- 4 -

Conception et implantation

UNIVERSITÉ DE LORRAINE

Analyse et Conception de Logiciels

Jean-Christophe Bach

- 4.3 -
Design patterns
- 3ème partie -

Licence

CC BY-SA 3.0

(Attribution - Partage dans les Mêmes Conditions 3.0 non transposé)

Hors logo de l'Université de Lorraine et images de la diapo 22

Bibliographie et sources d'inspiration de ce cours

- Livres :
  - *Design Patterns : Elements of Reusable Object-Oriented Software* – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides ; Addison-Wesley, 1994
  - *Head First* – Elisabeth Freeman, Eric Freeman, Bert Bates ; O'Reilly, 2004
- Liens web :
  - `http://www.vincehuston.org/dp/`
  - `http://www.hillside.net/patterns/`
  - `http://www.dofactory.com/Patterns/Patterns.aspx`
- Cours de :
  - Olivier Boissier et Gauthier Picard, Mines de Saint-Étienne (2009)
  - Bruno Mermet, Université du Havre (2009)

Aujourd'hui : 4 *design patterns*

- *Architectural*
  - Data Access Object
- *Creational*
  - Prototype
- *Structural*
  - Adapter
  - Façade

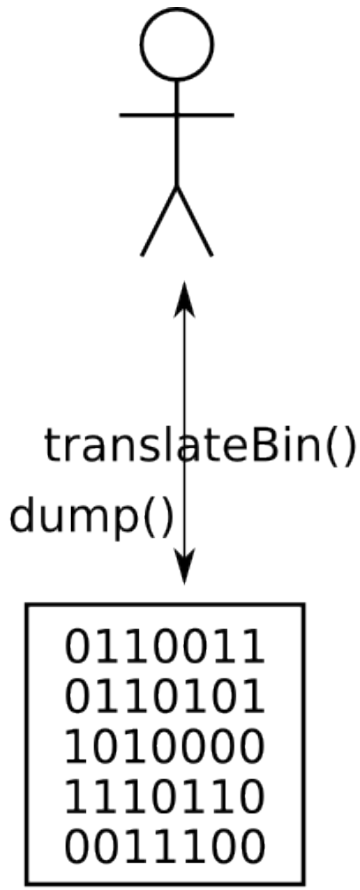
Data Access Object (DAO)

- Patron de conception architectural
- Bon complément du MVC

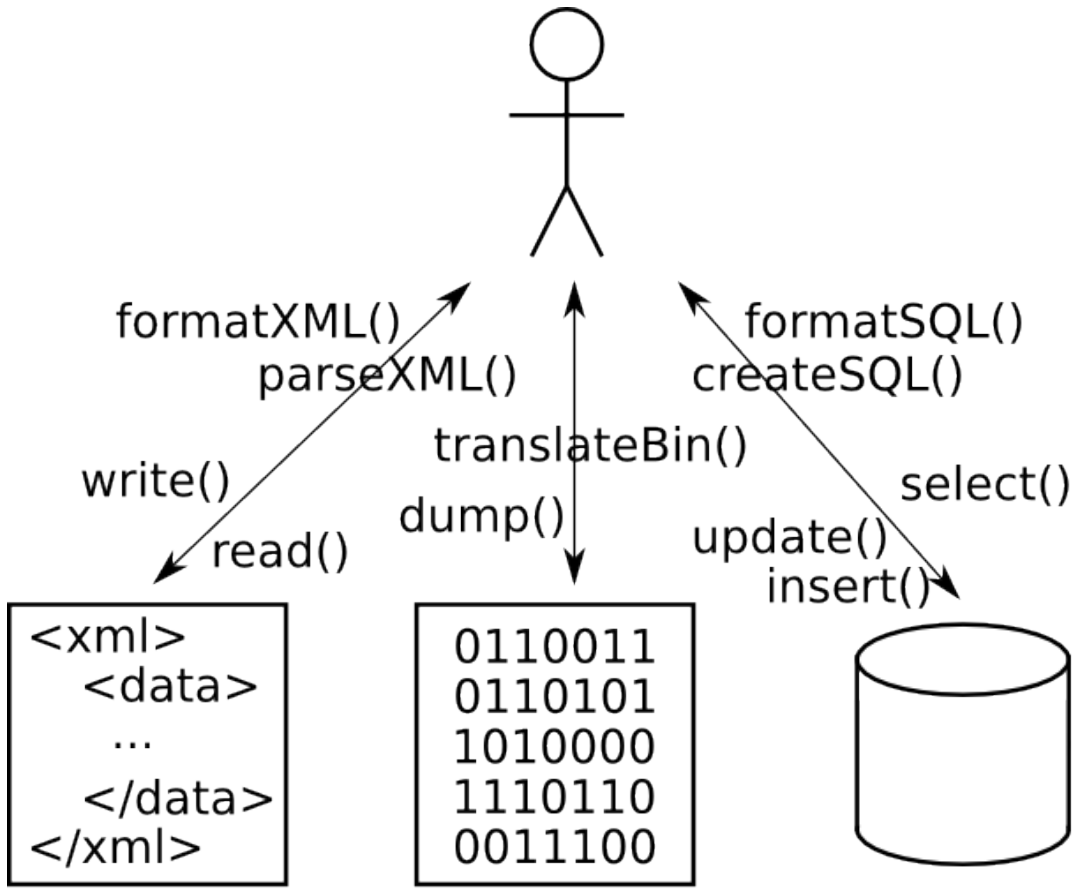
Problème

Comment implémenter la persistance de données ?

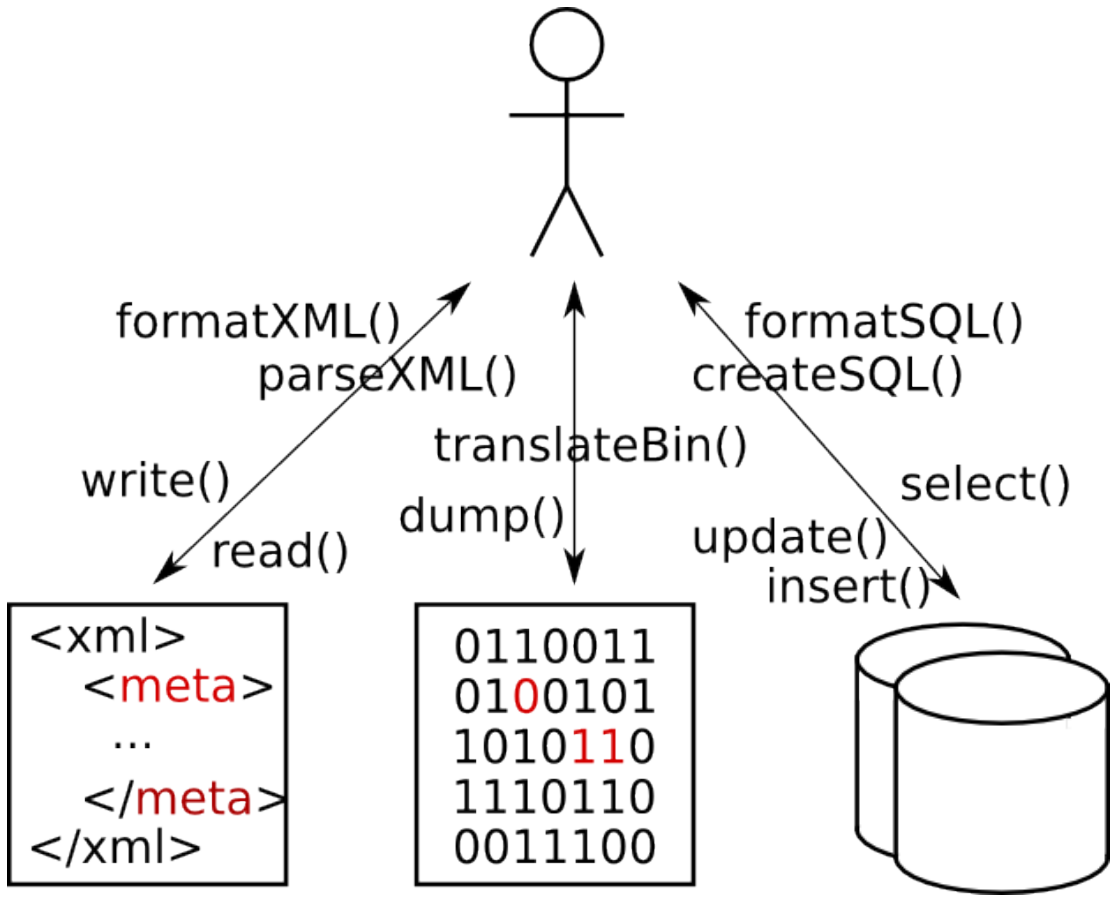
Problème


translateBin()
dump()
0110011
0110101
1010000
1110110
0011100

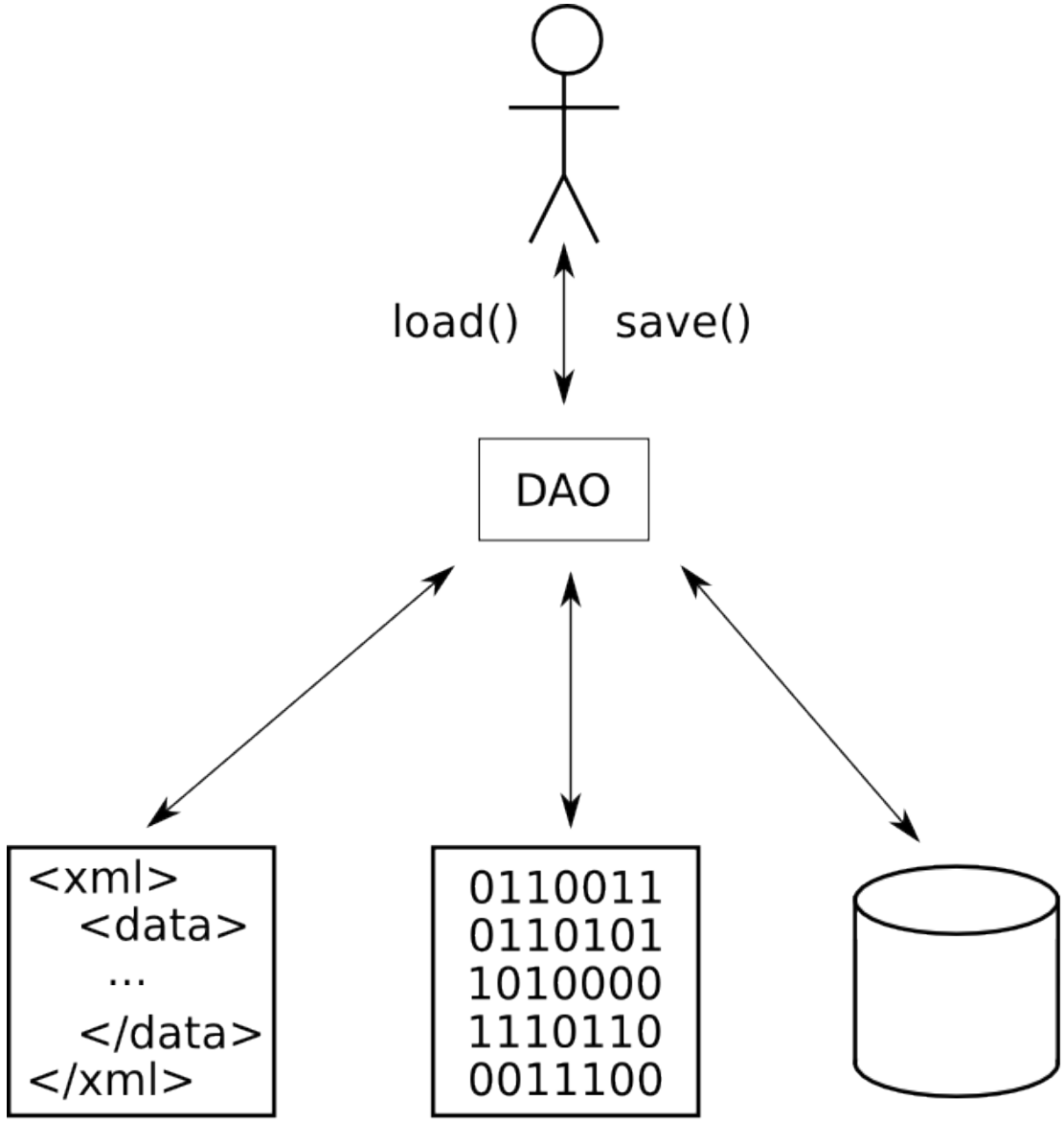
Problème


formatXML()
parseXML()
write()
read()
translateBin()
dump()
formatSQL()
createSQL()
select()
update()
insert()
<xml>
<data>
...
</data>
</xml>
0110011
0110101
1010000
1110110
0011100

Problème


formatXML()
parseXML()
formatSQL()
createSQL()
translateBin()
write()
read()
dump()
select()
update()
insert()
<xml>
<meta>
...
</meta>
</xml>
0110011
0100101
1010110
1110110
0011100

Problème


load()
save()
DAO
<xml>
<data>
...
</data>
</xml>
0110011
0110101
1010000
1110110
0011100

Data Access Object : intention

- Abstraire le stockage des données
- Encapsuler tous les accèes aux données

$\Rightarrow$ le DAO gère la connexion avec les données pour obtenir et stocker les données

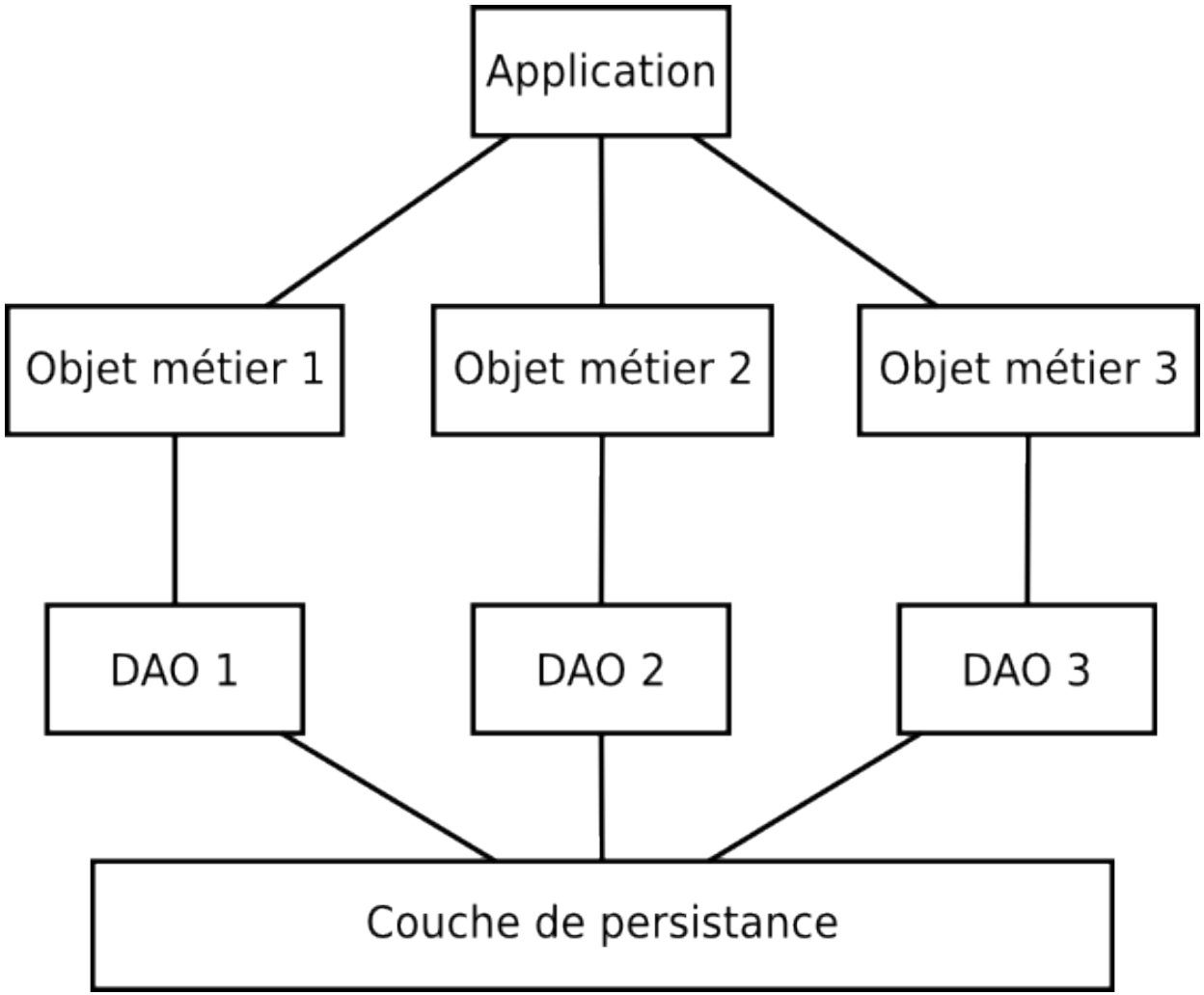
Data Access Object : indications d'utilisation

- Quand on souhaite faire de la persistance de données
- Quand on souhaite pouvoir sauvegarder dans différents formats de données ou indépendamment des supports et technologies de stockage

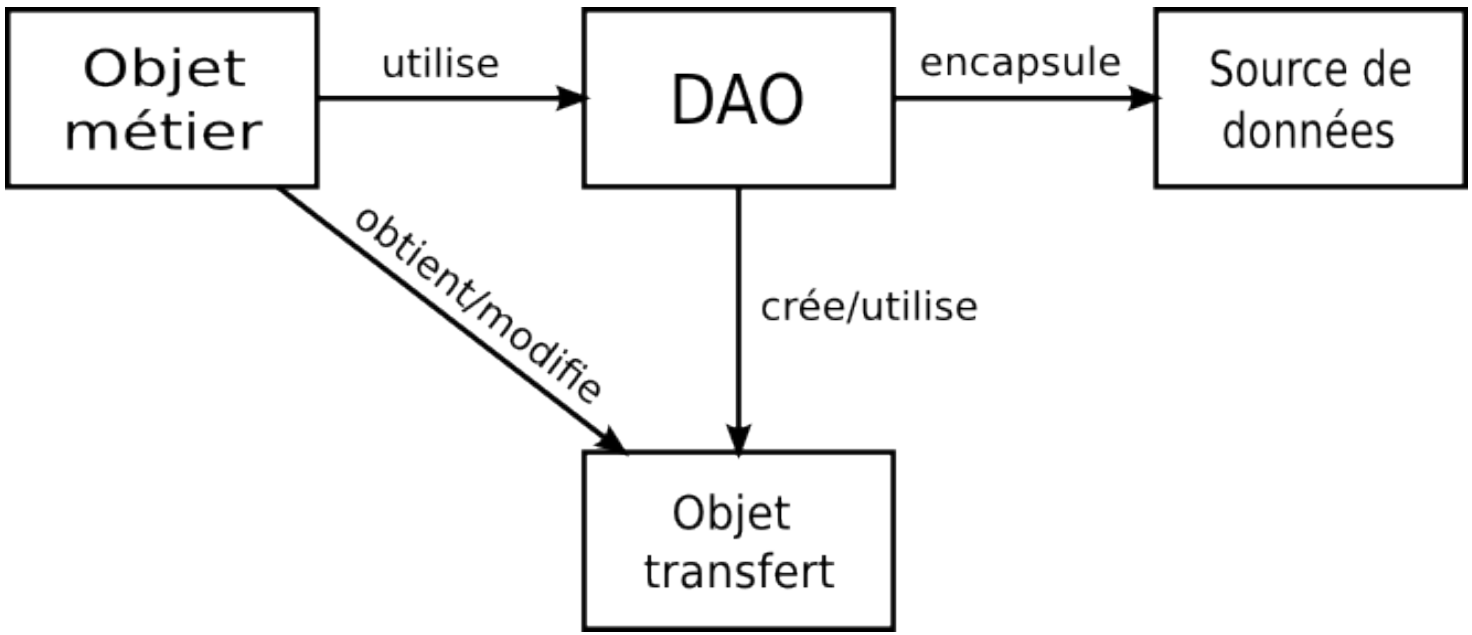
  ex :

  - un ou plusieurs fichiers
  - texte brut, XML, binaire, Base de données
  - technologies de BdD variées (SQLite, PostgreSQL, etc.)

Data Access Object : vue abstraite



Data Access Object : structure générale



Data Access Object : remarques

- Un objet DAO par classe métier (et non par objet)
  - objet DAO peut implémenter *Singleton*
- Couche de persistance uniforme
  - instanciation des DAO par famille
  - utilisation d'*Abstract Factory*
- Objet Transfert pas indispensable

Data Access Object : conséquences

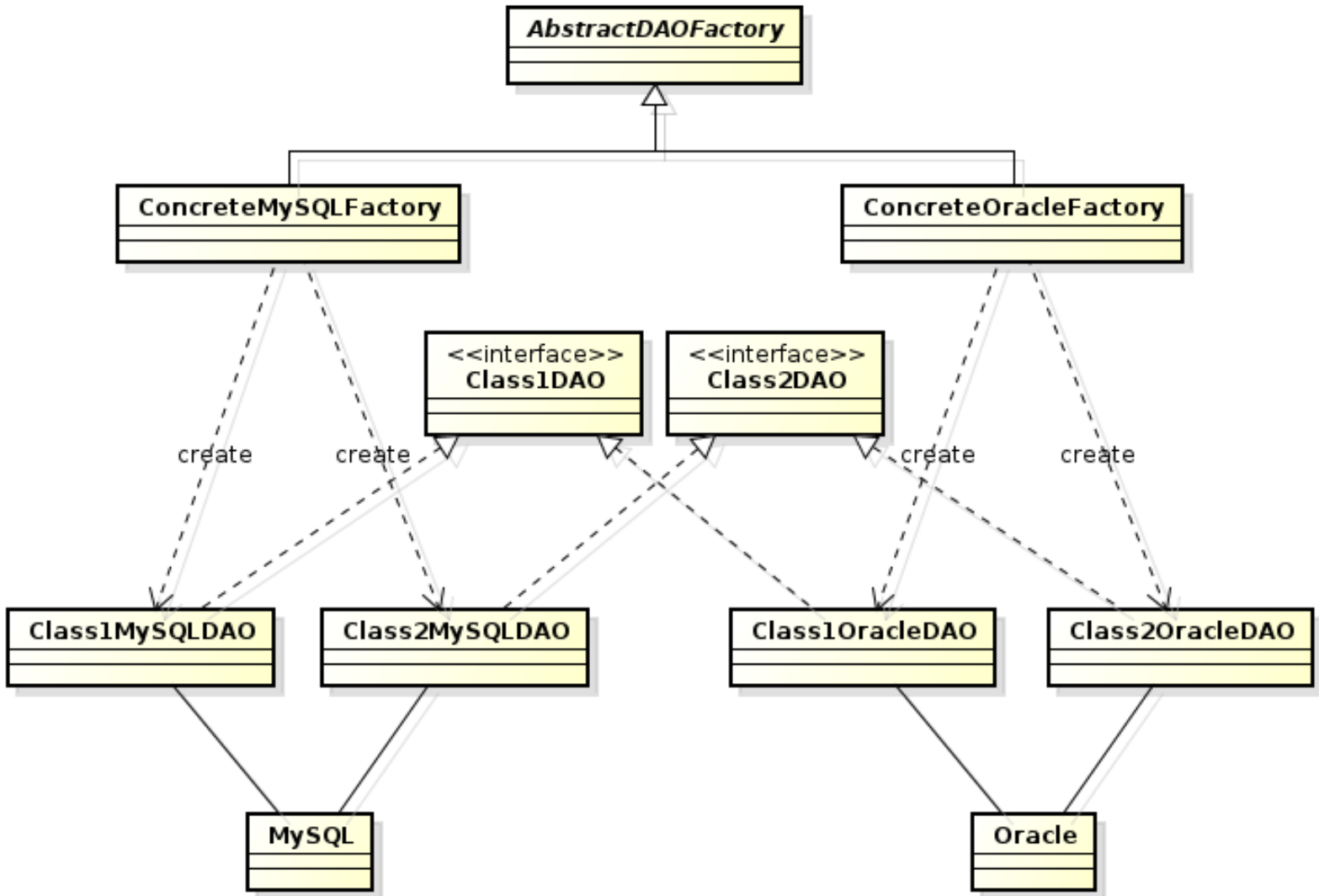
Avantages

- Robustesse au changement : changer le type de stockage sans changer l'architecture
- Flexibilité et qualité du code : programmation par interface
- Facilement et rapidement testable : tests unitaires + MOCKs plutôt que des tests d'intégration avec une base de données

Inconvénients

- Mise en œuvre plus complexe
- Multiplication des classes

Data Access Object : exemple d'implémentation (1/4)



Data Access Object : exemple d'implémentation (2/4)

Une Abstract Factory :

```
public abstract class AbstractDAOFactory {

  public abstract Class1DAO getClass1DAO();

  public abstract Class2DAO getClass2DAO();

  public static getAbstractDAOFactory(int sgbd) {
    switch(sgbd) {
      case Mysql:
        return new ConcreteMySQLFactory();
        break;
      case Oracle:
        return new ConcreteOracleFactory();
        break;
    }
  }
}
```

Data Access Object : exemple d'implémentation (3/4)

Implémentation concrète de l'Abstract Factory

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

  public Class2DAO getClass2DAO() {
    return Class2MySQLDAO.getInstance();
  }
}
```

Spécification d'un DAO

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

Data Access Object : exemple d'implémentation (4/4)

Implémentation d'un DAO

```
public class Class1MySQLDAO implements Class1DAO {
  private static instance = null;
  private Class1MySQLDAO() { /*ex : connexion à la base*/ }
  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
  }
}
```

Data Access Object : résumé

- Persistance, couche uniforme
  - un DAO par classe métier
  - instanciation des DAO par famille
- Découplage de l'application métier et des accès bas niveau aux données
- Souvent associé à d'autres patrons de conception
  - Factory
  - Singleton

Prototype

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

Problème

Comment lui fournir une armée ?

Problème

Travailler avec un cabinet de recrutement pour embaucher un *stormtrooper* ?

Problème

Travailler avec un cabinet de recrutement pour embaucher un *stormtrooper* exemplaire ?

Problème

Prototype : intention

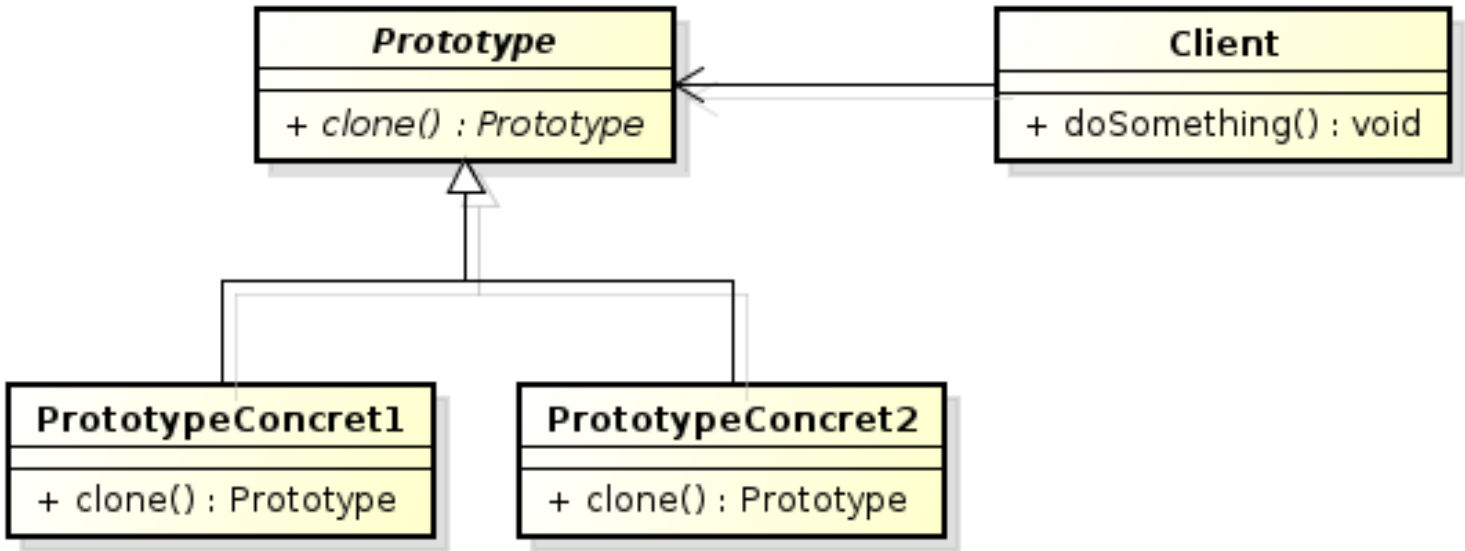
- Spécifier les types d'objets à créer en utilisant une instance prototype, puis créer de nouveaux objets en copiant cette instance
- Utiliser une instance d'une classe comme base de création des instances futures
- Éviter de laisser l'utilisateur faire le `new`

Prototype : indications d'utilisation

- Quand il faut créer un grand nombre d'instances similaires
- Quand la création de l'instance est complexe
- Quand la création de l'instance est consommatrice de ressources

  Ex : interrogation d'une base de données à la création

Prototype : structure



Prototype : structure

- Prototype
  - une classe abstraite (ou une interface)
  - impose la spécification d'une méthode `clone():Prototype`
- Classes dérivées
  - implémentent `clone()` par héritage ou surcharge
  - encapsulent l'usage de `new`
- Client
  - appel de la méthode `clone()` lorsqu'une instance est nécessaire
  - jamais d'utilisation de `new` pour le prototype

Prototype : conséquences

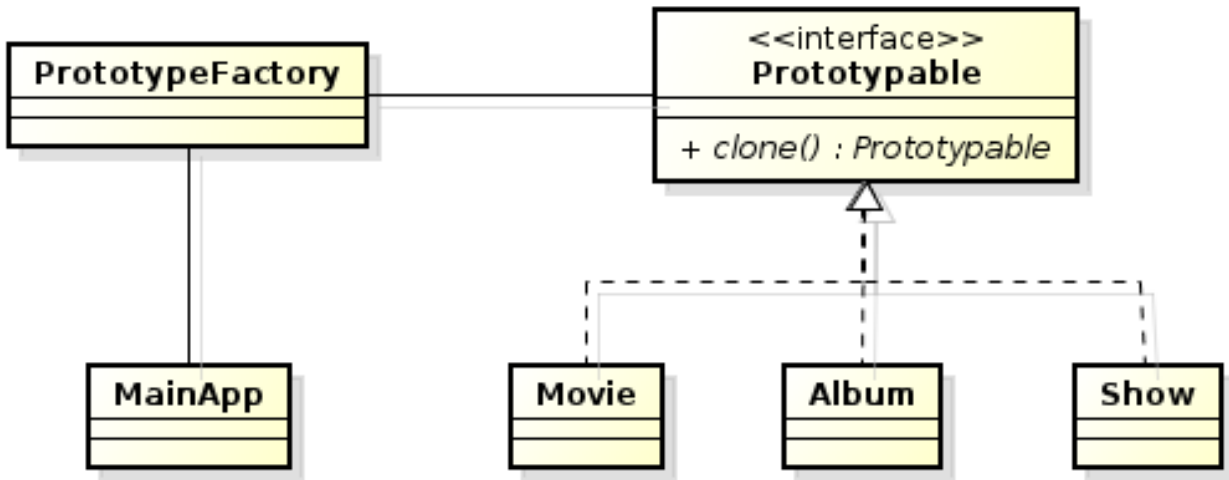
Avantages

- Meilleurs contrôle du processus de construction
- Gain de performance lors que la création est complexe ou coûteuse (par ex. si besoin requête en BdD)

Inconvénients

- Multiplication du nombre de classes

Prototype : exemple d'implémentation (1/4)



Prototype : exemple d'implémentation (2/4)

Application cliente : classe MainApp

```
public class MainApp {
  public static void main(String[] args) {
    ...
    Movie princessBride =
       PrototypeFactory.getInstance(ModelType.MOVIE);
    ...
    <code using princessBride>
    ...
  }
}
```

Le prototype abstrait : interface Prototypable

```
public interface Prototypable extends Cloneable {
    public Prototypable clone();
}
```

Prototype : exemple d'implémentation (3/4)

La fabrique d'instances : classe PrototypeFactory

```
public class PrototypeFactory {
  public static class ModelType {
    public static final String MOVIE = "movie";
    public static final String ALBUM = "album";
    public static final String SHOW = "show";
  }
  private static java.util.Map<String,Prototypable> prototypes =
          new java.util.HashMap<String,Prototypable>();
  static {
    prototypes.put(ModelType.MOVIE, new Movie());
    prototypes.put(ModelType.ALBUM, new Album());
    prototypes.put(ModelType.SHOW, new Show());
  }
  public static Prototypable getInstance(final String s) {
    return ((Prototypable) prototypes.get(s)).clone();
  }
}
```

Prototype : exemple d'implémentation (4/4)

Les prototypes concrets : classes Album, Movie, Show

```
public class Movie implements Prototypable {
  private String name = null;
  ...
  public Movie clone() {
    return (Movie) super.clone();
  }
}

public class Album implements Prototypable { ... }

public class Show implements Prototypable { ... }
```

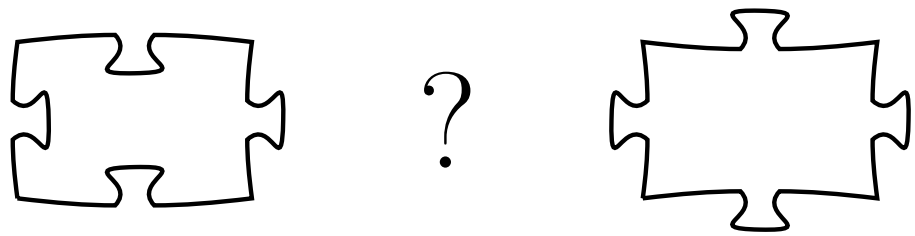
Prototype : résumé

- Simplification, optimisation de la création d'instances
- Souvent couplé au patron *Abstract Factory*
- Pas de `new` côté client
- `new` délégué à une méthode `clone()`

Adapter

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

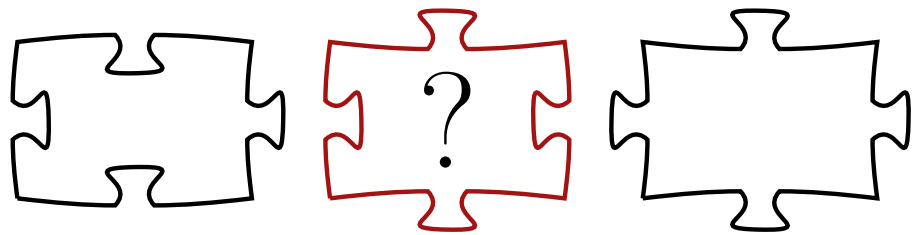
Problème



Client

Bibliothèque existante

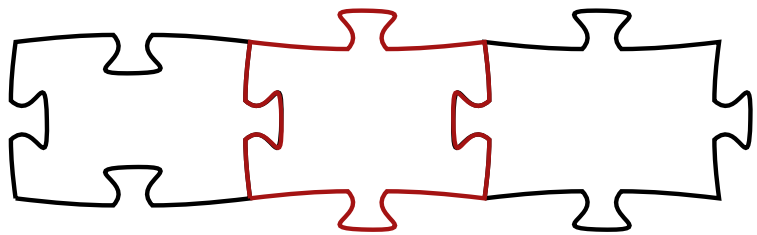
Problème



Client

Bibliothèque existante

Problème



Client

Bibliothèque existante

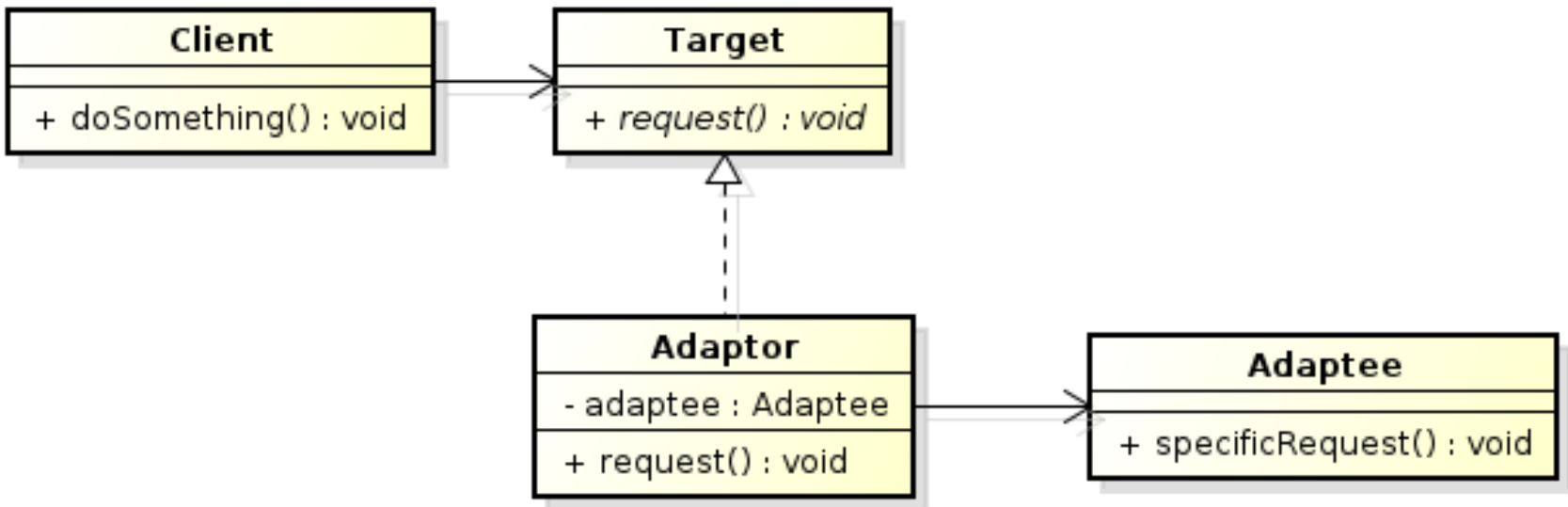
Adapter : intention

- convertir une interface d'une classe en celle attendue par le client
- assurer la compatibilité entre des éléments qui ne devraient pas fonctionner

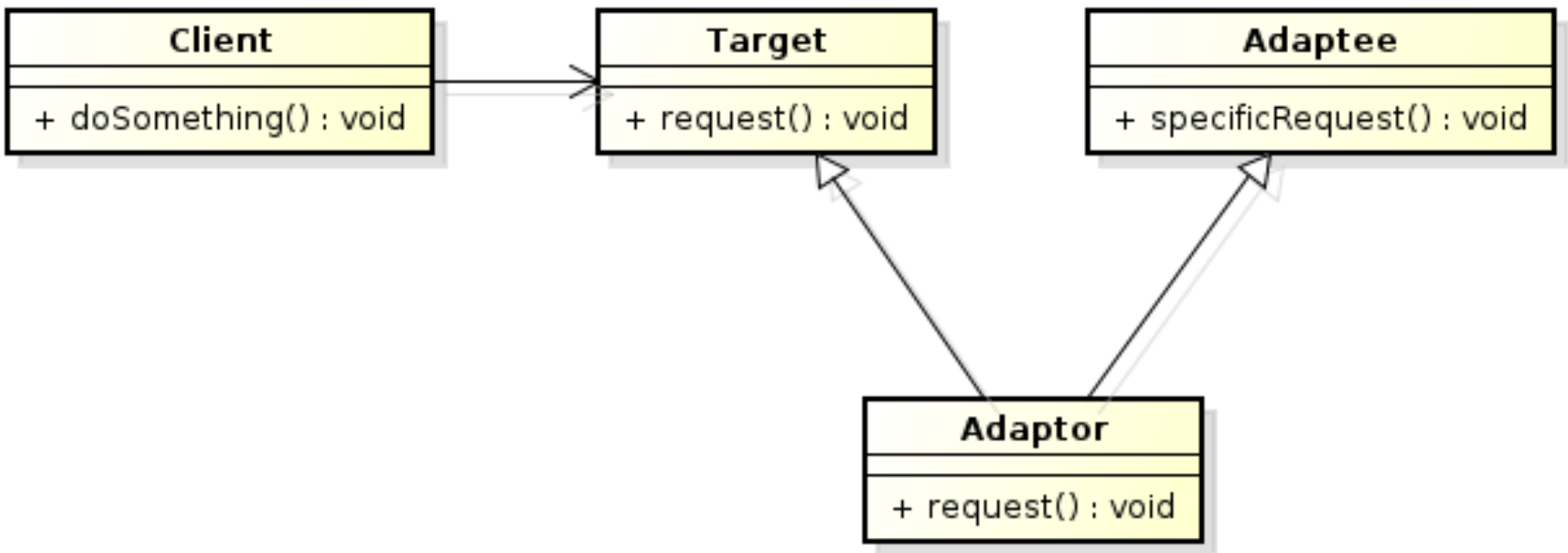
Adapter : indications d'utilisation

- Quand on doit intégrer une classe qu'on ne peut ou veut pas modifier
- Quand on veut éviter de redévelopper une bibliothèque ou d'écrire une variante de bibliothèque
- Quand on dispose d'une API tiers qui convient presque
- Quand on souhaite normaliser un code ancien

Object Adapter : structure



Class Adapter : structure



Adapter : structure

- Adaptee
  - la classe à adapter
  - on ne la modifie pas
- Target
  - l'interface du client
  - on ne la modifie pas
  - elle dicte l'adaptation (spécification de l'*Adaptor*)
- Adaptor
  - implémente l'interface client
  - effectue les appels à la bibliothèque à adapter

  $\Rightarrow$ traducteur *Target* $\leftrightarrow$ *Adaptee*

Adapter : conséquences

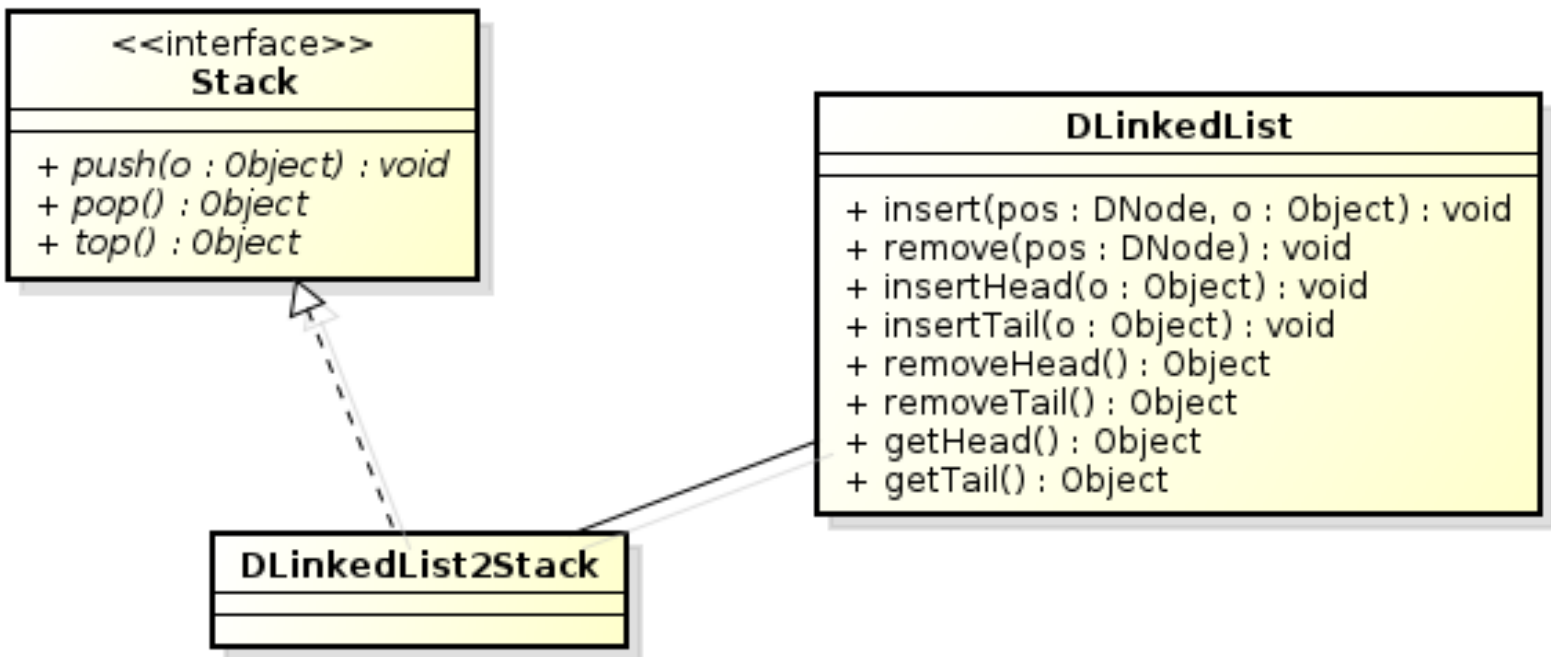
Avantages

- N'introduit qu'une classe
- Object Adapter
  - flexible (peu de code à ajouter pour déléguer à l'adapté)
  - peut adapter toute la hiérarchie de la classe adaptée
- Class Adapter
  - peut fonctionner avec plusieurs classes adaptées
  - pas besoin d'implémenter tout le comportement de l'adapté grâce à l'héritage
  - efficace (un seul objet)

Inconvénients

- Attention à la bidouille
- Object Adapter
  - redéfinition difficile des comportements des sous-classes de la classe adaptée
  - peu efficace (deux objets)
- Class Adapter
  - ne peut adapter que la classe explicitement adaptée

Adapter : exemple d'implémentation (1/5)



?

Adapter : exemple d'implémentation (2/5)

Target : interface Stack

```
interface Stack {
  void push(Object o);
  Object pop();
  Object top();
}
```

Adapter : exemple d'implémentation (3/5)

Adaptee : classe DLinkedList

```
class DLinkedList {
  public void insert (DNode pos, Object o) { ... }
  public void remove (DNode pos) { ... }
  public void insertHead (Object o) { ... }
  public void insertTail (Object o) { ... }
  public Object removeHead () { ... }
  public Object removeTail () { ... }
  public Object getHead () { ... }
  public Object getTail () { ... }
}
```

Adapter : exemple d'implémentation (4/5)

Adaptor : classe DLinkedList2Stack

```
class DLinkedList2Stack implements Stack {
  private DLinkedList dllist;
  public void push(Object o) {
    dllist.insertTail(o);
  }
  public Object pop() {
    return dllist.removeTail();
  }
  public Object top() {
    return dllist.getTail();
  }
}
```

Adapter : exemple d'implémentation (5/5)

Adaptor : classe DLinkedList2Stack (variante)

```
class DLinkedList2Stack extends DLinkedList implements Stack {

  public void push(Object o) {
    insertTail(o);
  }
  public Object pop() {
    return removeTail();
  }
  public Object top() {
    return getTail();
  }
}
```

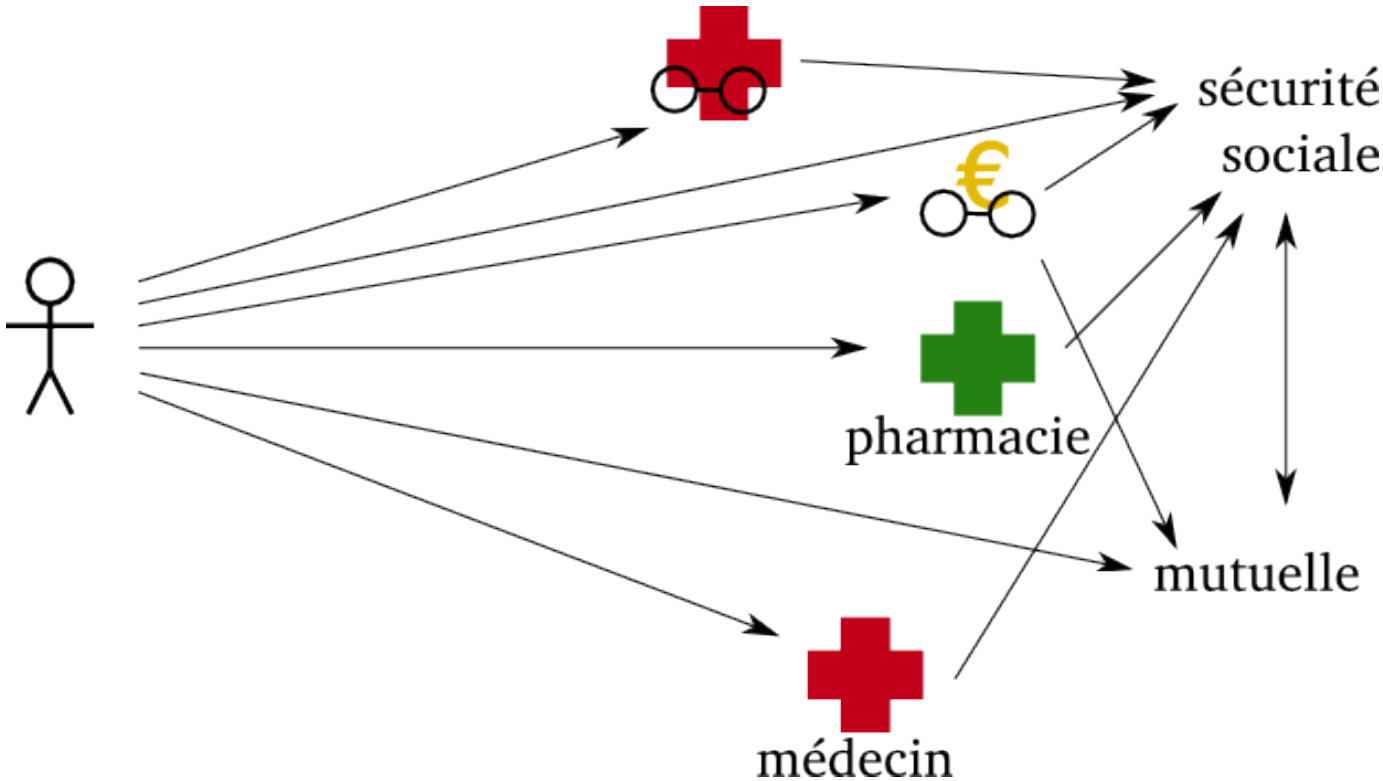
Adapter : résumé

- Réutilisation et adaptation de code
- Maintenance de code (maintien de la compatibilité avec des bibliothèques/API vieillissantes)
- Deux types : Object vs Class

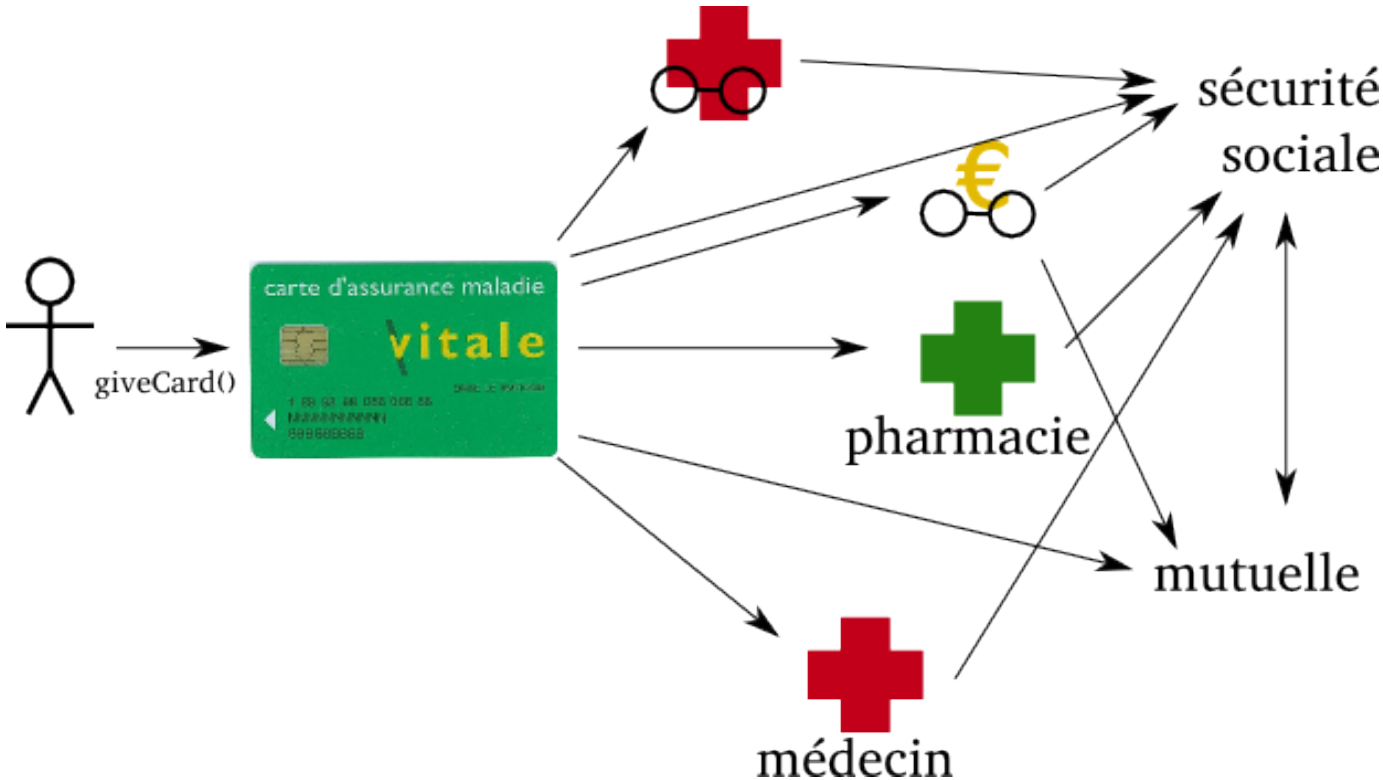
Façade

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

Problème



Problème


giveCard()
carte d'assurance maladie
vitale
sécurité
sociale
pharmacie
mutuelle
médecin

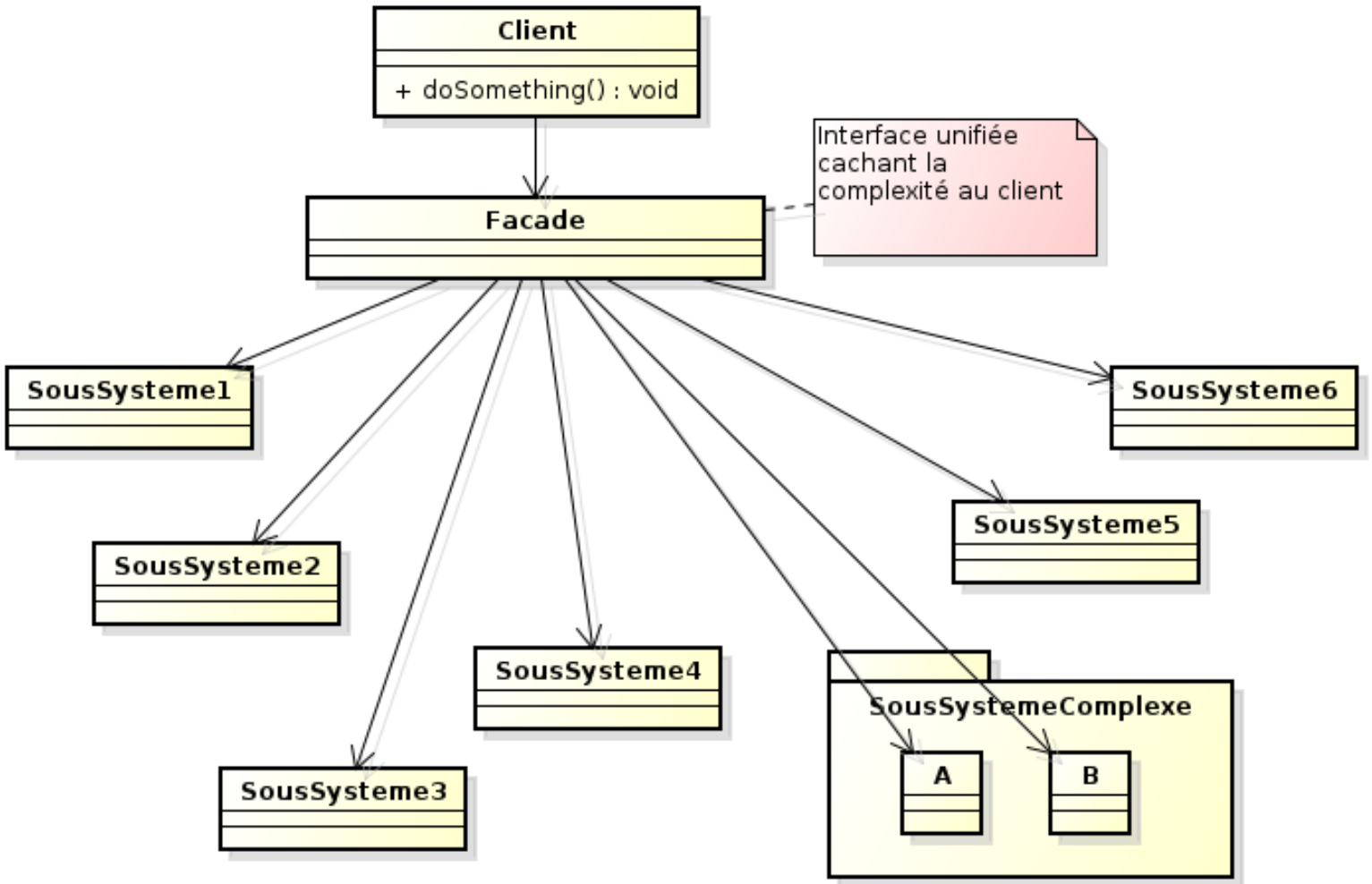
Façade : intention

- Fournir une interface unifiée pour un ensemble d'interfaces d'un sous-système
- Simplifier l'utilisation, la lisibilité et la compréhension de l'interface d'un système complexe

Façade : indications d'utilisation

- Quand on a plusieurs interfaces complexes et que l'on souhaite cacher la complexité ainsi que la multiplicité de ces interfaces au client
- Quand on souhaite faciliter l'utilisation, la lisibilité, la compréhension et les tests d'une bibliothèque
- Quand on souhaite réduire les dépendances entre les clients d'une bibliothèque et le fonctionnement interne
- Quand on souhaite nettoyer une API vieillissante ou mal conçue mais qu'on ne peut pas la modifier directement, augmenter son découpage, etc.

Façade : structure



Façade : structure

- Sous-systèmes
  - développés indépendamment ou non
  - multiples fonctionnalités
- Façade
  - utilise les sous-systèmes (encapsulation des appels)
  - fournit une interface au Client
- Client
  - ne voit qu'une interface
  - n'utilise pas directement les sous-systèmes

Façade : conséquences

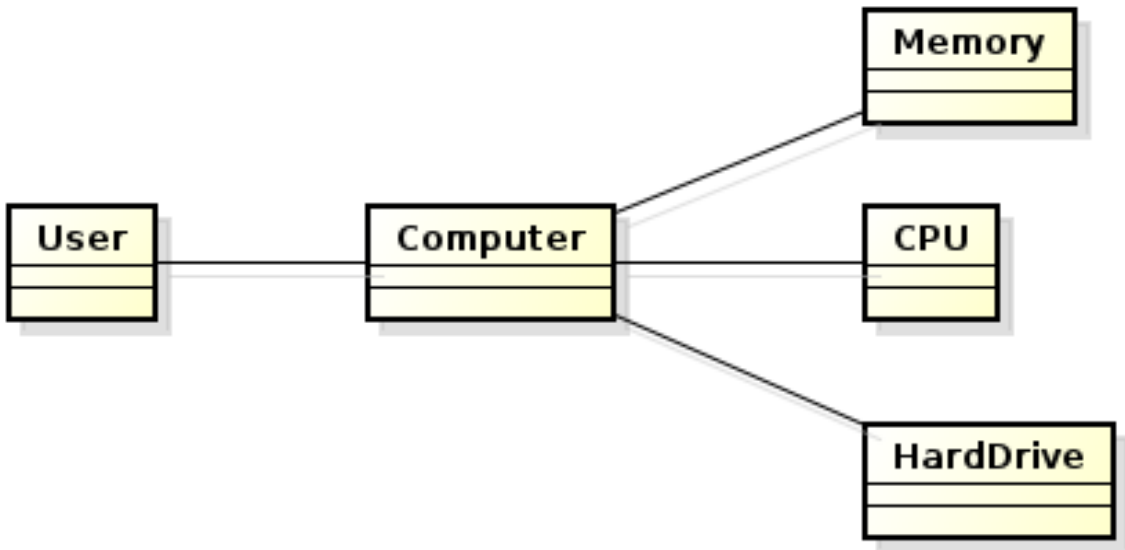
Avantages

- Couplage faible entre les classes et l'application
- Simplicité d'utilisation de sous-systèmes

Inconvénients

- Risques de pertes de fonctionnalités des sous-systèmes (dépend fortement de l'implémentation de la Façade)

Façade : exemple d'implémentation (1/4)



Façade : exemple d'implémentation (2/4)

Sous-systèmes complexes : classes Memory, CPU et HardDrive

```
class Memory {
  public void load(long position, byte[] data) { ... }
  ...
}

class CPU {
  public void freeze() { ... }
  public void jump(long position) { ... }
  public void execute() { ... }
  ...
}

class HardDrive {
  public byte[] read(long lba, int size) { ... }
  ...
}
```

Façade : exemple d'implémentation (3/4)

Façade : classe Computer

```
class Computer {
  private Memory ram;
  private CPU processor;
  private HardDrive hd;

  public Computer() {
    this.ram = new Memory();
    this.processor = new CPU();
    this.hd = new HardDrive();
  }

  public void start() {
    processor.freeze();
    ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
    processor.jump(BOOT_ADDRESS);
    processor.execute();
  }
```

Façade : exemple d'implémentation (4/4)

Application cliente : classe User

```
class User {
  public static void main(String[] args) {
    Computer computer = new Computer();
    computer.start();
  }
}
```

Façade : résumé

- Simplification de l'interface client
- Accès à des sous-systèmes complexes délégués à la Façade
- Découplage de l'application et des sous-systèmes

Conclusion

- Data Access Object
  - persistance indépendamment du support
  - couplage avec Factory
- Prototype
  - utiliser une instance pour créer les futures instances
  - l'application client ne fait jamais `new` mais `clone()`
- Adapter
  - traducteur *interface client* $\leftrightarrow$ *code existant*
  - Object vs Class
- Façade
  - interface unifiée et simplifiée au client
  - application indépendante de l'implémentation des sous-systèmes

Questions ?