Dossier de conception
  - Plan de tests
  - …

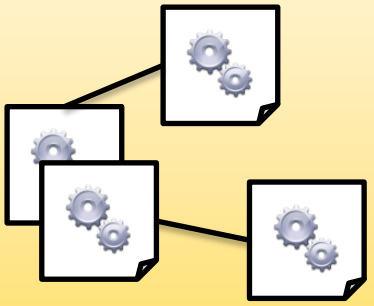
Architecture Logicielle

- Diagramme de contexte
- Architecture logique
- Architecture physique
- Modélisation UML
  - Diagramme de cas d'utilisation
  - Diagramme de collaboration
  - Diagramme de composants
- **Bonnes pratiques : Patterns architecturaux**

Modélisation

- Modélisation UML
  - Diagramme de classes
  - Diagramme d'objets
  - Diagramme de séquences
  - Diagramme de collaboration
  - Diagramme d'états
  - Diagramme d'activités
- **Bonnes pratiques : Design Patterns**

Code



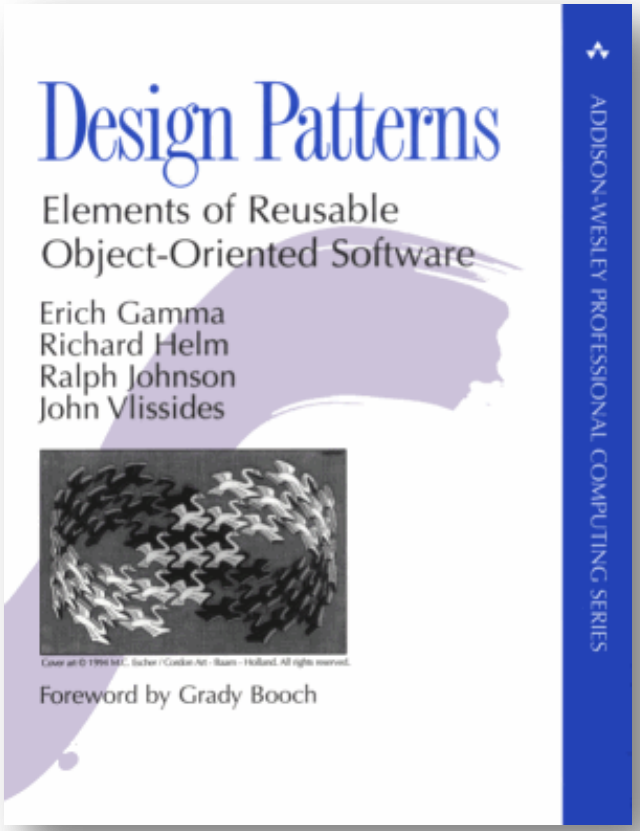
A quoi servent les *design patterns* ?

- Rendre disponible et explicite de bonnes pratiques de conception :
  - Capturer un savoir-faire, le rendre pérenne et réutilisable
- Nommer et rendre explicite une structure de haut niveau qui n'est pas directement exprimable sous forme de code
- Créer un vocabulaire commun pour les développeurs et les concepteurs

Design pattern en informatique : GoF

- GoF's contributions in design pattern
  - Gang of Four was a team of four members: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

(L-R) Ralph, Erich, Richard and John



23 design patterns

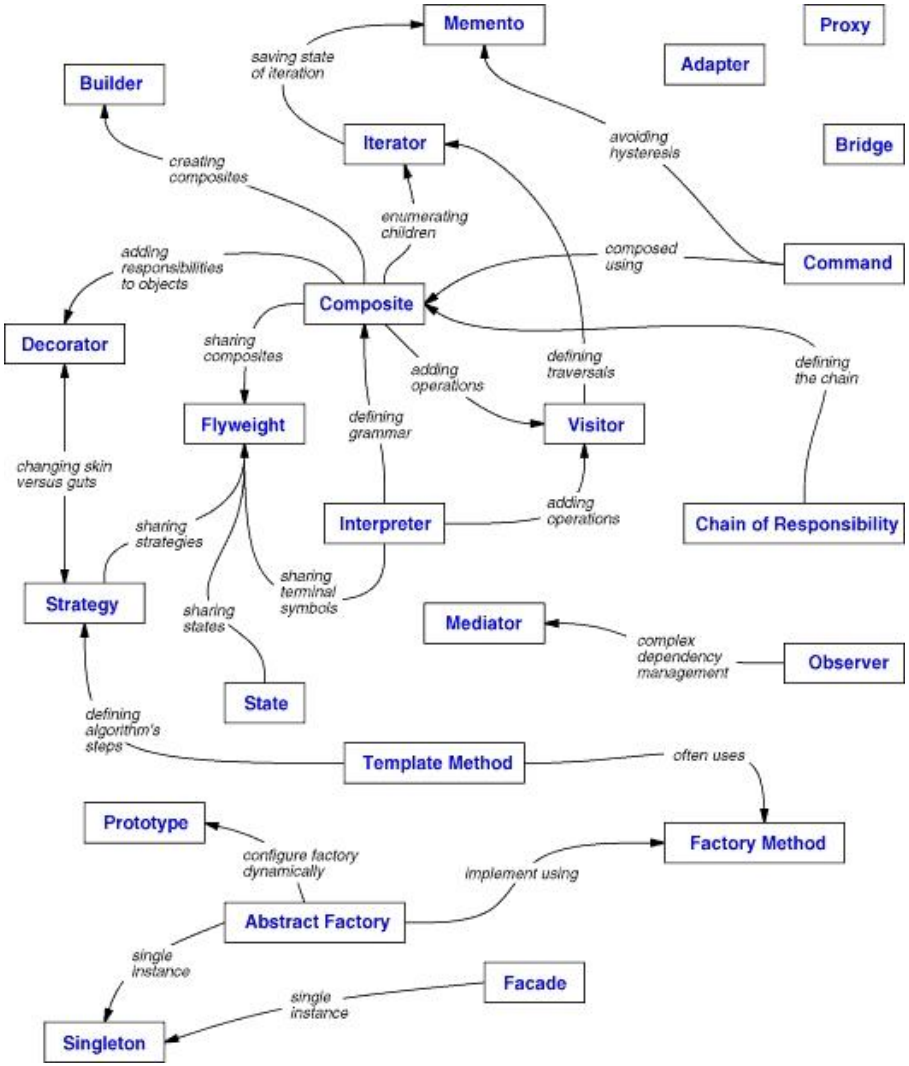
Documentation d'un pattern selon le GoF

- Nom et classification
- Intention/but
- Autres noms connus
- Motivation (scénario)
- Applicabilité
- Structure
- Participants (classes, objets…)
- Collaboration
- Conséquences
- Implantation
- Exemple de code
- Usages connus
- Patterns associés

Classification des *design patterns*

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsability • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Relations entre les *patterns*



Design Patterns Créateurs

Deux exemples : Abstract Factory et Singleton

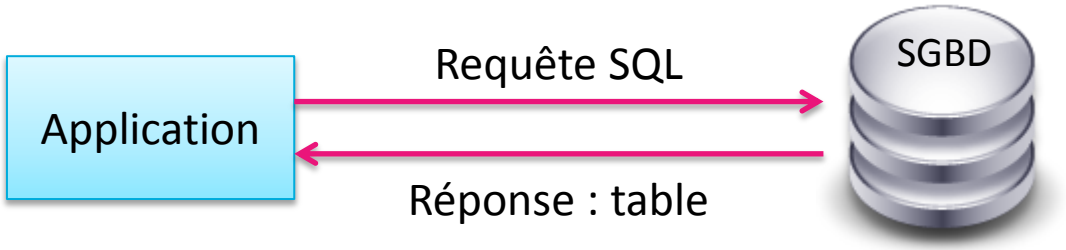
Objectif des patterns créateurs

- Abstraire le processus d'instanciation
- Rendre indépendant la façon dont les objets sont créés, composés, assemblés et représentés
- Encapsuler la connaissance de la classe concrète qui instancie
- Cacher ce qui est créé, qui créé, comment et quand

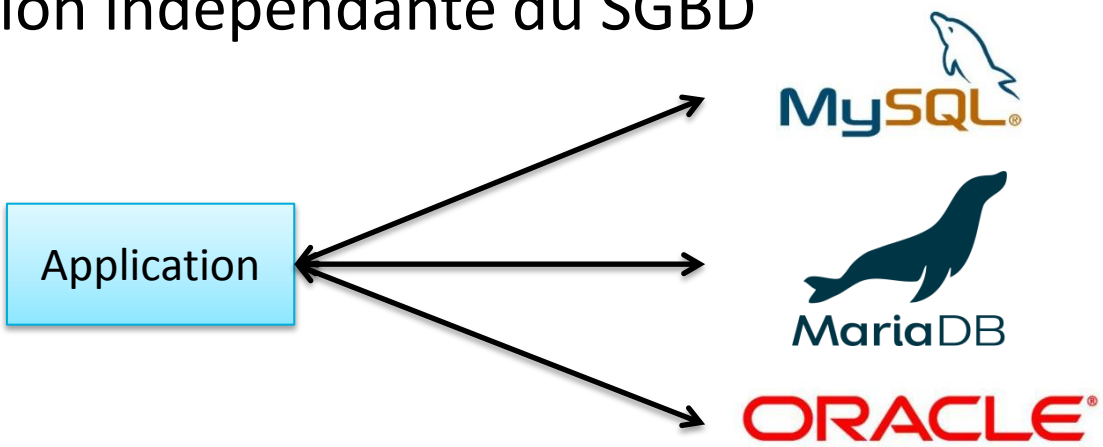
Cas d’utilisation : Utilisation applicative d’une base de données

Bases de données et application

- Modèle client/serveur :
  - Communication directe entre l'application et le SGBD



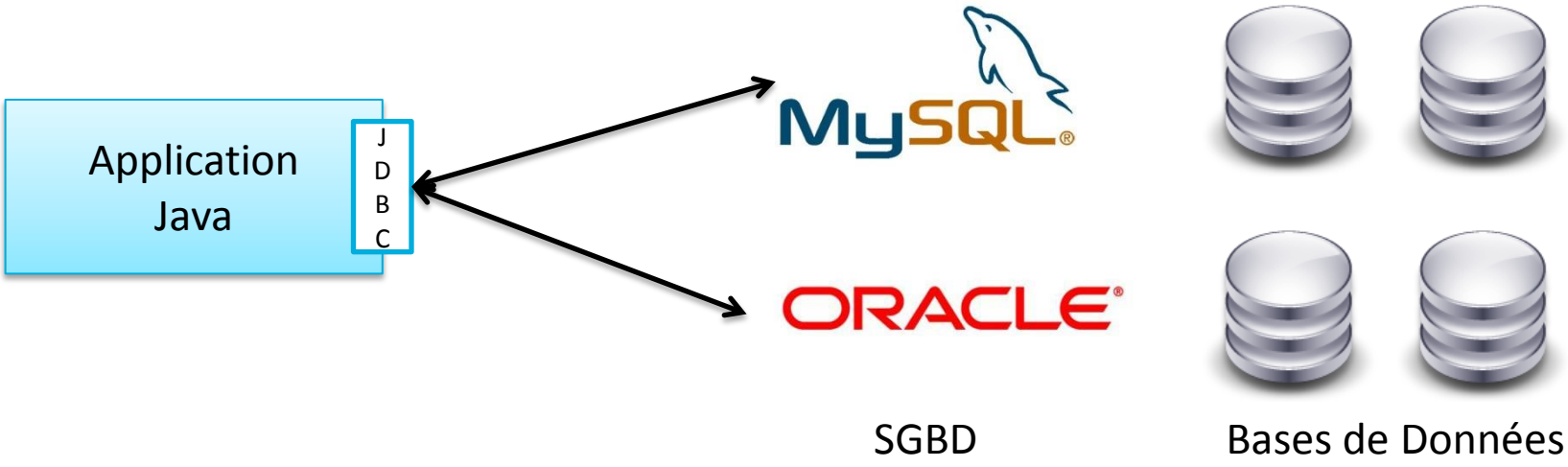
- Solution indépendante du SGBD



stephanie.chollet@grenoble-inp.fr

Interaction avec une application Java

- Driver JDBC (*Java DataBase Connectivity*)
  - Interface de programmation :
    - Ensemble de classes et d'interfaces permettant à des applications Java d'utiliser les services proposés par un ou plusieurs SGBD
      - Envoyer des requêtes
      - Récupérer une structure contenant le résultat d'une requête
    - API : java.sql.*



stephanie.chollet@grenoble-inp.fr

Utilisation du driver JDBC

- Algorithme :
  1. Initialiser le driver JDBC avec :
     - driver : le nom de la classe appropriée au SGBD
     - url : l'adresse du SGBD
     - login et mdp : nom de l'utilisateur et mot de passe
  2. Ouvrir une connexion
  3. Créer des requêtes SQL
     - Sélection, mise à jour, suppression
  4. Exécuter les requêtes
  5. Fermer la connexion

stephanie.chollet@grenoble-inp.fr

Problème classique – 1/2

- Comment faire correspondre les deux modèles (relations et objets) ?

Base de données relationnelle

Application Java

?

Relations (tables)
Tuples

Objets
Instances

stephanie.chollet@grenoble-inp.fr

Problème classique – 2/2

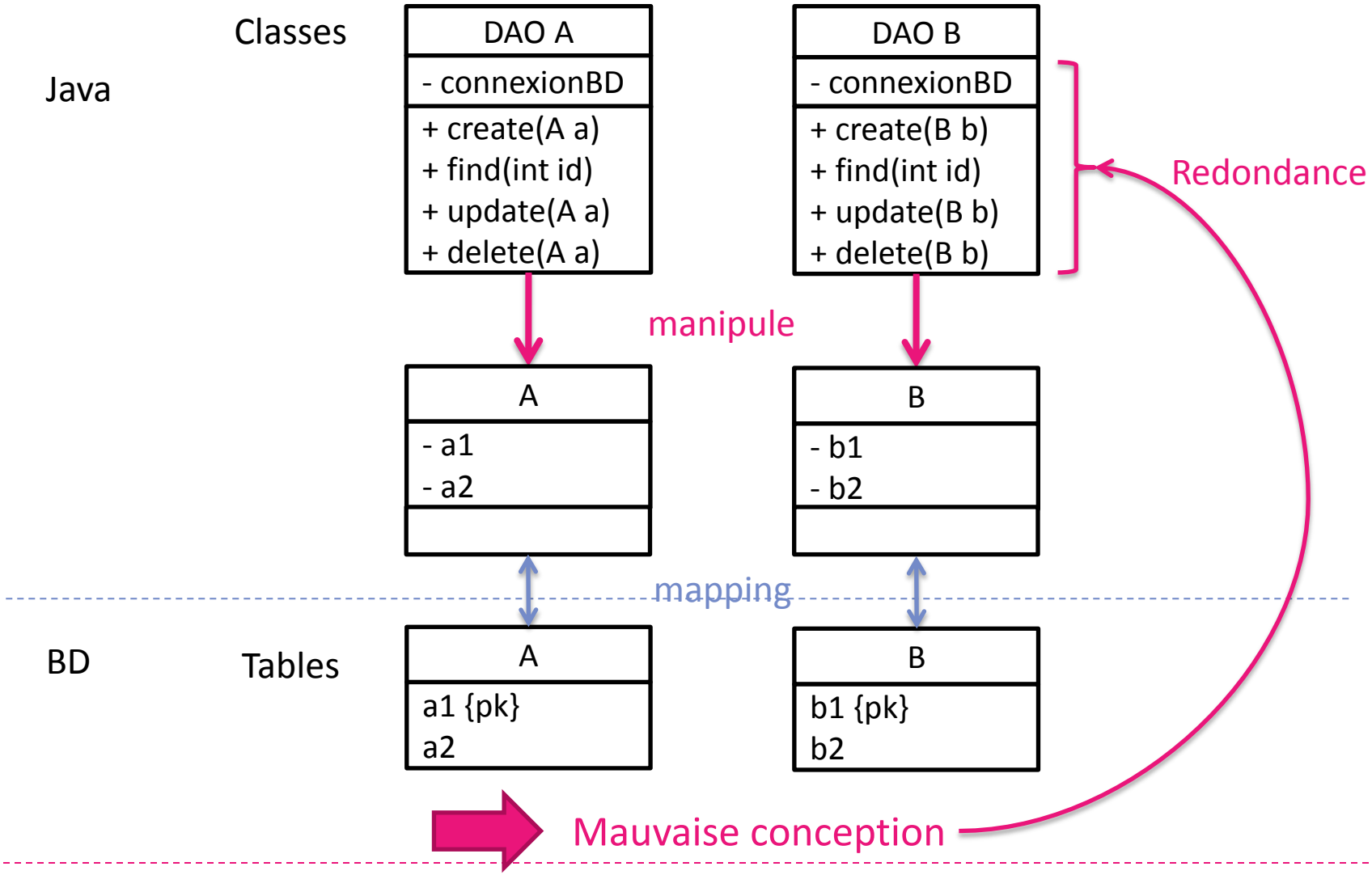
- Quelles sont les opérations qui sont réalisées sur la base de données ?
  - **C**reate : insertion de tuples
  - **R**ead : recherche de tuples selon certains critères
  - **U**pdate : mise à jour de tuples
  - **D**elete : suppression de tuples

Utilisation du design pattern DAO (*Data Access Object*) pour manipuler la base de données

Le DAO est un patron architectural et non un patron venant des design patterns du GoF

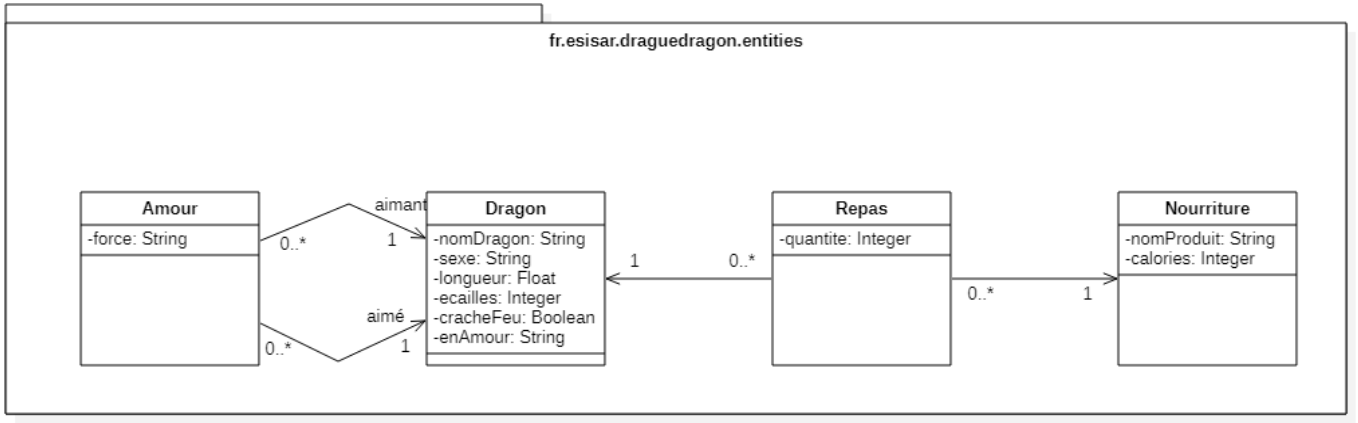
stephanie.chollet@grenoble-inp.fr

Solution 1



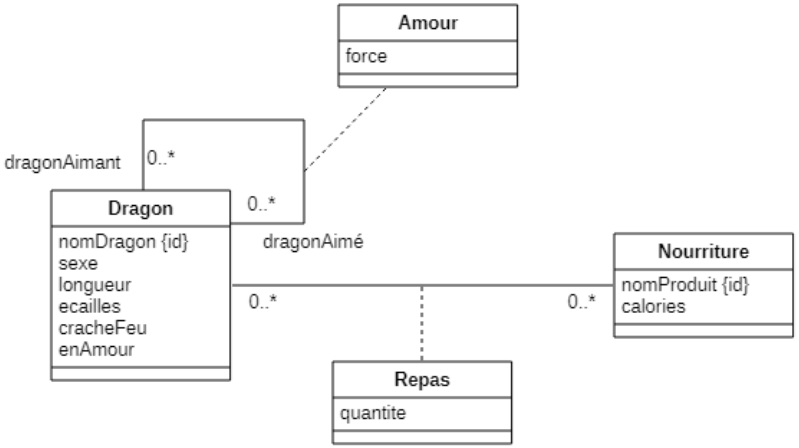
stephanie.chollet@grenoble-inp.fr

Exemple : Modèle de données



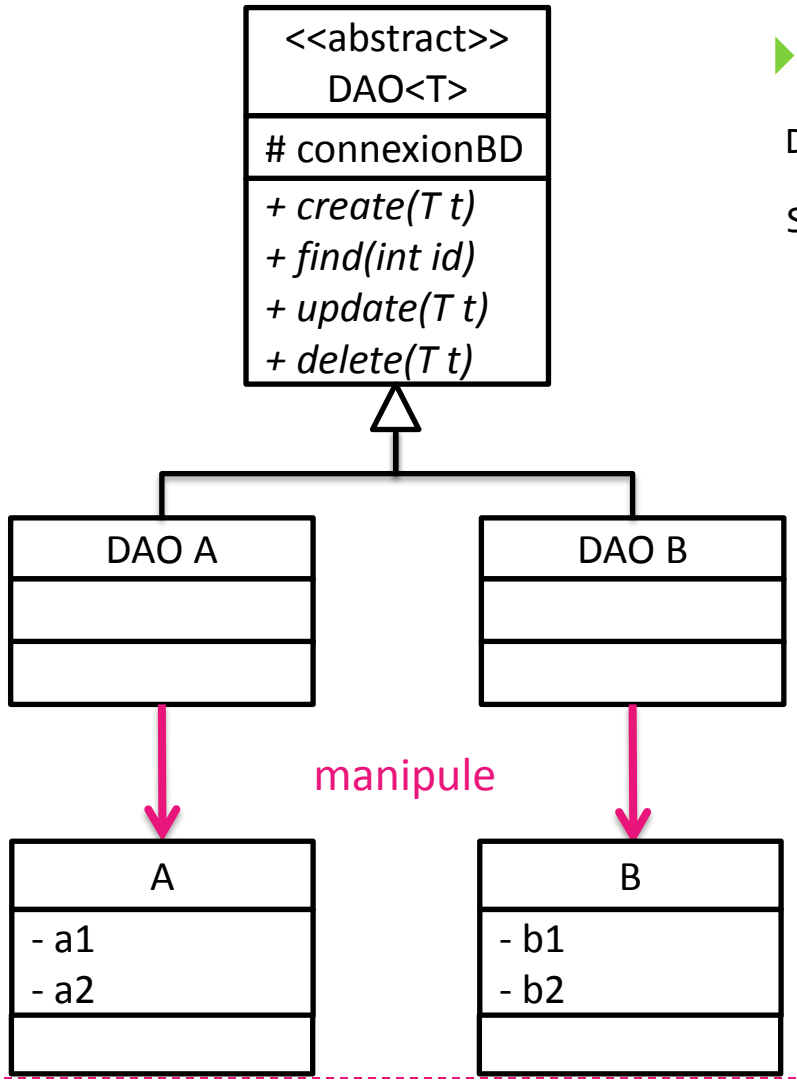
Modèle de données

Schéma conceptuel de la base de données



Solution 2 : Généralisation avec des génériques

Java



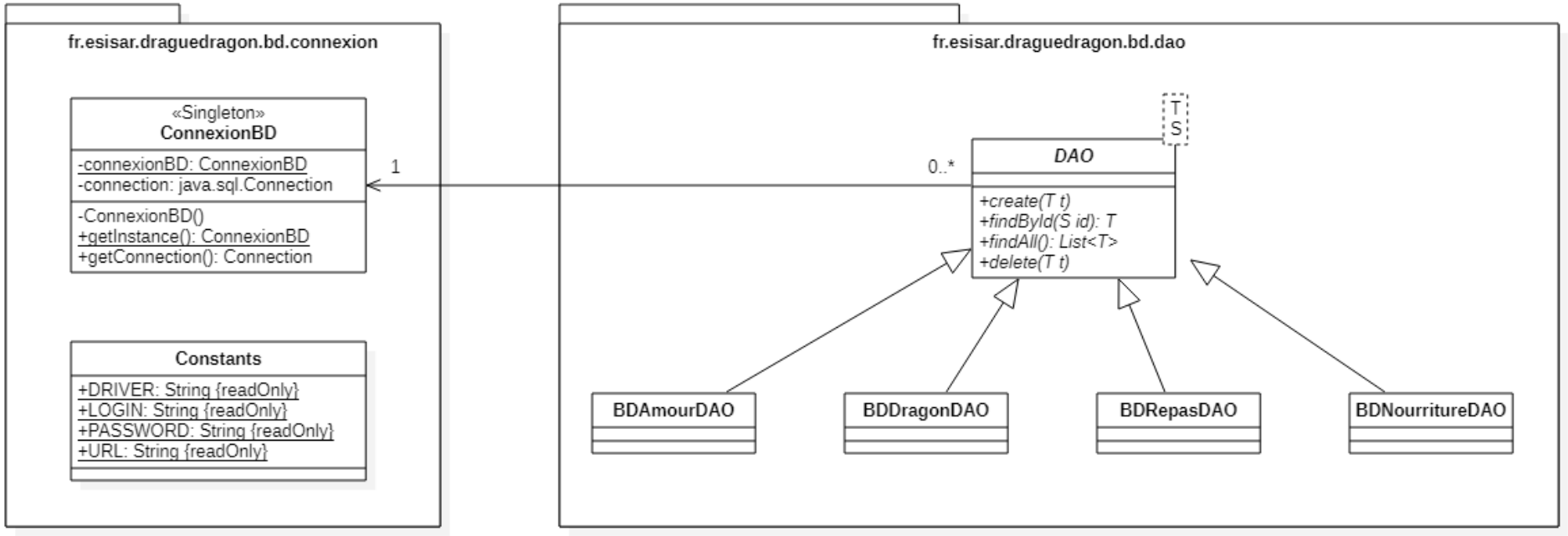
- Exemple :

```
DAO<Etudiant> etudiantDAO = new EtudiantDAO();

System.out.println(etudiantDAO.find(12345));
```

Bien mais on peut faire mieux !

stephanie.chollet@grenoble-inp.fr

Exemple : Généralisation du DAO



Solution 3 : Encapsulation de l'instanciation

- Encapsulation de l'instanciation des objets dans une classe
  - Evite les modifications sur la façon de créer des objets concrets

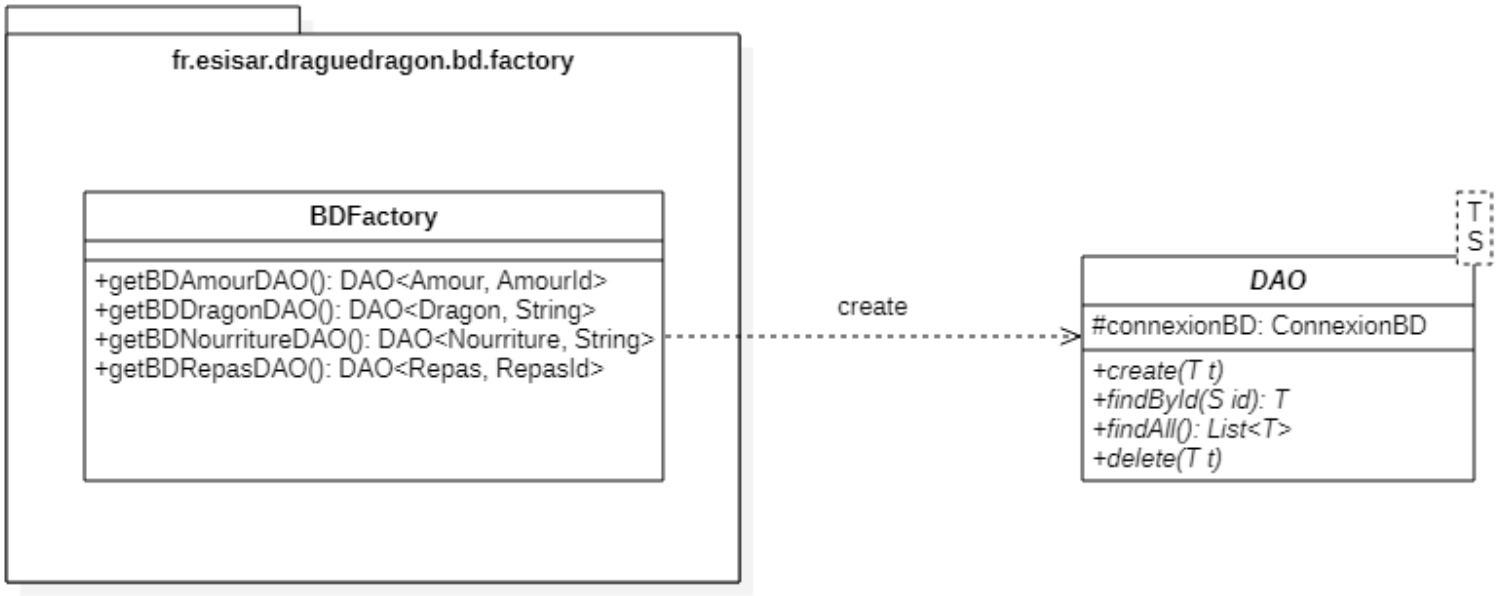

```
DAO<Etudiant> etudiantDAO = DAOFactory.getEtudiantDAO();

System.out.println(etudiantDAO.find(12345));
```

Bien mais on peut faire mieux !

Avec le pattern Factory

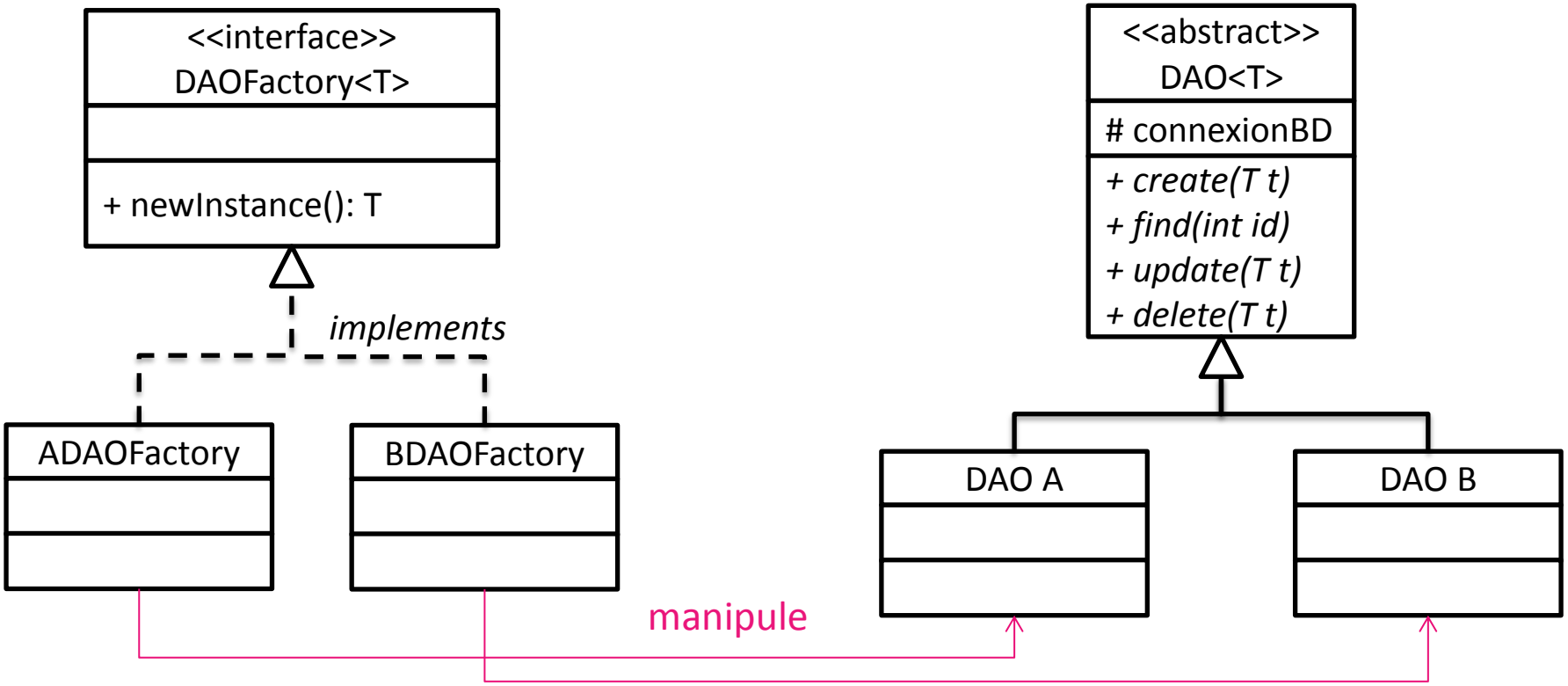
stephanie.chollet@grenoble-inp.fr

Exemple : Encapsulation de l'instanciation



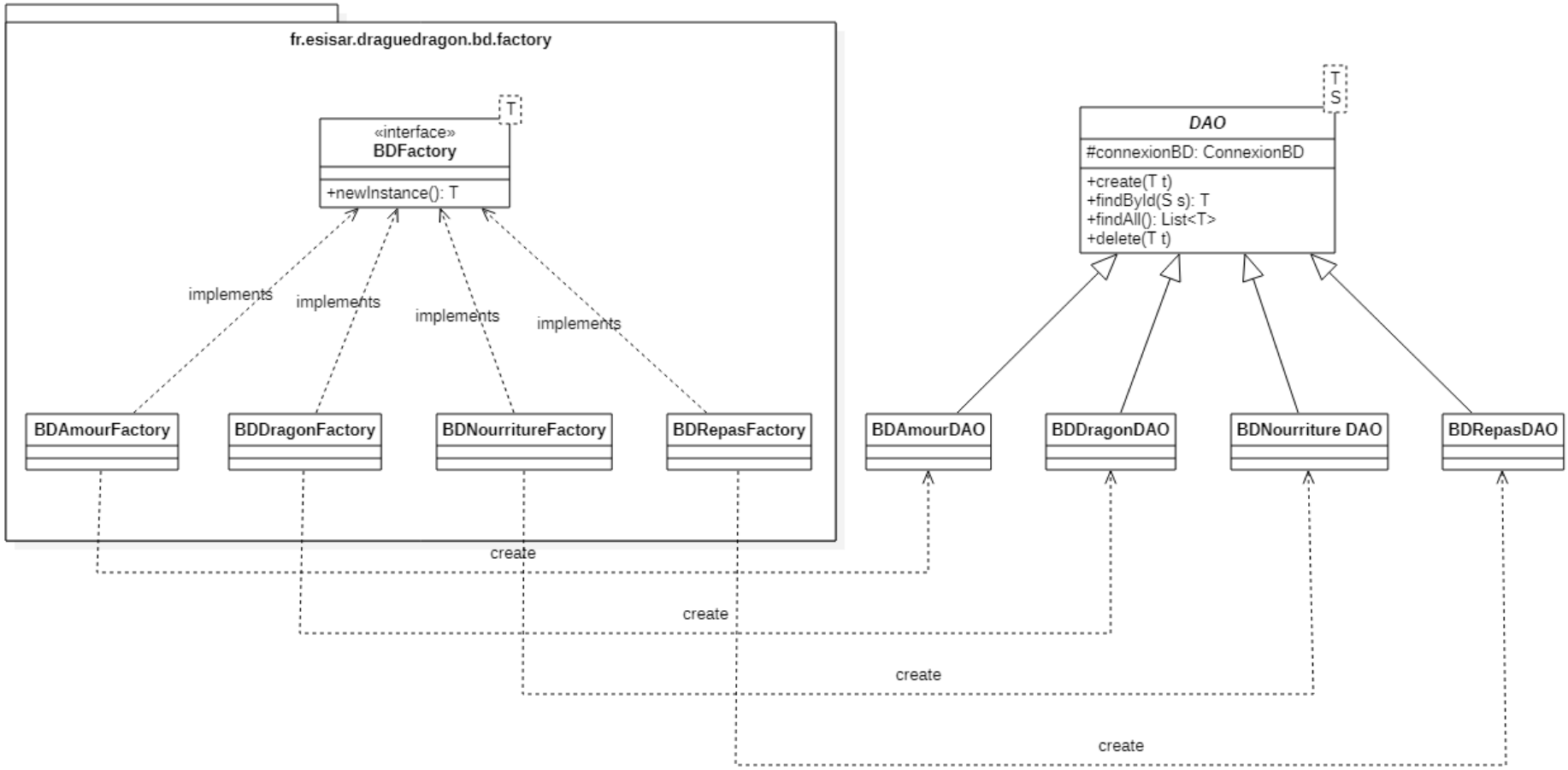
Solution 4 : Design Pattern Factory

- Définir une interface pour la création d'objet, en laissant aux classes concrètes le choix de l'instanciation



stephanie.chollet@grenoble-inp.fr

Exemple : Factory



(Solution 5)

- Si on veut avoir plusieurs sources de données avec des formats différents :
  - Base de données relationnelles
  - Fichier XML, JSON…

```xml
<etudiants>
   <etudiant>
      <nom>Dupond</nom>
      <prenom>Pierre</prenom>
   </etudiant>
   <etudiant>
      <nom>Durand</nom>
      <prenom>Paul</prenom>
   </etudiant>
</etudiants>
```

```json
"etudiants": [
    {
        "nom": "Dupond",
        "prenom": "Pierre"
   },
   {
        "nom": "Durand",
        "prenom": "Paul"
   }
]
```

Bien mais on peut faire mieux !

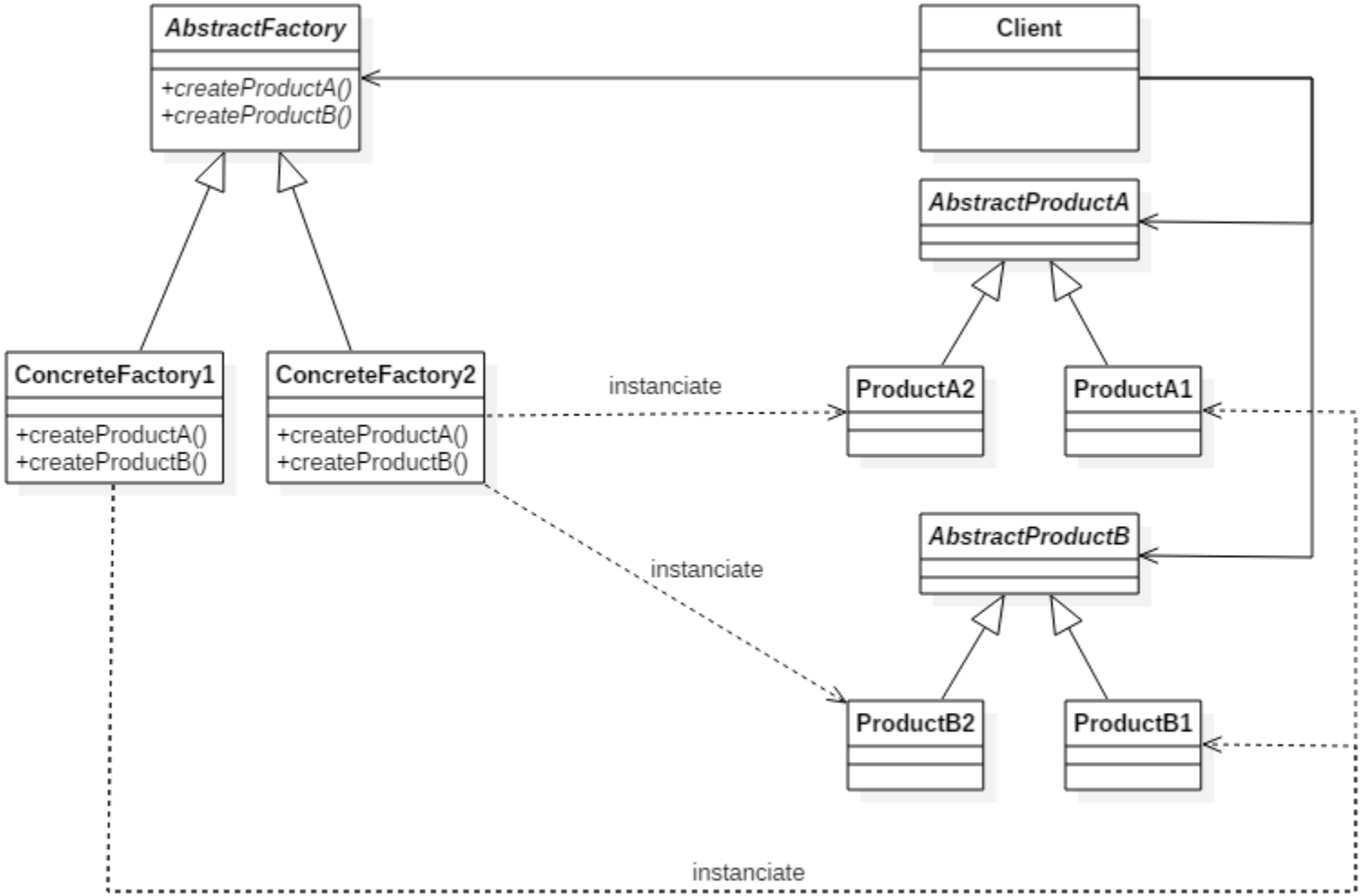
Avec le pattern AbstractFactory

stephanie.chollet@grenoble-inp.fr

Exemple 1 : Abstract Factory

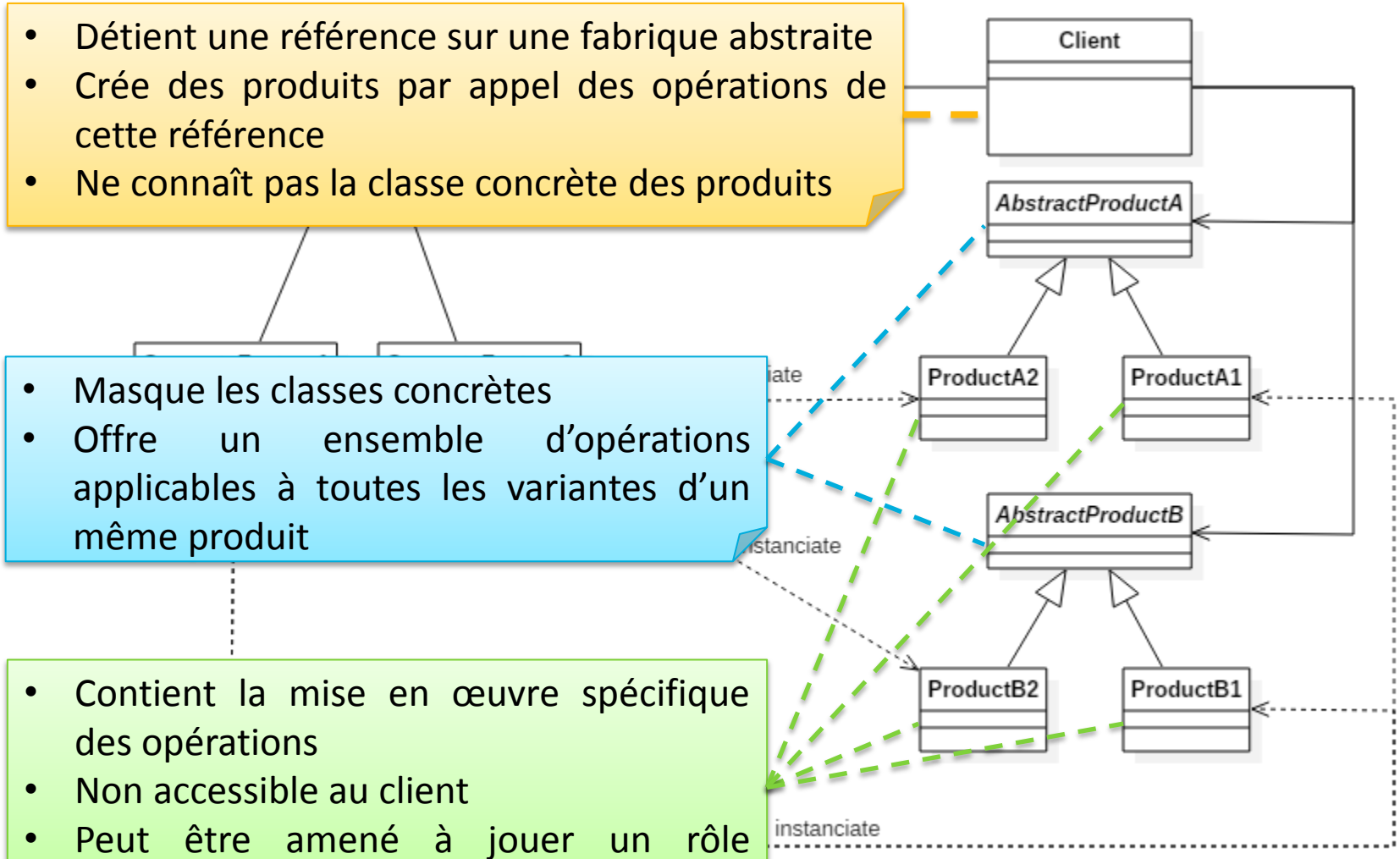
- But :
  - Fournir une interface pour créer des familles d'objets liés ou interdépendants sans avoir à préciser au moment de leur création la classe concrète à utiliser
- L'utilisation de ce motif est pertinente lorsque :
  - Le système doit être indépendant de la création des objets qu'il utilise
  - Le système doit être capable de créer des objets d'une même famille
- Avantages :
  - Isoler la création des objets retournés par la fabrique.
  - Faciliter le remplacement d'une fabrique par une autre selon les besoins.
- Exemple :
  - Création d'une interface homme-machine indépendante de la plate-forme

Exemple 1 : Abstract Factory
Structure générale



Exemple 1 : Abstract Factory
Structure générale

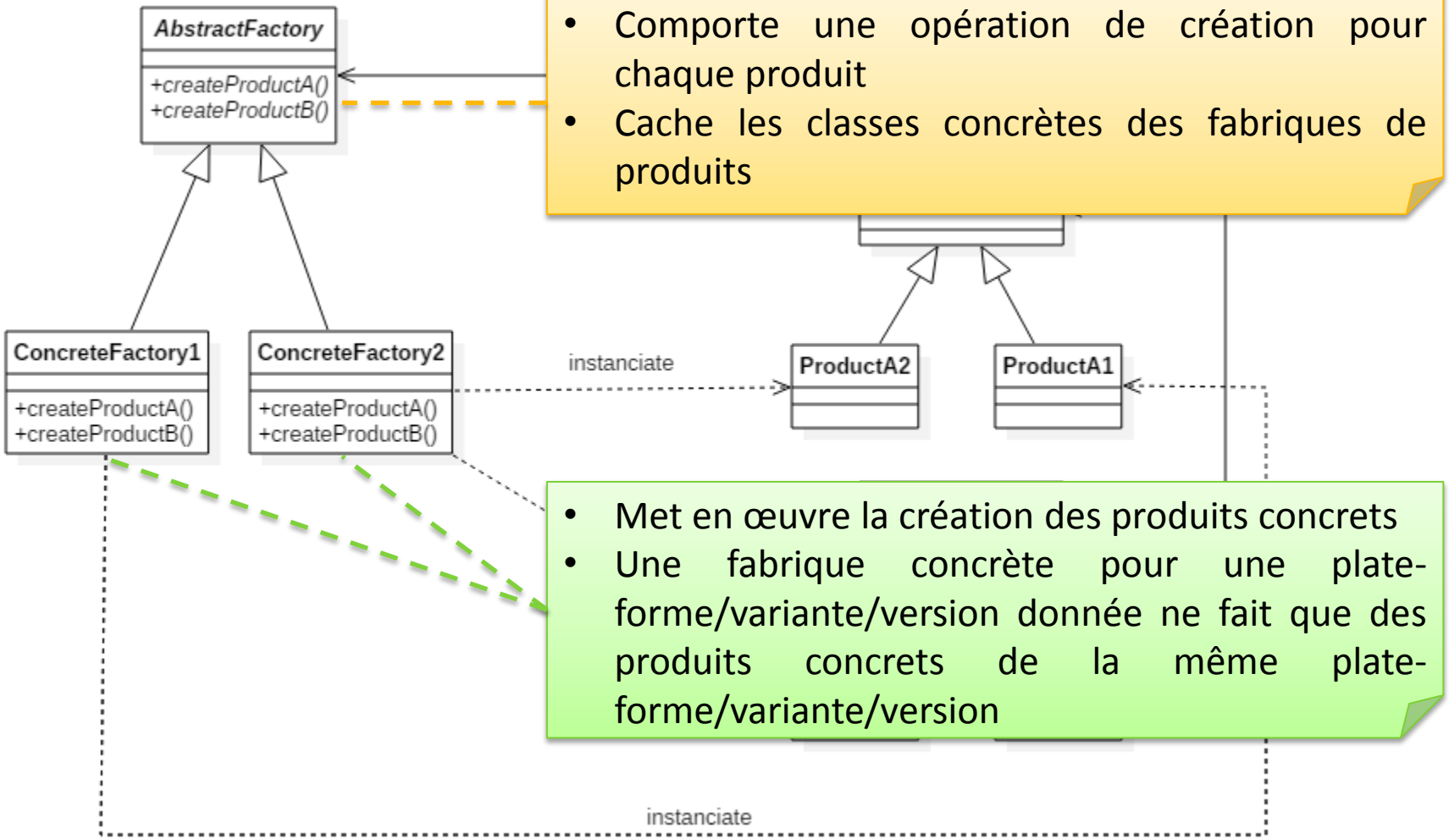
- Détient une référence sur une fabrique abstraite
- Crée des produits par appel des opérations de cette référence
- Ne connaît pas la classe concrète des produits



- Masque les classes concrètes
- Offre un ensemble d'opérations applicables à toutes les variantes d'un même produit

- Contient la mise en œuvre spécifique des opérations
- Non accessible au client
- Peut être amené à jouer un rôle d'adaptateur

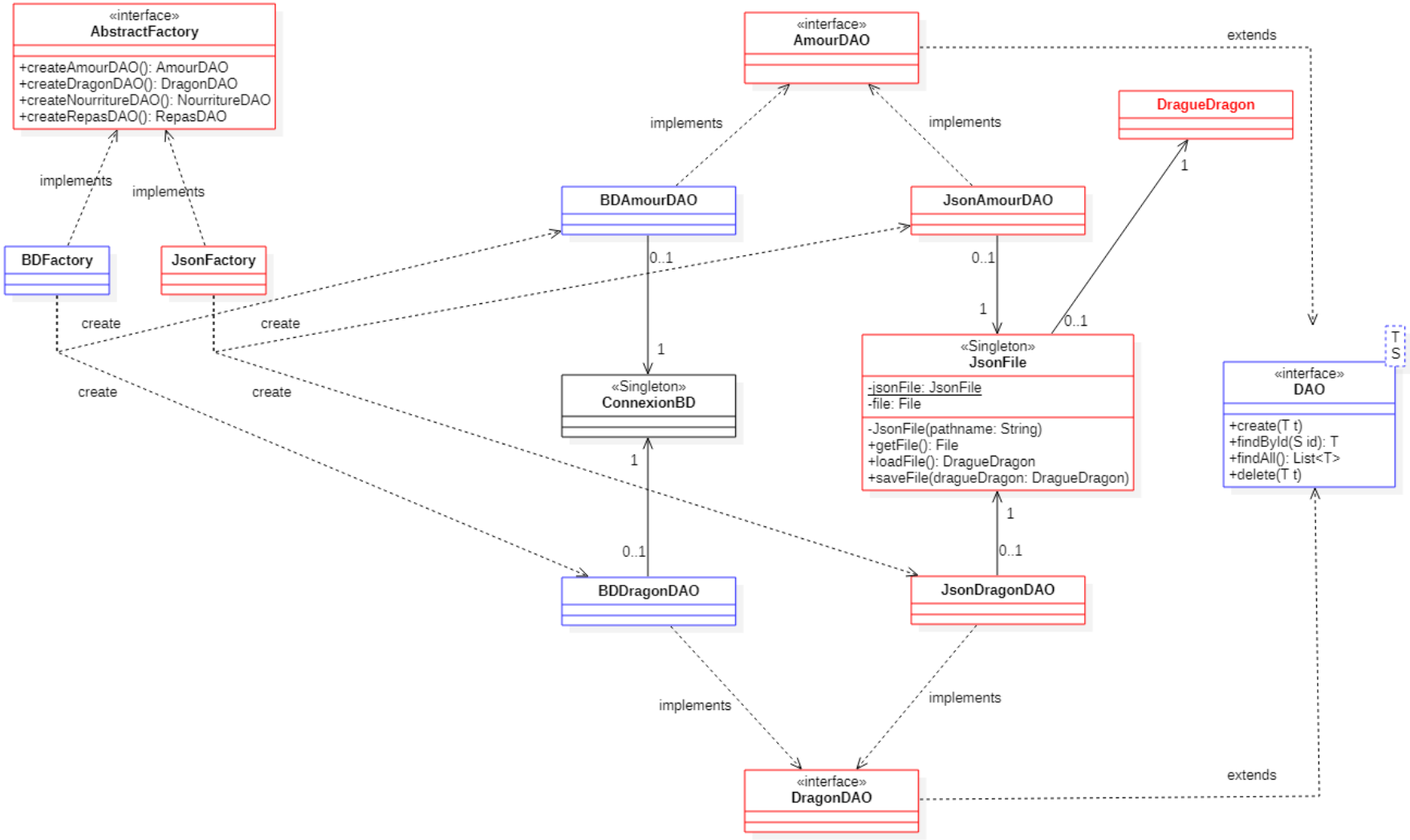
Exemple 1 : Abstract Factory
Structure générale



- Comporte une opération de création pour chaque produit
- Cache les classes concrètes des fabriques de produits

- Met en œuvre la création des produits concrets
- Une fabrique concrète pour une plate-forme/variante/version donnée ne fait que des produits concrets de la même plate-forme/variante/version

Exemple : Abstract Factory

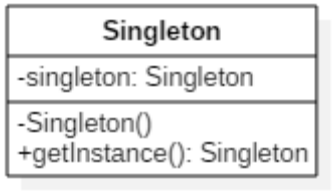


Exemple 2 : Singleton

- But :
  - Restreindre l'instanciation d'une classe à un seul objet

- Exemple :
  - Un objet unique pour un driver (base de données, etc)

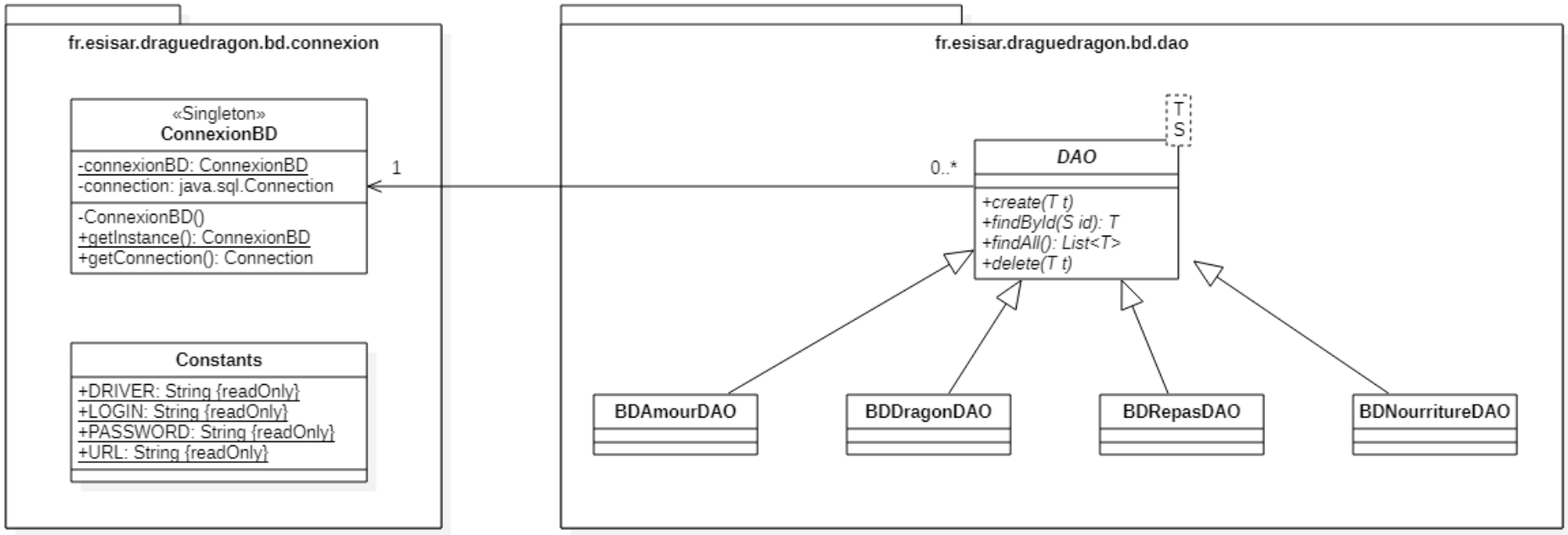
Exemple 2 : Singleton
Structure générale et exemple d'implantation




```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {
        super();
    }

    public static Singleton getInstance() {
        if(singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

Implantation non thread-safe !

Exemple : Singleton de connexion à une BD



Les autres patterns créateurs

- Builder:
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Factory Method:
  - Define an interface for creating an object, but let subclasses decide which class to instanciate. Factory Method lets a class defer instanciation to subclasses
- Prototype:
  - Specify the kinds of object to create using a prototypical instance, and create new objects by copying this prototype

Design Patterns Comportementaux

Un exemple : Observer

Objectifs des patterns comportementaux

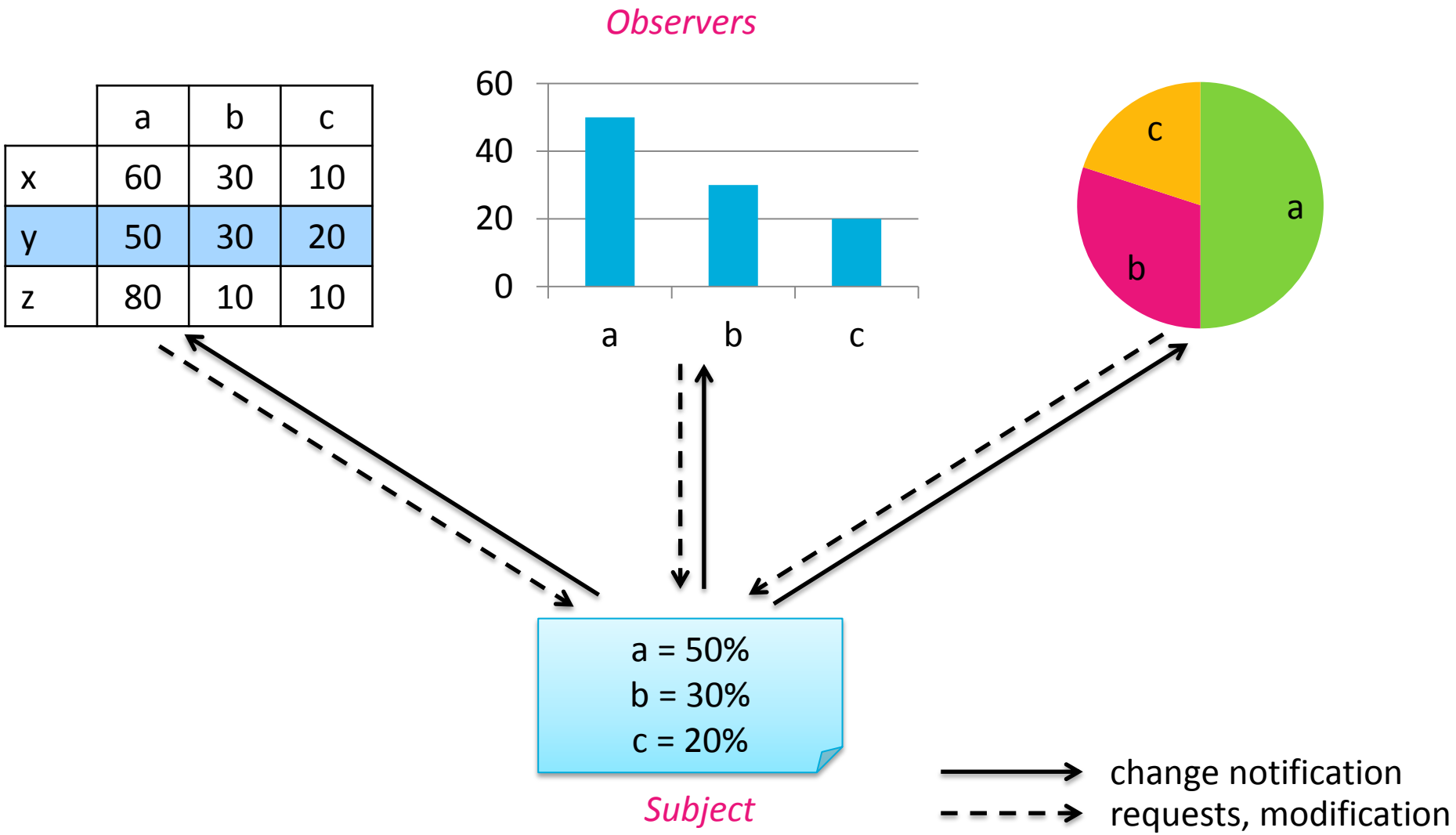
- Décrire des algorithmes
- Décrire des comportements entre objets
- Décrire des formes de communication entre objets

Exemple 3 : Observer

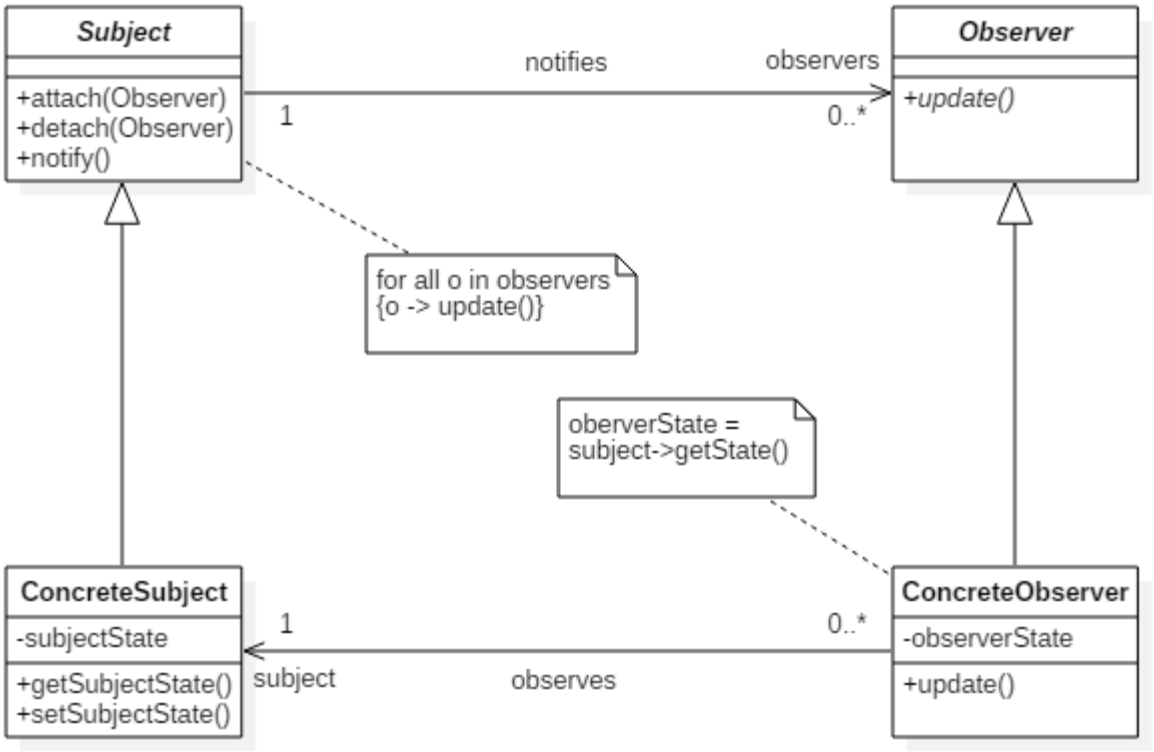
- But :
  - Coordonner des objets (un sujet et un ou plusieurs observateurs) de telle sorte que lorsque l'état interne d'un sujet change, tous les observateurs en sont automatiquement notifiés et se mettent à jour

- L'utilisation de ce motif est pertinente :
  - Lorsque le changement d'un objet se répercute à d'autres
  - Lorsqu'un objet doit prévenir d'autres objets sans pour autant les connaître

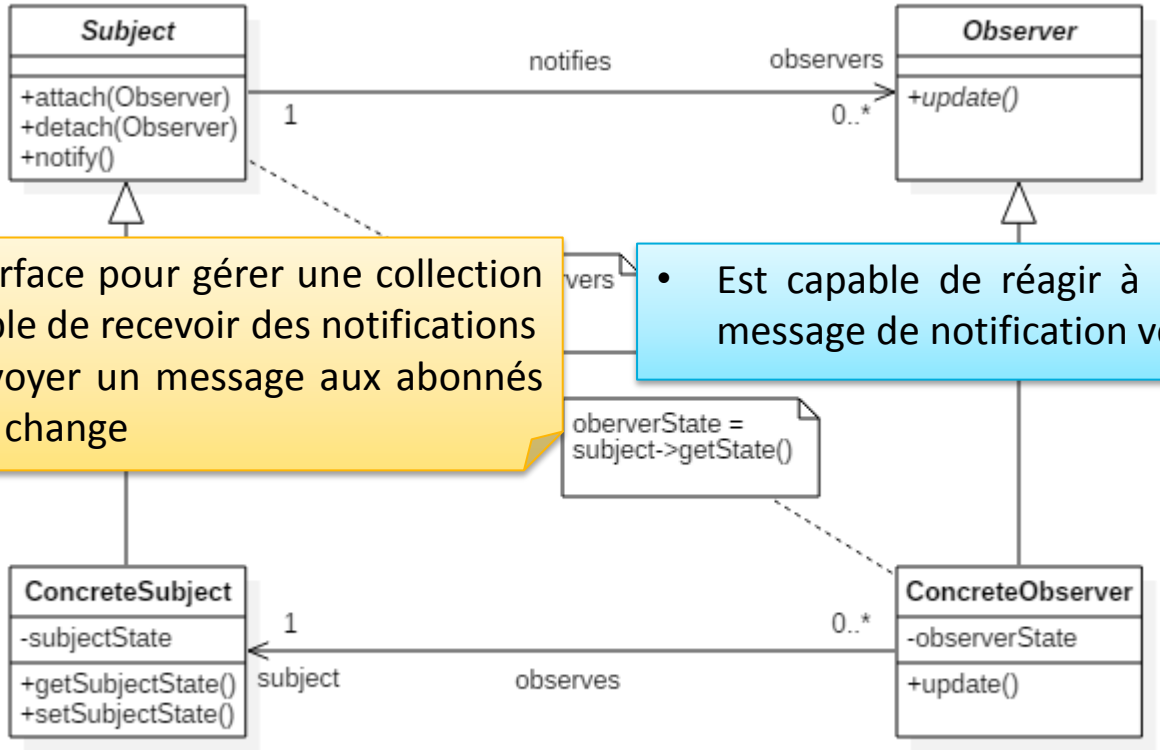
Exemple 3 : Observer
Une illustration



Exemple 3 : Observer
Structure générale



Exemple 3 : Observer
Structure générale



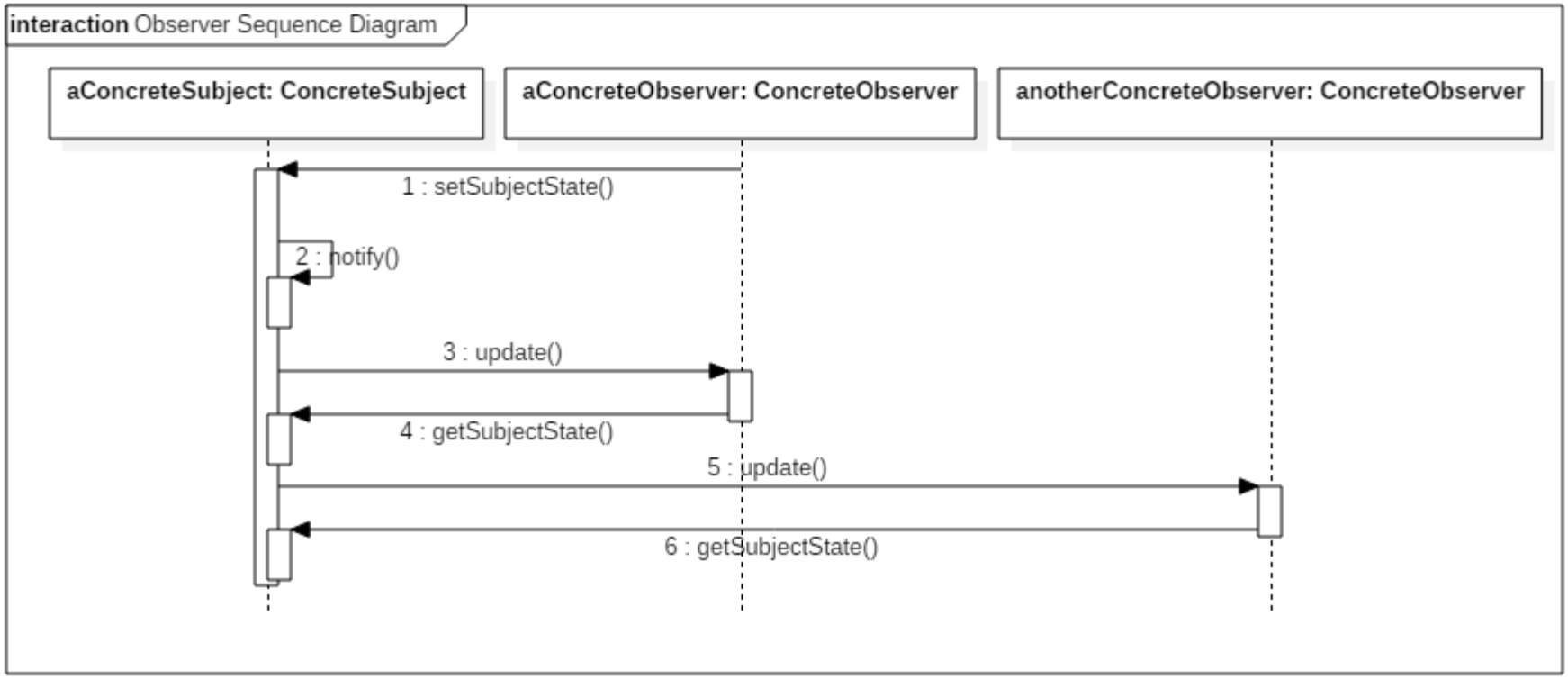
- Fournit une interface pour gérer une collection d'abonnés capable de recevoir des notifications
- Est chargé d'envoyer un message aux abonnés lorsque son état change

- Est capable de réagir à la réception d'un message de notification venant d'un sujet

- Contient les opérations d'accès à l'état
- Envoie une notification à ses observateurs lorsque son état change

- Maintient une référence à un sujet
- Stocke l'état du sujet qui doit être consistant avec celui du sujet
- Implante la mise à jour pour garder la consistance de l'état avec celui du sujet

Exemple 3 : Observer
Collaborations



Les autres patterns comportementaux – 1/3

- Chain of responsibility:
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle a request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command:
  - Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations
- Interpreter:
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Les autres patterns comportementaux – 2/3

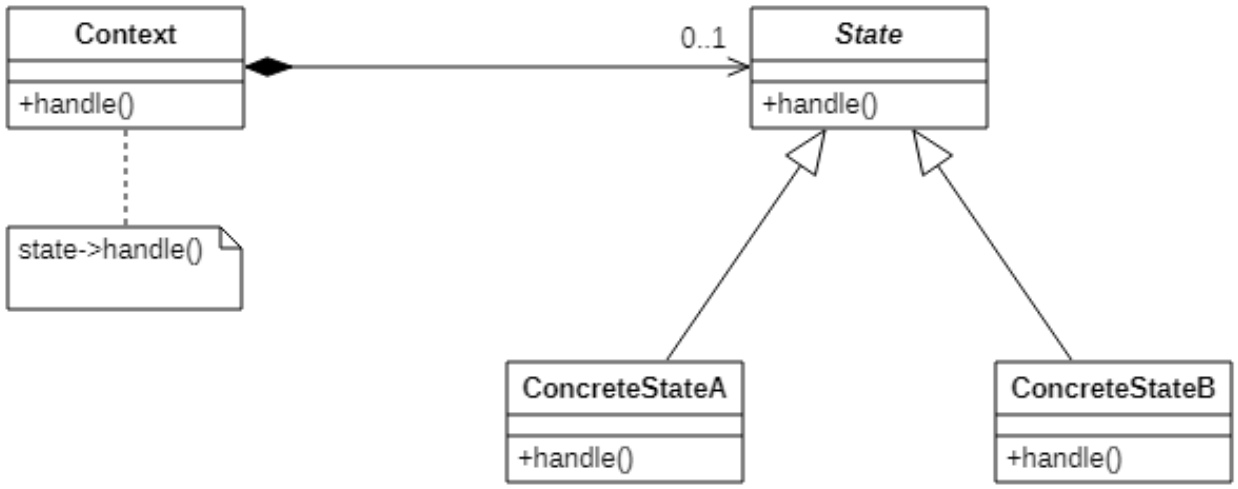
- Iterator:
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator:
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento:
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- State:
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Les autres patterns comportementaux – 3/3

- Strategy:
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method:
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor:
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Exercice

- Mise en application du patron State sur le robot de cuisine



Design Patterns structuraux

Un exemple : Composite

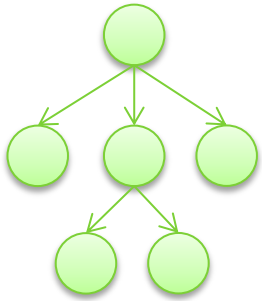
Objectif des patterns structuraux

- Découpler les interfaces et les implémentations des classes et des objets

- Décrire comment les objets sont assemblés

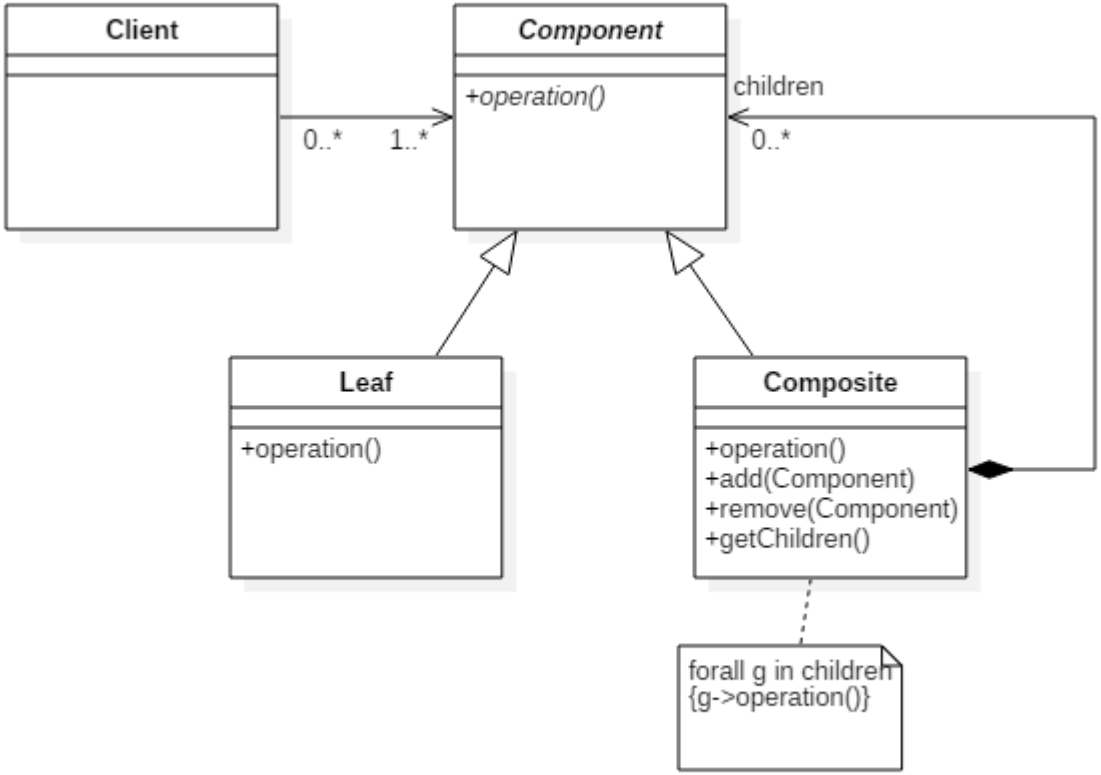
Exemple 4 : Composite

- But :
  - Représenter une hiérarchie d'objets
  - Ignorer la différence entre un composant simple et un composant en contenant d'autres



- L'utilisation de ce motif est pertinente :
  - Lorsque l'on veut représenter une hiérarchie
  - Lorsque l'on veut qu'un client puisse utiliser de manière uniforme les nœuds et les feuilles de la hiérarchie

Exemple 4 : Composite
Structure générale



Les autres patterns structuraux – 1/2

- Adapter:
  - Convert the interface of a class into another interface client expect. Adapter lets classes work together that could'nt otherwise because of incompatible interfaces.
- Bridge:
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Decorator:
  - Attach additional responsabilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Les autres patterns structuraux – 2/2

- Facade:
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight:
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy:
  - Provide a surrogate or placeholder for another object to control access to it.

Conclusion

Ce que n'est pas un pattern !

- Une brique :
  - Un pattern dépend de son environnement
  - Ce n'est pas du code
- Une règle :
  - Un pattern ne peut pas s'appliquer mécaniquement
  - Ne pas hésiter à l'adapter à ses besoins
- Une méthode :
  - Ne guide pas une prise de décision : un pattern est la décision prise
- Sans problèmes potentiels :
  - Plus de classes, plus de dépendances, besoin de documenter…
  - Mal utilisé peut affecter les performances !

Inconvénients des design patterns

- Efforts de synthèse : reconnaître, abstraire…
- Apprentissage, expérience
- Les patterns « se dissolvent » en état utilisés
- Nombreux…
  - Lesquels sont identiques ?
  - De niveaux différents
  - S'appuient sur d'autres patterns

Avantages des design patterns

- Un vocabulaire commun qui facilite la communication et la compréhension
- Capitalisation de l'expérience
- Niveau d'abstraction plus élevé qui permet d'élaborer des constructions logicielles de meilleure qualité
- Réduire la complexité
- Guide/catalogue de solutions

Attention !

Les antipatterns sont des erreurs courantes de conception de logiciels. Leur nom vient du fait que ces erreurs sont apparues dès les phases de conception du logiciel, notamment par l'absence ou la mauvaise utilisation de design pattern.

- Différents types :
  - De développement :
    - Abstraction inverse, action à distance, ancre de bateau, attente active, inter-blocages et famine, erreur de copier/coller, programmation spaghetti, réinventer la roue (carrée), surcharge des interfaces, l'objet divin
  - D'architecture :
    - Architecture as requirements, architecture by implication, coulée de lave, deuxième système, marteau doré