Introduction au génie logiciel

Modélisation :
Rappels sur l'étude statique

INFO0504 2021-2022

Université de Reims Champagne-Ardenne

Supports utilisés

- Spécification UML 2.5

- UML 2 par la pratique, Pascal ROQUES, éditions Eyrolles, ISBN 978-2-212-13344-8
- Architecture Logicielle, Jacques PRINTZ, 3ème édition, DUNOD, ISBN 978-2-10-057865-8

- *Design Patterns in JAVA, Vaskaran Sarcar, B/W Edition, IBSN 978-1-517-07144-8*

La modélisation idéale

- Objectifs :
  - De représentation non ambigüe,
  - De compréhension générale,
  - D'expressivité intuitive,
  - D'indépendance vis-à-vis des domaines techniques.

La modélisation réaliste

- Objectifs :
  - De limiter les ambigüités;
  - D'être accessible au plus grand nombre;
  - De correspondre à des vues classiques;
  - De représentativité d'une partie des domaines (informatiques).

Modélisation et UML

- UML ⇔ « *Unified Modeling language* »

- Mis en place par l'OMG (*Object Management Group*)

- Dernière version : 2.5.1 (décembre 2017)

⇔ modifications mineures de la 2.5 (juin 2015)

Modélisation et UML

- Ce qui définit UML (traduction littérale):
  - Une syntaxe abstraite basée sur une définition formelle des méta-modèle du MOF (*Meta-Object Facility*)

Modélisation et UML

- Ce qui définit UML (traduction littérale):
  - Des modèles à la sémantique clairement définie indépendante des technologies et s'intégrant dans un processus de génération par ordinateur.

Modélisation et UML

- Ce qui définit UML (traduction littérale):
  - Des éléments graphiques aisément compréhensibles à travers un ensemble de diagrammes décrivant les caractéristiques des systèmes modélisés.

Prise en main de l'UML

- 3 axes possibles :
  - Axe fonctionnel : définitions des actions du systèmes et relations entre les différents acteurs
  - Axe statique : représentation du système de manière global, de son architecture et éventuellement de son déploiement.
  - Axe dynamique : évolution du système en fonction des stimuli

Prise en main de l'UML

- 3 axes possibles :
  - Axe fonctionnel : définitions des actions du systèmes et relations entre les différents acteurs
  - **Axe statique : représentation du système de manière global, de son architecture et éventuellement de son déploiement.**
  - Axe dynamique : évolution du système en fonction des stimuli

Etude statique

- Vue architecturale du système
- Proche de la modélisation orientée objet
- Permet la génération automatique* de code

diagramme de classe

Etude statique

- Le diagramme de classe permet de :
  - visualiser les relations entre les objets :
    - Généralisation (héritage)
    - Agrégation/composition
    - Association
    - …
  - Guider le développeur indépendamment du langage

Etude statique

- Une classe possède au moins un nom

- Une classe peut aussi détailler :
  - Son type : interface, classe abstraite, …
  - Ses attributs (nom et accessibilité)
  - Ses capacités ⇔ opérations
  - Sa multiplicité

Etude statique

- Classe minimale

Pokemon

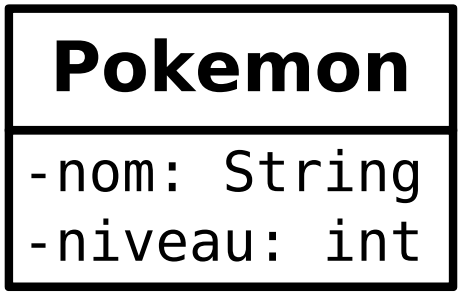
Etude statique

- Ajout d'attributs :
  - Privés : non visible pour les autres classes (même les sous-classes)

    ➔ au besoin créer des accesseurs
  - Protégés : comme « privé » sauf pour les sous-classes (les classes « filles »)
  - « Package » : visible pour toutes les classes d'un même « package » (voir diagramme correspondant)
  - Publics : pour les constantes généralement

Etude statique

- Ajout d'attributs :



- Accessibilité :
  - Prive ⇔ -
  - Protégée ⇔ #
  - Publique ⇔ +
  - « Package » ⇔ ~

Etude statique

- Ajout d'attributs :
  - d'instances : type par défaut où la valeur est spécifique à chaque instance
  - de classe : attribut dont la valeur est liée à la classe et non à l'instance

➔ attribut statique

- Exemples d'attribut de classe :
  - Une constante;
  - Un compteur d'instance;

Etude statique

- Ajout de compétences (opérations) :

- Accessibilité et type comme les attributs

- Opérations utilitaires (accesseurs, …)

Etude statique

- Relations entre les classes :
  - Dépendance

    ⇔ relation « utilise une instance de »

  - Association

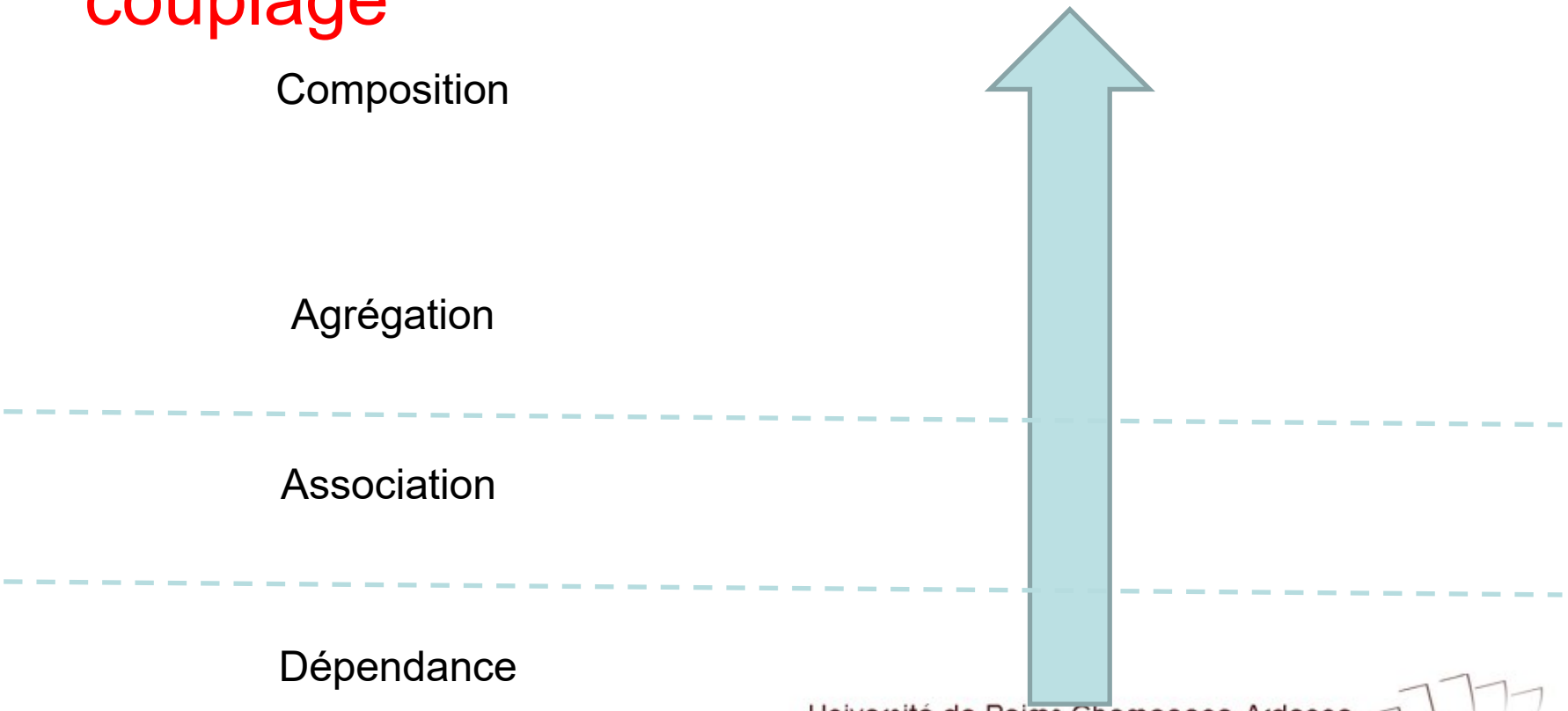
    ⇔ relation « possède une instance de »

  - Agrégation / Composition

    ⇔ relation « est composée de »

Etude statique

- Relations entre les classes : évolution du couplage



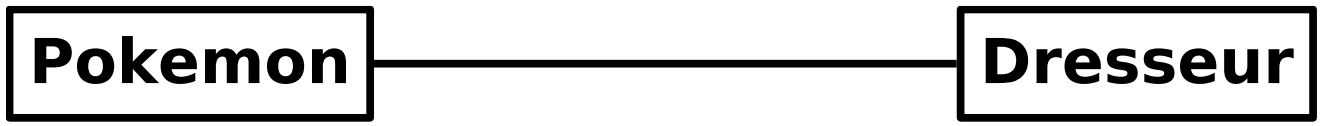
Etude statique

- Exemple de dépendance

  - La classe « Compétence » peut être utilisée comme :
    - Argument d'une opération
    - Ou au sein d'une opération
    - Ou encore comme valeur de retour

Etude statique

- Exemple d'association non orientée anonyme

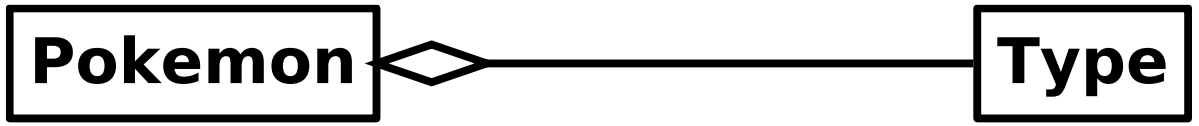


  - la classe « Pokemon » possède une instance de dresseur
  - la classe « Dresseur » possède une instance de Pokemon

    ➔ Cette relation n'est pas structurelle

Etude statique

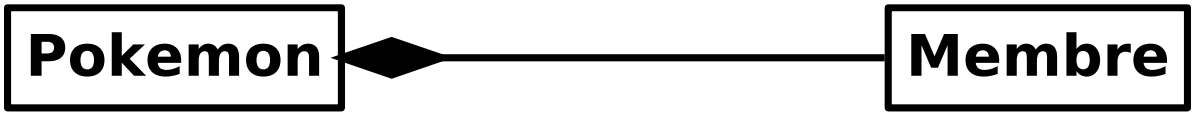
- Exemple d'agrégation



  - une instance de « Pokemon » possède forcément un type
  - une instance de « Type » peut exister indépendamment d'une instance de « Pokemon »

Etude statique

- Exemple de composition



  - une instance de « Pokemon » possède forcément (au moins) un « Membre »;
  - une instance de « Membre » est liée à instance de « Pokemon » et ne peut exister sans.

Etude statique

- Toutes les relations peuvent être :
  - Nommées
  - Orientées
  - Multiple entre objets
  - Associées à des contraintes
  - Qualifiées

Etude statique

- Relations spécifiques à la construction des classes :
  - Généralisation :

    ⇔Une classe est une spécialisation d'une classe plus générique (pouvant être abstraite);

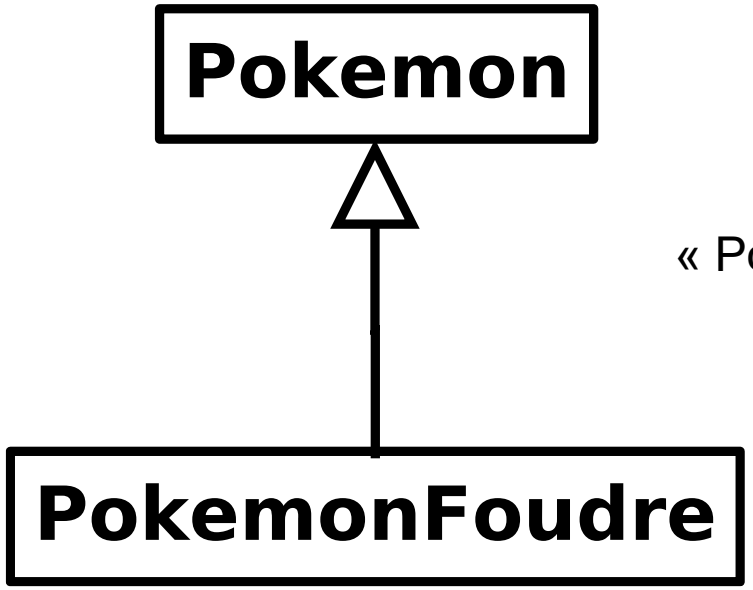
    ⇔Relation « est un ».
  - Réalisation :

    ⇔Une classe définie l'implémentation d'opérations (méthodes) définies dans une interface;

    ⇔Relation « peut être un ».

Etude statique

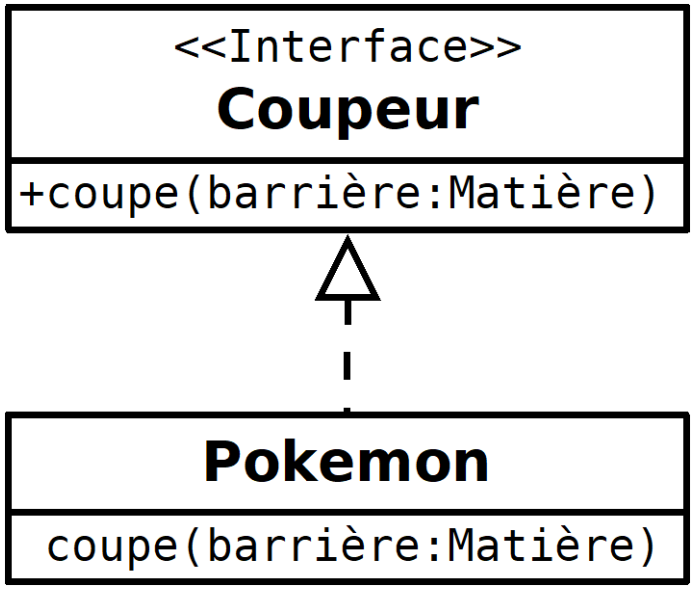
- Exemple de généralisation :



« PokemonFoudre » = « Pokemon » + …

Etude statique

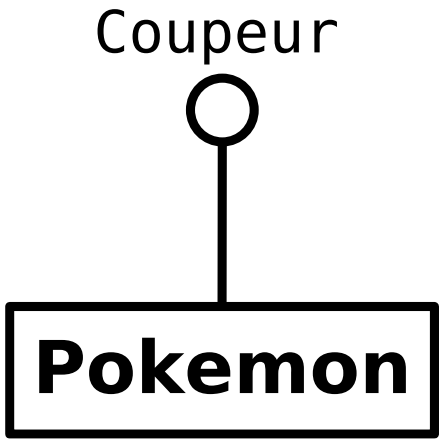
- Exemple de réalisation :



Un « Pokemon » peut être un « Coupeur » est donne l'implémentation de l'opération « coupe »

Etude statique

- Exemple de réalisation :



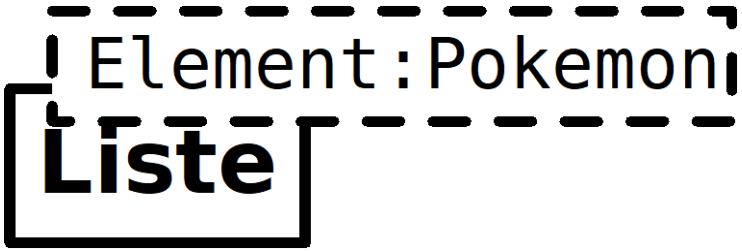
Un « Pokemon » peut être un « Coupeur »
est donne l'implémentation de l'opération
« coupe »

Etude statique

- Définition de traitements génériques

  ⇔ non liés à une classe spécifique

- Notion de « template » ➔ typé à l'instantiation

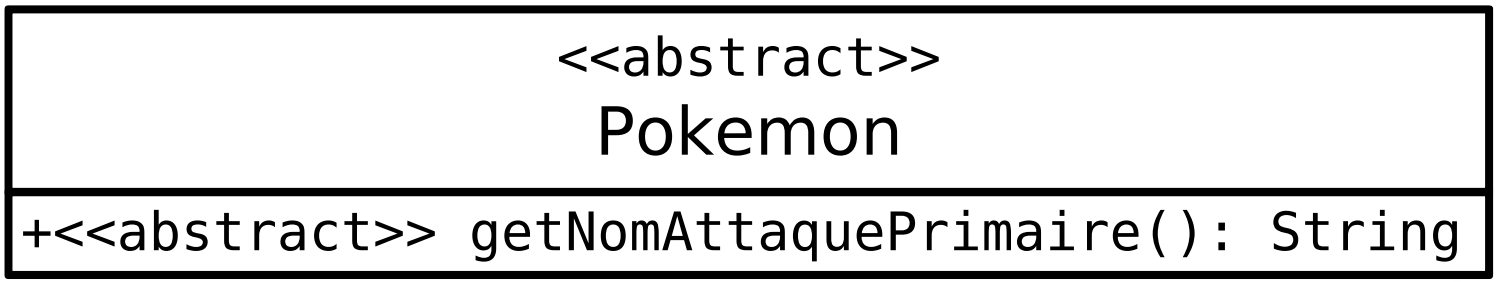


Etude statique

- Définition d'une entité sans pouvoir en définir une partie du fonctionnement interne.

  ⇔ Certaines parties dépendent d'une spécialisation

  ⇔ existence d'opérations abstraites



Etude statique

- Le diagramme de classe repose sur :
  - Le niveau d'abstraction choisi;
  - Le découpage cohérent en « package »;
  - L'expérience du modélisateur;
  - L'utilisation de « recettes » connues de la Conception Orientée Objet (COO).

Etude statique

- En COO, une « recette » ⇔ design pattern
- « Design pattern » (patron de conception)

  ⇔ un modèle valide et reconnu

  ⇔ un couple problème/solution

  ⇔ une vue indépendante d'un langage de programmation (objet)

Les « design patterns »

- Basés sur le livre (1994) :

« Design Patterns: Elements of Reusable Object-Oriented Software »

ISBN : 0201633612

- Du « Gang Of Four » ⇔ GOF :
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

Les « design patterns »

- Classés en 3 familles de patrons:
  - De création : lié à la construction des objets qui se veut être indépendante de leur structure
  - Structurels : détaille comment gérer la composition (au sens générique) des objets
  - De comportement : se basent sur la communication entre les objets et leurs rôles respectifs.

Les « design patterns »

- Il est important de :
  - Comprendre les « design pattern »
    ⇔ cas pratique de la COO
  - (re)connaître ces « design pattern » pour :
    - Au mieux éviter de les réinventer
    - Au pire proposer des modélisation non fonctionnelle pour l'objectif fixé

Les « design patterns »

- Certains peuvent sembler triviaux
- D'autres très techniques et abstraits ☺
- Leur point commun ⇔ proposer pour la COO des solutions :
  - de haut niveau
  - indépendante
  - fonctionnelle

Etude de quelques patrons de conception

Sera complété en M1

Les familles de patrons

- Création ⇔ vont porter sur l'instanciation efficace des objets.
- Structurel ⇔ vont jouer sur les relations entre les objets.
- De comportement ⇔ vont adapter la réactivité du système par rapport à son état.

Les « design patterns »

- De création :
  1. Le « singleton » :

     - Objectif : construire un objet et en assurer l'existence en un unique exemplaire
     - Difficulté : facile
     - Exemple : gérer le catalogue d'une bibliothèque

Les « design patterns »

- De création :
  1. Le « singleton »

Catalogue
-static catalogue: Catalogue
-Catalogue() +static getCatalogue(): Catalogue +insertEntry(in entry:Entry)

Les « design patterns »

- De création :
  1. Le « singleton » : cas d'accès
     - Non concurrent :

       Si instance == null

       création instance

       renvoi instance
     - Concurrent :
       - Synchronisation des appels à « getCatalogue »
       - Ou création de l'instance au chargement de l'objet
         - » La méthode se résume ainsi à retourner l'instance

Les « design patterns »

- De création :

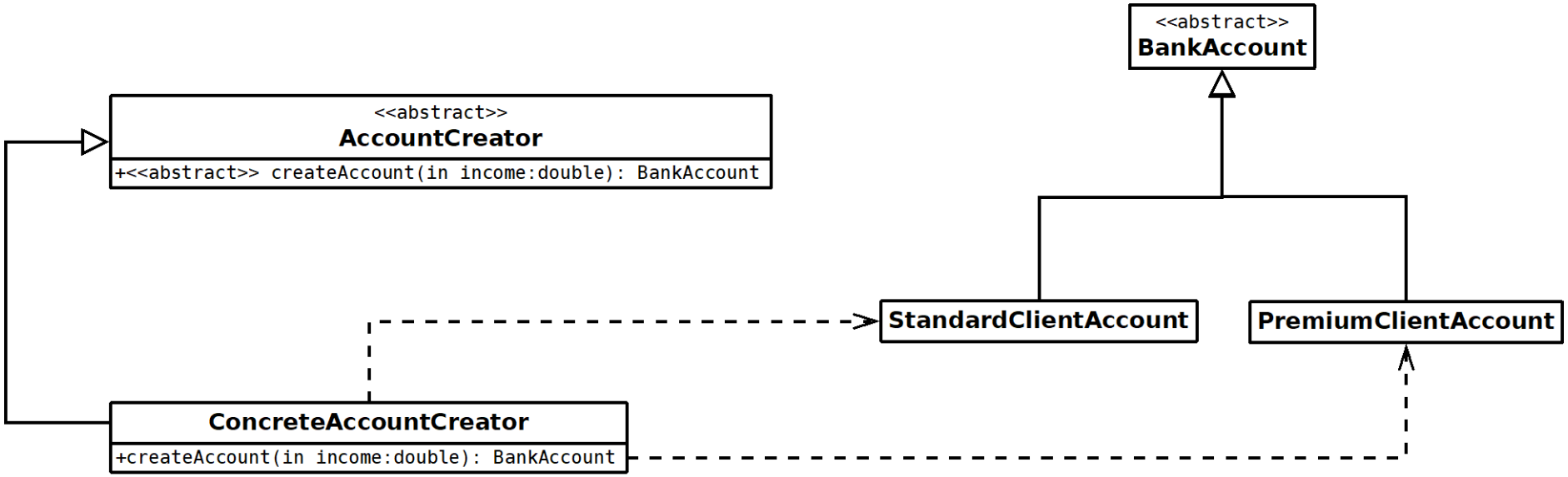
Non, je ne me suis pas trompé sur la numérotation, je suis juste faignant dans mes copier/coller

3. Le « factory method » :

  - Objectif : un classe cliente n'a pas plus besoin de connaître le type concret d'une classe pour la créer ➔ baisse du couplage
  - Difficulté : intermédiaire
  - Exemple : création de compte bancaire

Les « design patterns »

- De création :
  3. Le « factory method »



Les « design patterns »

- De création :
  3. Le « factory method » :
     - Seule la classe concrète de création connaît les vrais types des classes « BankAccount »
     - On peut aisément multiplier les types de compte et les classes de création associées
     - Le demande de création et la création sont 2 opérations indépendante des types réels

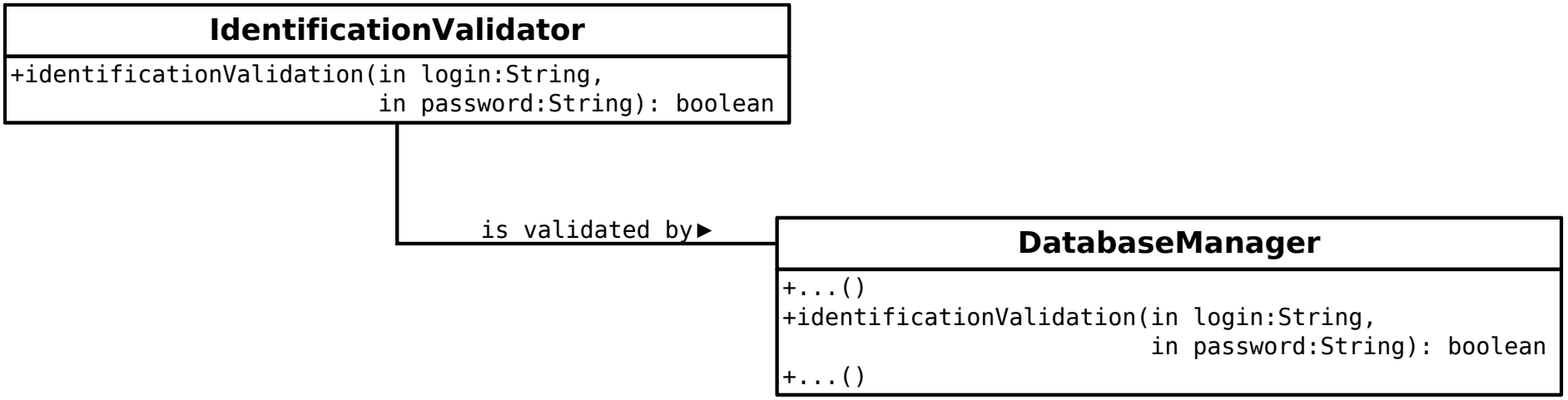
       ➔ baisse du couplage

Les « design patterns »

- Structurel :
  1. Le « proxy » :
     - Objectif : se propose de donner accès à une fonctionnalité en lieu et place de l'objet qui la propose vraiment.
     - Difficulté : facile
     - Exemple : identification d'utilisateur

Les « design patterns »

- Structurel :
  1. Le « proxy »



Les « design patterns »

- Structurel :
  1. Le « proxy » :
     - Permet de limiter l'accès à certains objets critiques
     - Permet d'associer une fonctionnalité à un type d'objet
     - Peut être mis à disposition en réseau (faible impact mémoire)
       - exemple en JAVA : objet sur le « RMI registry »
     - Peut entrainer une surabondance d'objets

Les « design patterns »

- Structurel :

  3. Le « composite » :

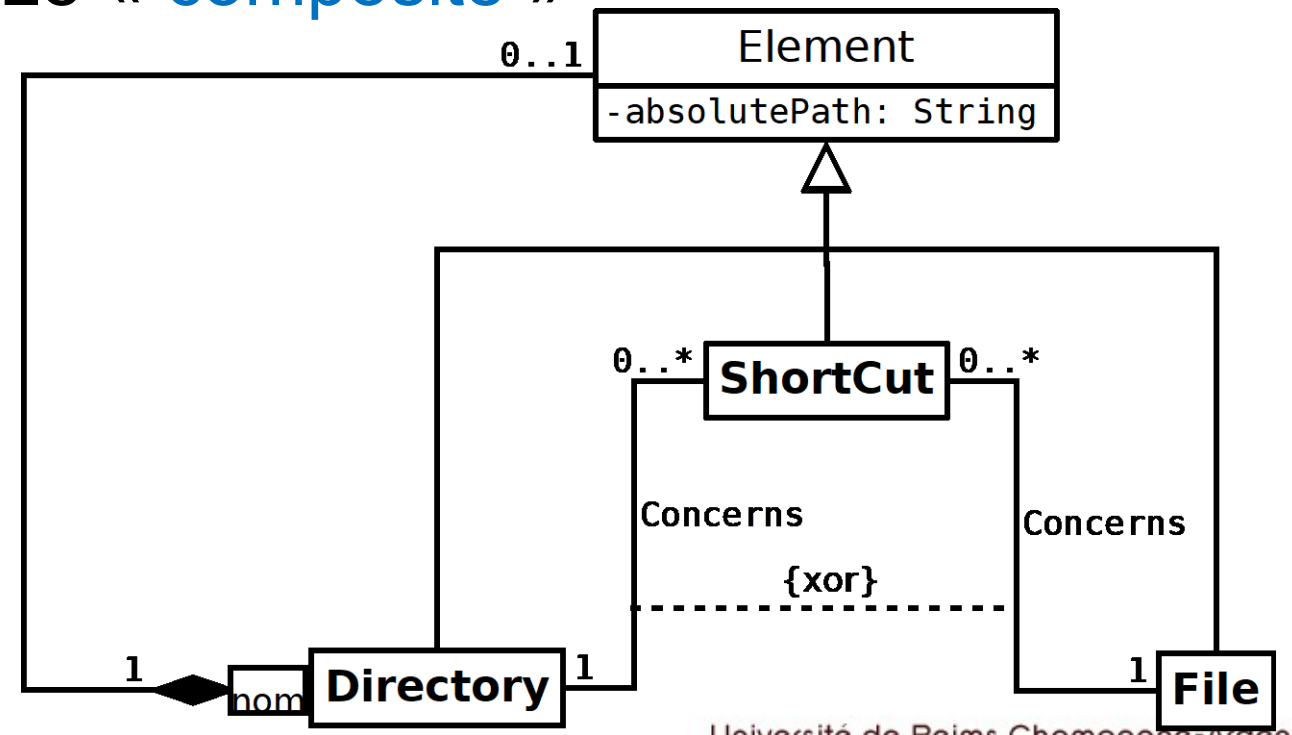
    - Objectif : s'abstraire de la différenciation entre objets seul et ensemble d'objets dans leur utilisation

    - Difficulté : facile (intuitif)

    - Exemple : le système de fichiers

Les « design patterns »

- Structurel :
  3. Le « composite »



Les « design patterns »

- Structurel :

  3. Le « composite » :

     - Permet de gérer des ensemble d'objet ou un objet indifféremment ⇔ le client n'a pas la conscience de ce qui est manipulé

     - Suit la structure « intuitive » des problèmes hiérarchique

     - Nécessite de faire attention à l'efficacité de la gestion mémoire (insertion, accès et délétion) et de ce qui est « type » dépendant.

Les « design patterns »

- Structurel :

  6. Le « decorator » :

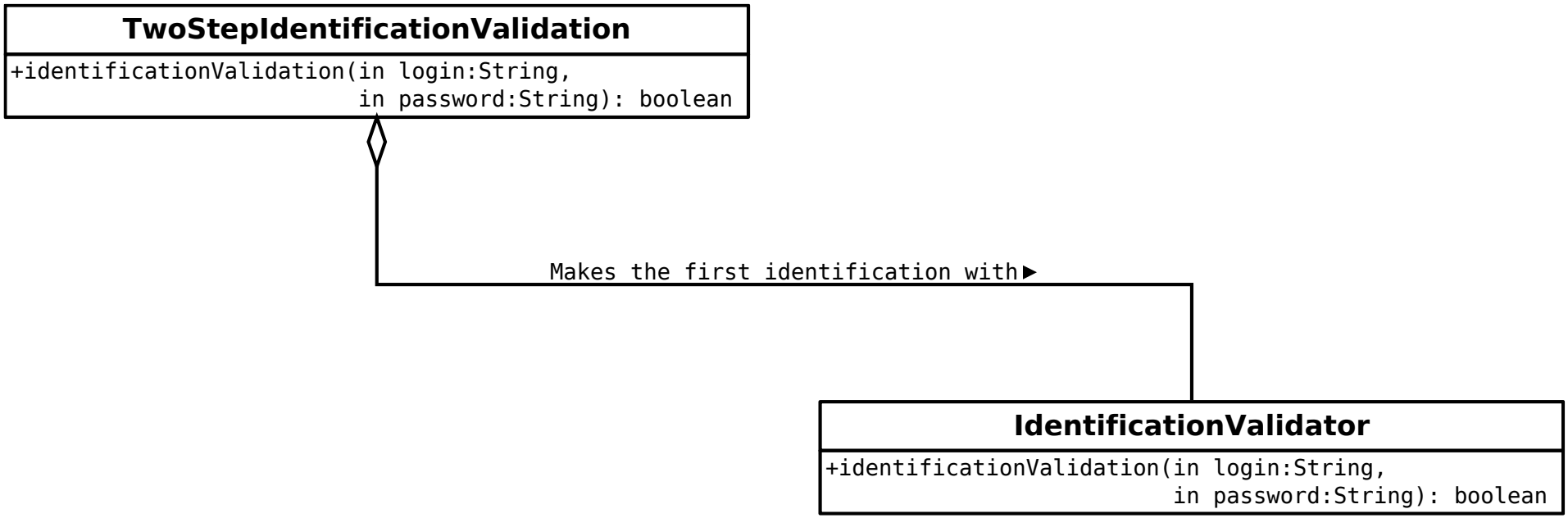
    - Objectif : permet l'ajout d'une fonctionnalité supplémentaire à un objet sans en modifier son comportement

    - Difficulté : facile

    - Exemple : l'identification en 2 étapes

Les « design patterns »

- Structurel :
  6. Le « decorator »



Les « design patterns »

- Structurel :

  6. Le « decorator » :

     - Est une solution rapide de rajout de fonctionnalités
     - Rajoute du couplage entre les objets ⇔ des fonctionnalités liées sont réparties sur plusieurs objets
     - Ne doit pas pallier à une mauvaise modélisation
       - Ajout d'une fonctionnalité qui aurait du être intégré à l'objet « décoré »

Les « design patterns »

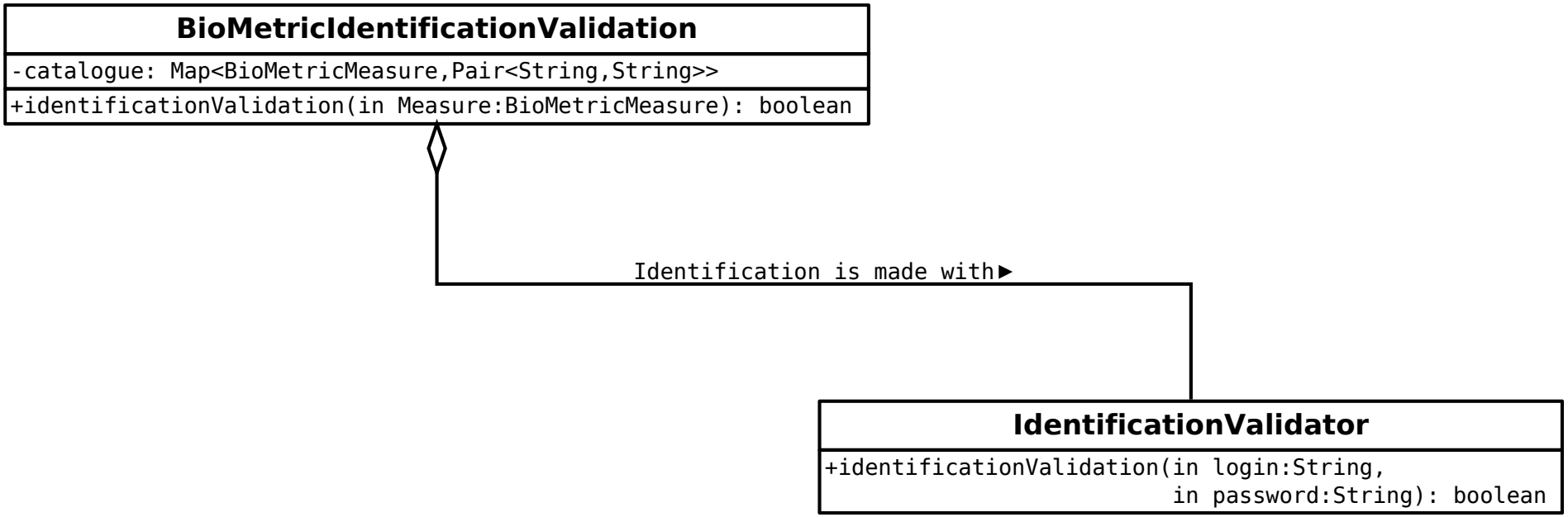
- Structurel :

  6. Le « adapter » :

    - Objectif : permettre à un objet ne rentrant dans la modélisation prévue de fonctionner quand même
    - Difficulté : moyenne
    - Exemple : l'identification biométrique

Les « design patterns »

- Structurel :
  6. Le « adapter »



Les « design patterns »

- Structurel :

  6. Le « adapter » :

     - Permet de contourner le typage
     - Ne doit pas être confondu avec le « decorator »
     - Doit essayer de rester logique au niveau modélisation

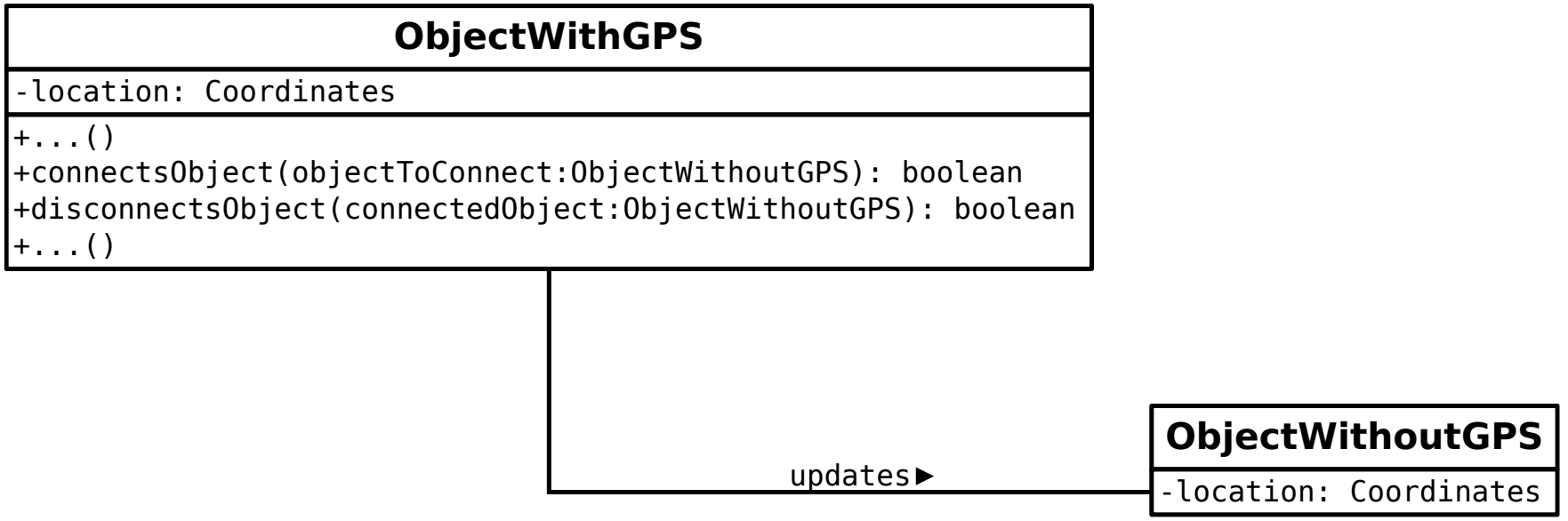
     ➔ c'est souvent une solution de simplicité pour l'implémentation

Les « design patterns »

- De comportement :
  1. Le « observer » :
     - Objectif : un ensemble d'objets veulent être mis à jour en même temps qu'un objet cible.
     - Difficulté : facile
     - Exemple : localisation

Les « design patterns »

- De comportement :
  1. Le « observer »



Les « design patterns »

- De comportement :
  1. Le « observer » :
     - C'est l'objet « observé » qui s'occupe de la mise jour des « observateurs »
     - Relation tout ou rien par défaut
     - Est à la base du système d'observation/réaction des interfaces graphiques

Les « design patterns »

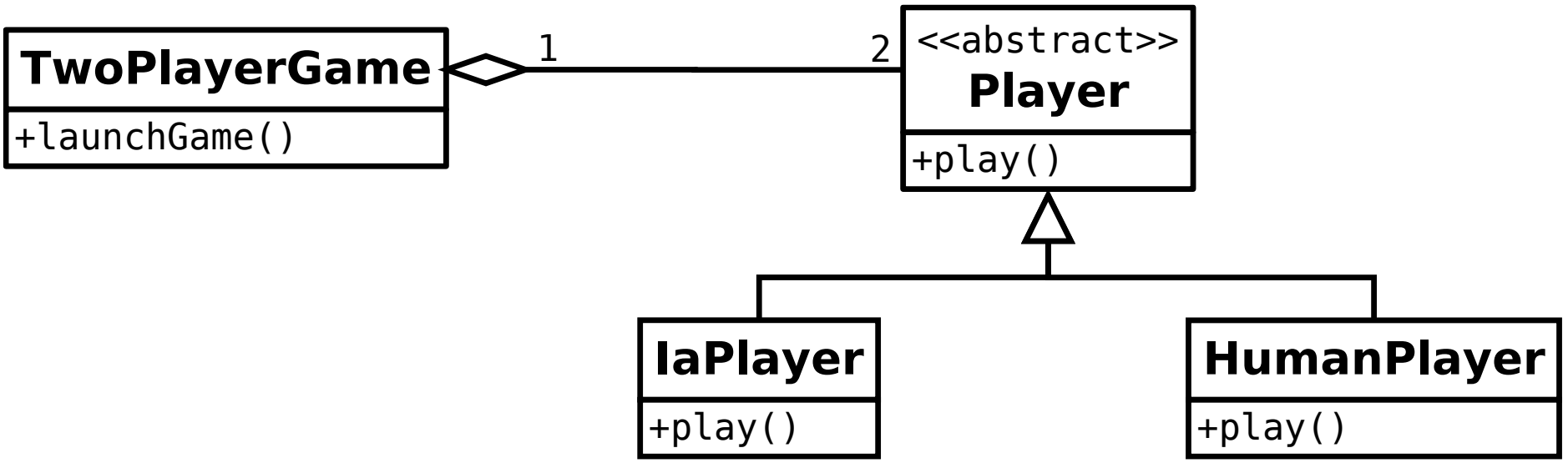
- De comportement :

  2. Le « strategy » :

     - Objectif : faire abstraction du fonctionnement même d'un algorithme face aux objets en interaction.
     - Difficulté : moyenne
     - Exemple : jeu à 2 joueurs

Les « design patterns »

- De comportement :
  2. Le « strategy »



Les « design patterns »

- De comportement :
  2. Le « strategy » :
     - Se comprend aisément dans le cas des jeux (d'où l'exemple).
     - Permet une adaptation dynamique du meilleur algorithme à mettre en place
     - Permet une évolution des approches sans réécriture de la plateforme globale

Les « design patterns »

- De comportement :

  5. Le « iterator » :

     - Objectif : faire abstraction de la manière dont le parcours séquentiel (efficace) d'une structure de données est faite

     - Difficulté : facile

     - Exemple : JAVA
       - Boucle « for » VS « iterator »

Les « design patterns »

- De comportement :

  5. Le « iterator » :

    - Permet de pas se préoccuper de la gestion des accès à une structure de données

      ➔ **Attention d'en connaître le complexité**

    - Est un « pattern » classique des API objets

Les « design patterns »

- De comportement :

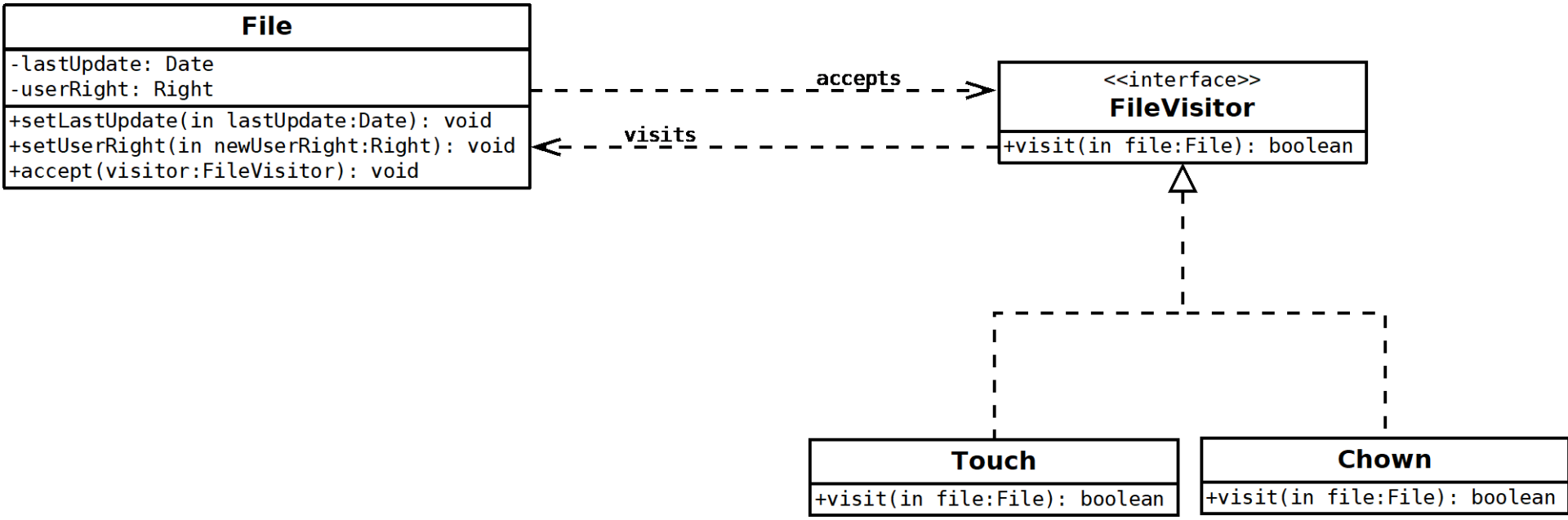
  6. Le « visitor » :

    - Objectif : permet d'effectuer de nouveaux types de modification de l'état d'un objet sans changer son interface :
      - La modification est faite par un autre objet ⇔ le visiteur
      - Elle est cependant contrôlée par le « visité »

    - Difficulté : moyenne (pas forcément intuitif)
    - Exemple : les commandes « touch » et « chown »

Les « design patterns »

- De comportement :
  6. Le « visitor »



Les « design patterns »

- De comportement :

  6. Le « visitor » :

     - L'objet « visitable » doit pouvoir accepter des « visiteur »
     - On peut coupler « visitor » et « iterator » pour modifier un ensemble d'objets
     - Permet une extension des fonctionnalités de l'objet initial

Les « design patterns »

- Un projet ⇔ combinaison de « patterns »
  - Exemples :
    - « iterator » + « visitor »
    - « factory » + « singleton »
    - ~~« abstract factory » + « prototype »~~
- Existe-t-il des recettes de combinaisons de « pattern » ⇔ « pattern » de « design pattern » ?

L'architecture MVC

- MVC pour :
  - Modèle : gestion des données
  - Vue : le rendu proposé au client
  - Contrôleur : médiateur entre le client, le modèle et la vue

L'architecture MVC

- Le modèle :
  - Assure les accès aux données brutes :
    - Base de données ⇔ requête
    - Accès distants ⇔ objets distants
  - Est lié au contrôleur (entrée/sortie)
  - N'est pas lié directement :
    - au client
    - Ni à la vue

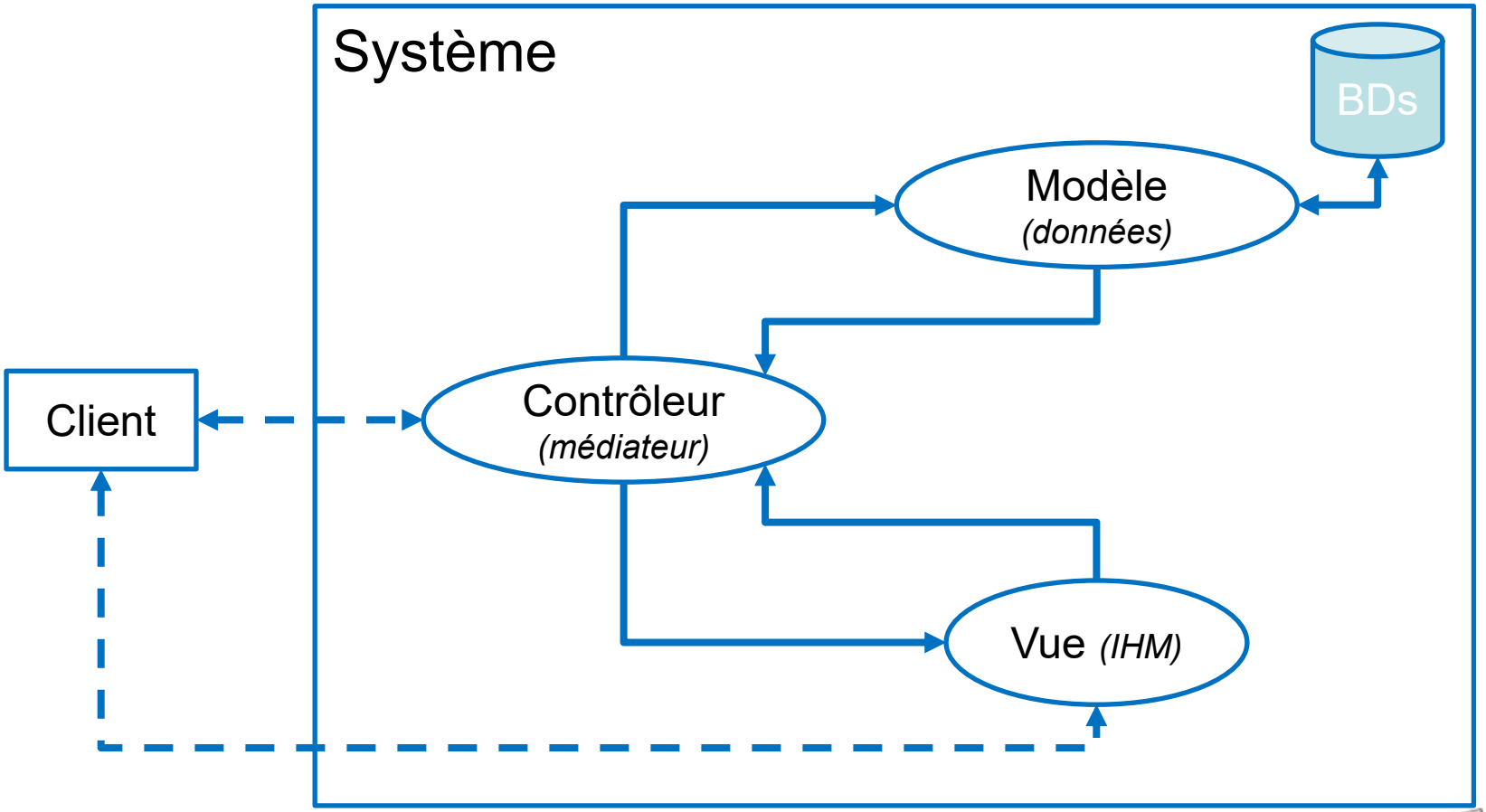
L'architecture MVC

- La vue :
  - Assure le rendu des informations :
    - Formatage du contenu : ce que voit le client n'est pas forcément le format réel de l'information
    - Ergonomie des services : les besoins du client sont illustrés simplement
  - N'est lié au modèle
  - Est lié :
    - au contrôleur (entrée/sortie)
    - Au client (entrée/sortie)

L'architecture MVC

- Le contrôleur :
  - Assure les traitements internes : calcul, déploiement, croisement de l'information, …
  - Coordonne en
    - entrée :
      - L'accès aux données brutes via le modèle
      - La mise à jour de la vue en fonction des traitements
    - sortie :
      - La redistribution des retours des requêtes du modèle à la vue après traitement
  - Est lié au client, au modèle et à la vue

L'architecture MVC



L'architecture MVC

- Il existe diverses implémentation du modèle MVC ⇔ « framework » :
  - Brute
  - Évoluée :

    (non exhaustif)
    - Pattern « factory » pour le choix du contrôleur
    - Pattern « chain of responsabilities » pour le choix du modèle
    - Un contrôleur / requête client
    - Méta-contrôleur / Méta-modèle
    - …