Planification d'un *Sprint*

- Confirmer les stories prêtes
  - ✓Définition de prêt et fini
- Evaluer la nombre de points d'une story
  - ✓Ponts de récit ou journée idéale (homme/jour)
- Organisation de l'essaimage (plusieurs stories en parallèle, répartition des ressources)
- Décomposition des stories en tâches
- Affectation des tâches aux développeurs

Exécution d'un *sprint*

- Conception, réalisation et test des *stories*
- Organisation, affectation des tâches
- Inclut les *sprints* journaliers

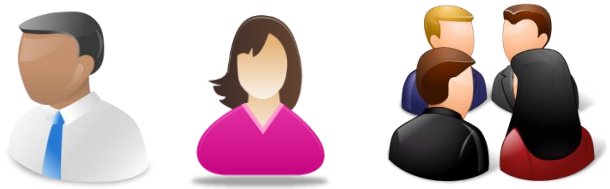
La mêlée quotidienne

- Courte : ~15mn
- Bilan: mise à jour du tableau de la story
  - Qu'est-ce que j'ai fait hier ?
  - Qu'est que je vais faire aujourd'hui ?
  - Quels sont les obstacles que j'ai rencontrés ?
- Objectif :
  - Rythmer le sprint (stories finies, prêtes)
  - Recenser les obstacles

Revue d'un *sprint*

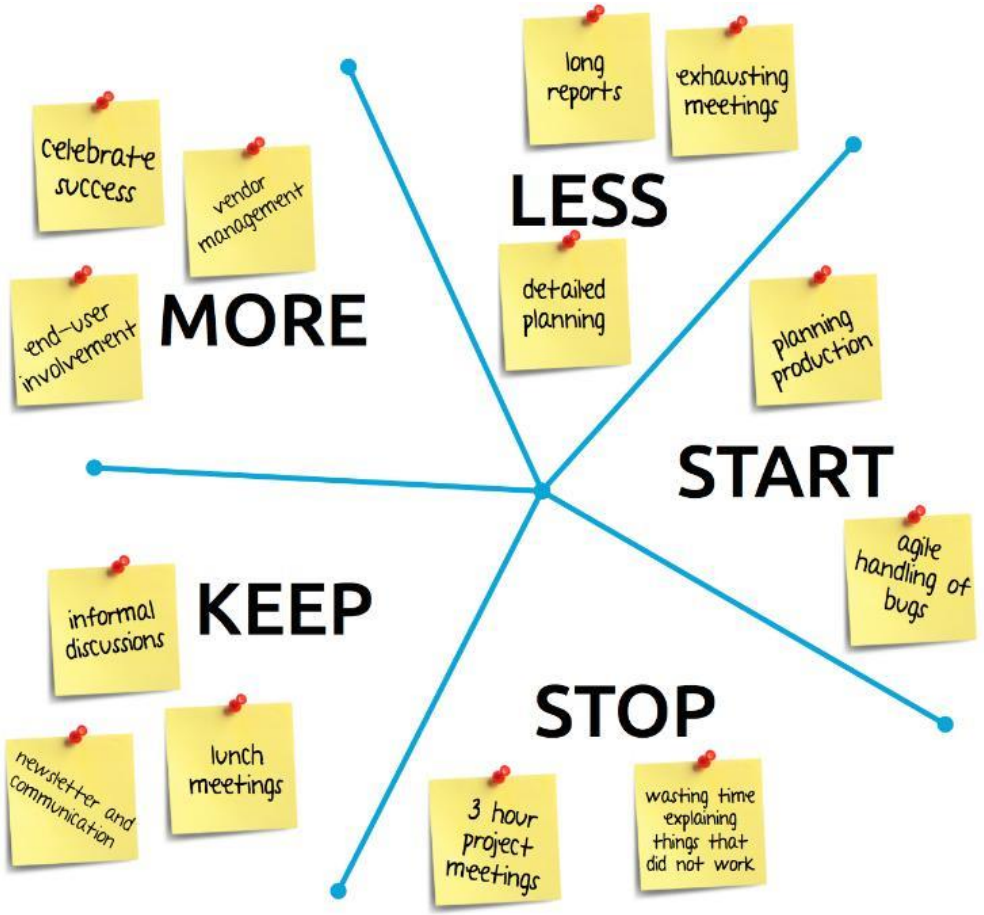
- Démonstration de chaque story finie
- Collecte du feedback
- Evaluation du niveau de réalisation de l'objectif
- Evaluation de l'impact du travail réalisé et décision d'une release (livraison) ou pas

Rétrospective d'un *sprint*



- Collecter les information sur le sprint passé par rapport à la pratique SCRUM
  - ✓ Ce qui c'est bien passé, moins bien passé
- Identifier les choses à améliorer
- Décider d'améliorer certaines choses
- Combinée à la revue, de courte durée

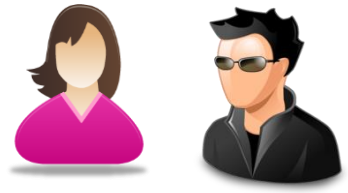
Rétrospective de sprint : « Starfish »



Solidification d'une *release*

- Tests de qualité de services (performances, coût, sécurité …)
- Documentation
- …

Validation d’une release

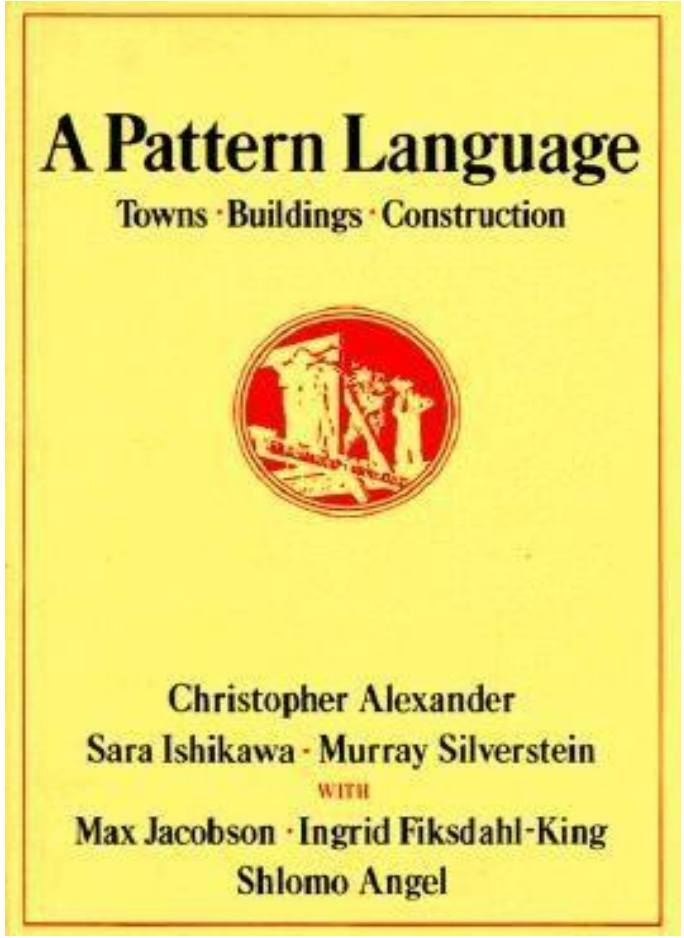


- Test d’usage avec le client, des utilisateurs

Design Patterns

Histoire des patterns

- Travaux de l'architecte (génie civil) Christopher Alexander.
- Manières d'améliorer le processus de conception des bâtiments et des zones urbaines.
- Propose des patterns en décrivant le problème de chacun ainsi que sa solution.
- Pattern est une Solution à un problème dans un contexte donné.
- Utiliser dans plusieurs domaines y compris le génie logiciel.



Patterns dans le Génie Logiciel

- L'approche orientée objet vise à bâtir des applications à base des composants simples et réutilisables.
  - ✓ Cette approche peut vite devenir un piège lorsque le découpage s'effectue sans règles précises (programmation à la volée).
  - ✓ Concepteur dépassé par la complexité du codage (effet spaghetti)
- Difficile à maintenir, corriger les bugs, faire évoluer…

Solution: exploiter un niveau supplémentaire dans la conception objet : Les modèles de conception ou design patterns

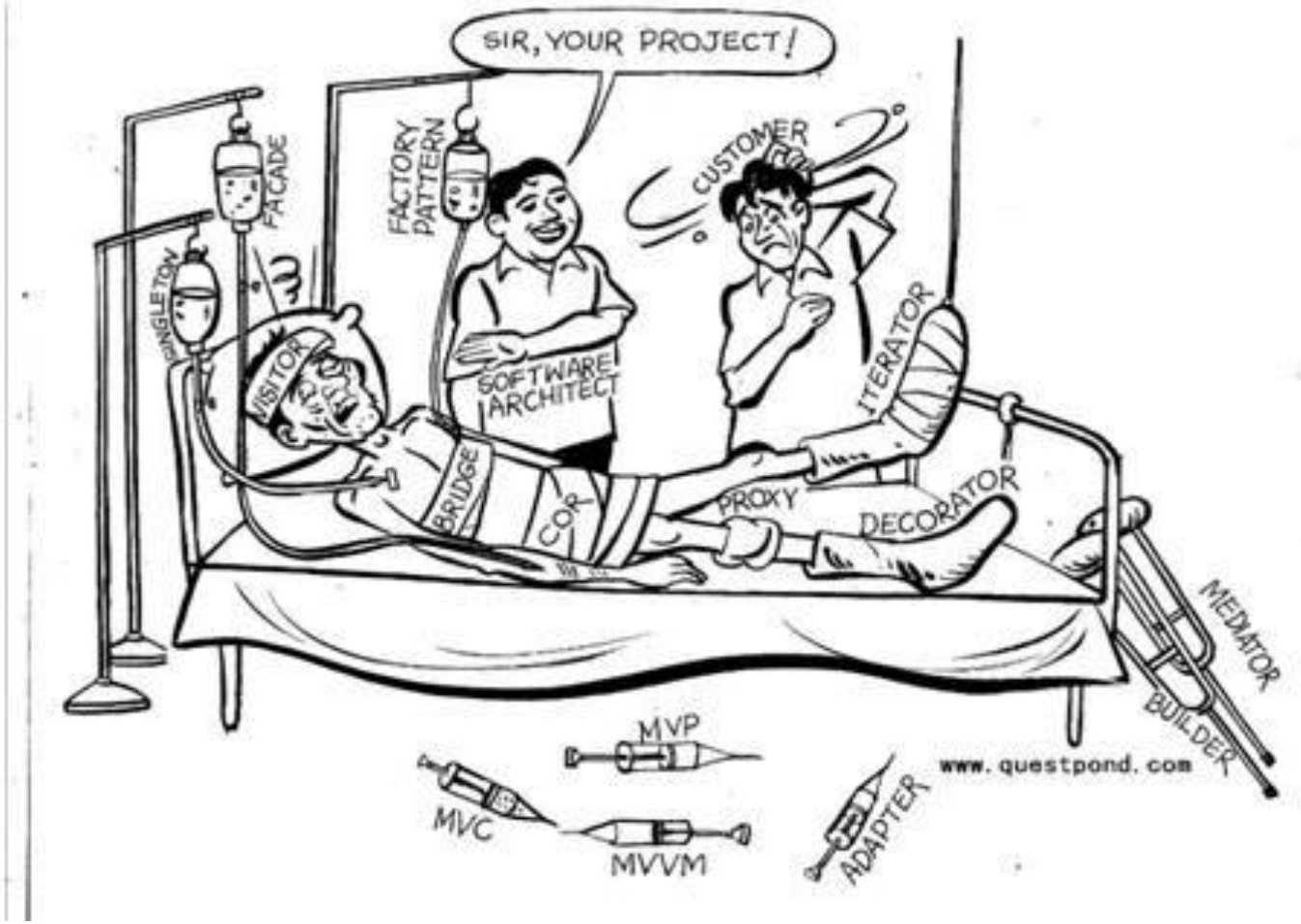
Historique des design patterns

- 1987 - Cunningham et Beck utilisent les idées d'Alexander pour
  - ✓ développer un petit langage de patrons pour Smalltalk
- 1990 – Le Gang des 4 (« Gang of Four » : Gamma, Helm, Johnson and Vlissides) commence à travailler à la compilation d'un catalogue de patrons de conceptions
- 1991 - Bruce Anderson donne le premier workshop de patterns au OOPSLA
- 1993 - Kent Beck et Grady Booch sponsorisent la première réunion de ce qui est maintenant connu comme le groupe Hillside (http://www.hillside.net/)
- 1995 - Le Gang des 4 (GoF) publie le livre des Design patterns

Quand doit-on utiliser les DP?

- **Mythe:** Pour une bonne architecture tout les design patterns doit être implémenté dans un projet

- **Fait:** Les modèles émergent naturellement et sont complètement à la demande.



Quand doit-on utiliser les DP?

- "Concevoir un logiciel orienté objet est difficile, et concevoir un logiciel orienté objet réutilisable est encore difficile" –Erich Gamma.

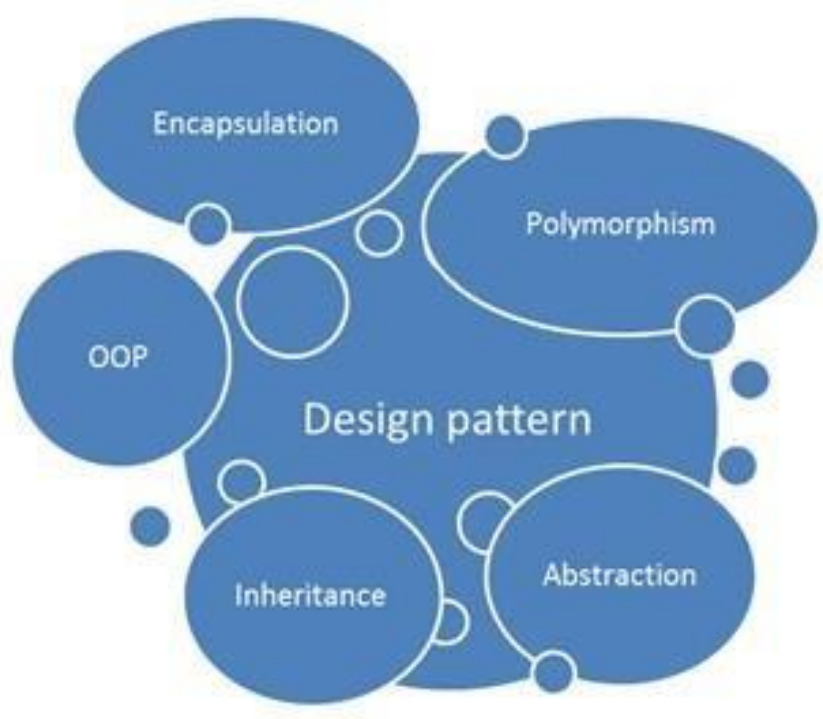
- Apprendre à développer un bon logiciel est similaire à apprendre à bien jouer au échec.

Design patterns VS Architecture pattern VS Architecture Style

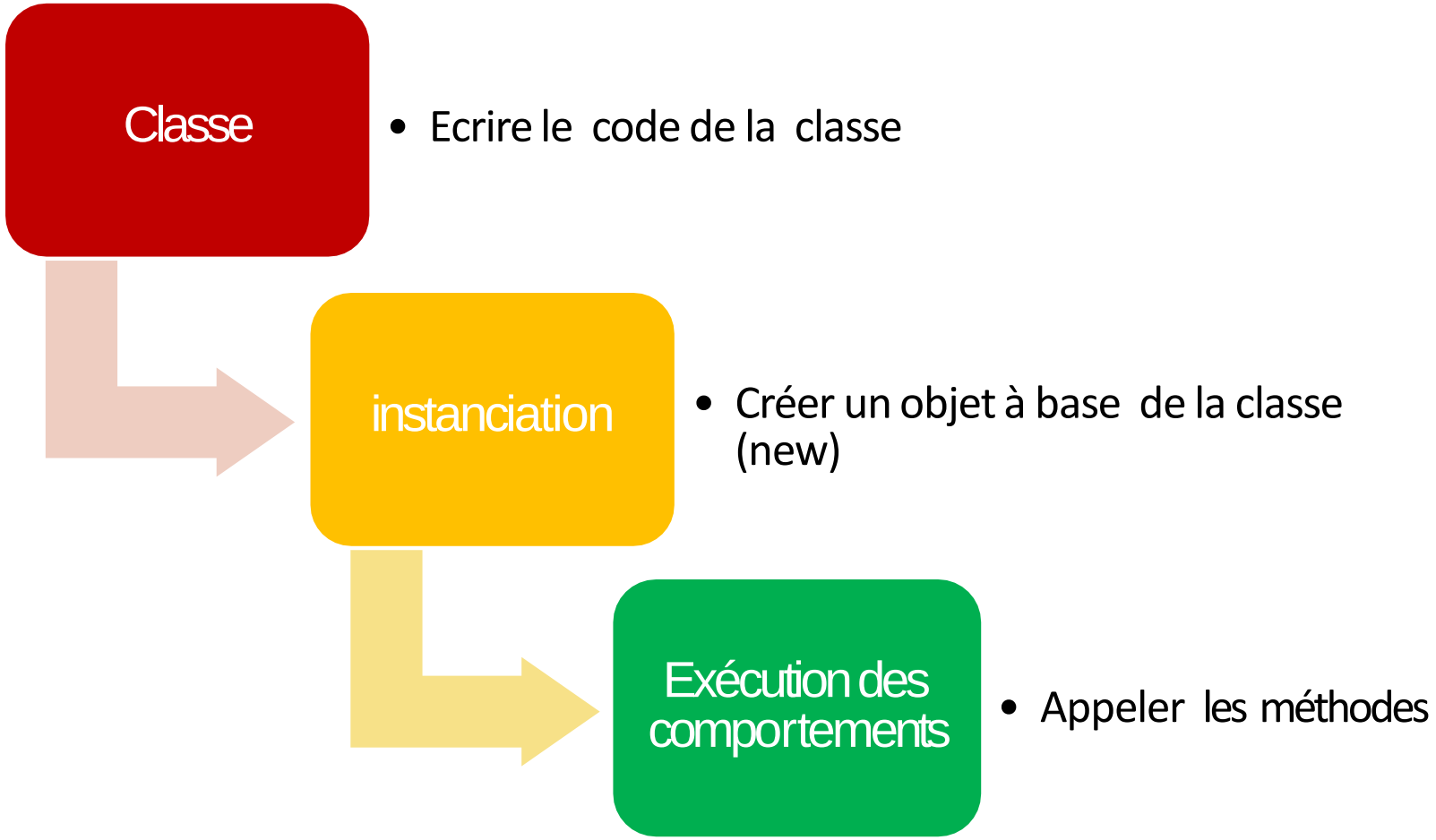
Design patterns	Architecture pattern	Architecture Style
Niveau code	Niveau composant	Niveau d'abstraction élevée (théorie)
Résoudre des problèmes récurrents au niveau code	résoudre des problèmes récurrents d'architecture	Spécifier la façon avec laquelle un système sera créer ou va fonctionner
if someone says "X" is a Design Pattern, Expect code	if someone says "Y" is an Architecture Pattern expect some kind of component level block diagram	If some one says "Z" is an Architecture Style expect a theoretical aspect
Singleton, Adapter	**MVC, MVP**	**SOA, Rest, Client/serveur**

Les DP comme solution

- Les Design Patterns sont des solutions testées de plusieurs problèmes fréquemment rencontrés lors de la phase de conception d'applications (POO).
- Bénéfices
  - ✓ Fiables et réutilisable
  - ✓ facilement compréhensible et identifiable,
  - ✓ Améliorer la communication et la collaboration entre développeurs.



Phase de la POO

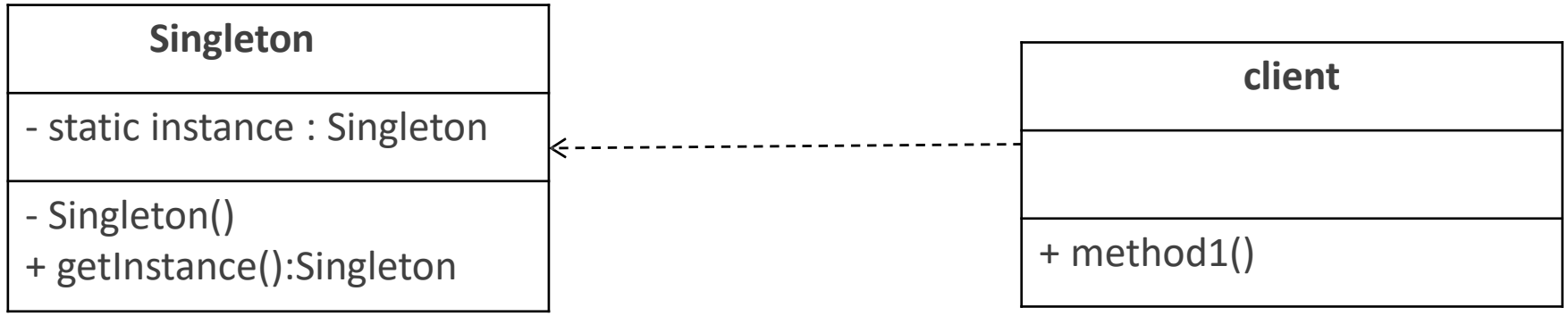


Classification des design patterns

- le GOF a défini 23 design patterns fondamentaux regroupés en 3 catégories:
  - ✓ Les Patterns de création
    - Les DP aident à créer des objets pour vous, au lieu d'avoir à instancier les objets directement.
    - Abstract Factory, Builder, Factory Method, Prototype, Singleton
  - ✓ Les Patterns de structure
    - Les DP de structure aident à composer des groupes d'objets en des structures plus larges, telles que des interfaces utilisateur complexes.
    - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
  - ✓ Les patterns de comportement
    - Les DP de comportement aident à définir la communication entre les objets du système et définir comment le flux est contrôlé.
    - Chain of Resp., Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Patterns de création

- **Singleton**
  - Problème :
    - Limiter l'instanciation des objets à un nombre réduit (une seule Fois : GOF).
  - Solution :



Vous pouvez utiliser le pattern Singleton dans le cas d'un objet de connexion à une base de données

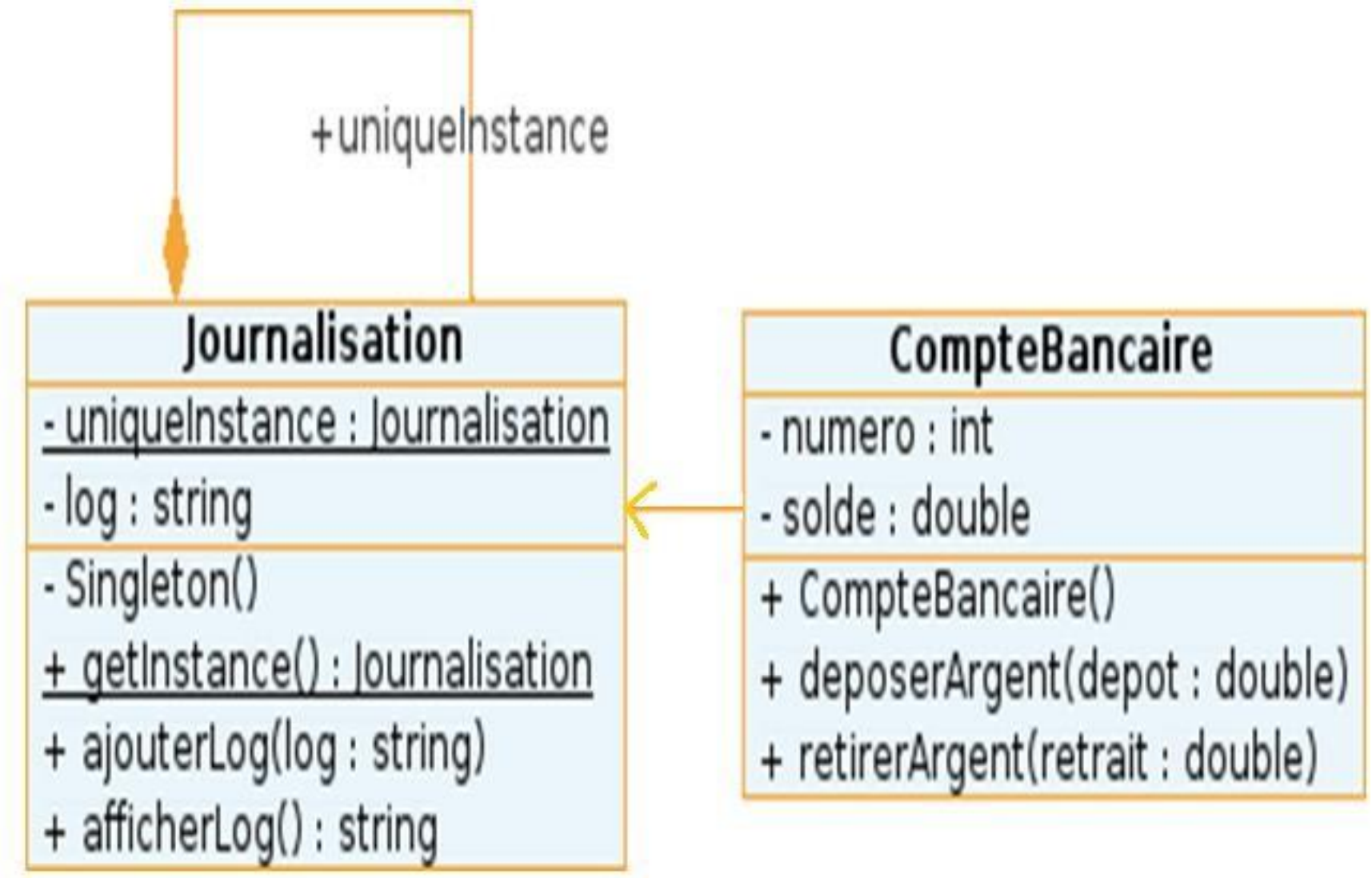
Exemple

```
public class Singleton {
  private static Singleton instance;
  private Singleton() {
      instance = new Singleton();
  }
  public static Singleton getInstance() {
    return instance;
  }
}
```

```
public class Client {
public static void main(String[] args) {
   //Singleton s = new Singleton();
   impossible  Singleton s1 =
   Singleton.getInstance();  Singleton s2 =
   Singleton.getInstance();
   System.out.println(s1);
   System.out.println(s2);
 }

}
```

Exemple


+uniqueInstance
Journalisation
- uniqueInstance : Journalisation
- log : string
- Singleton()
+ getInstance() : Journalisation
+ ajouterLog(log : string)
+ afficherLog() : string
CompteBancaire
- numero : int
- solde : double
+ CompteBancaire()
+ deposerArgent(depot : double)
+ retirerArgent(retrait : double)

Patterns de création

- Fabrique (Factory):

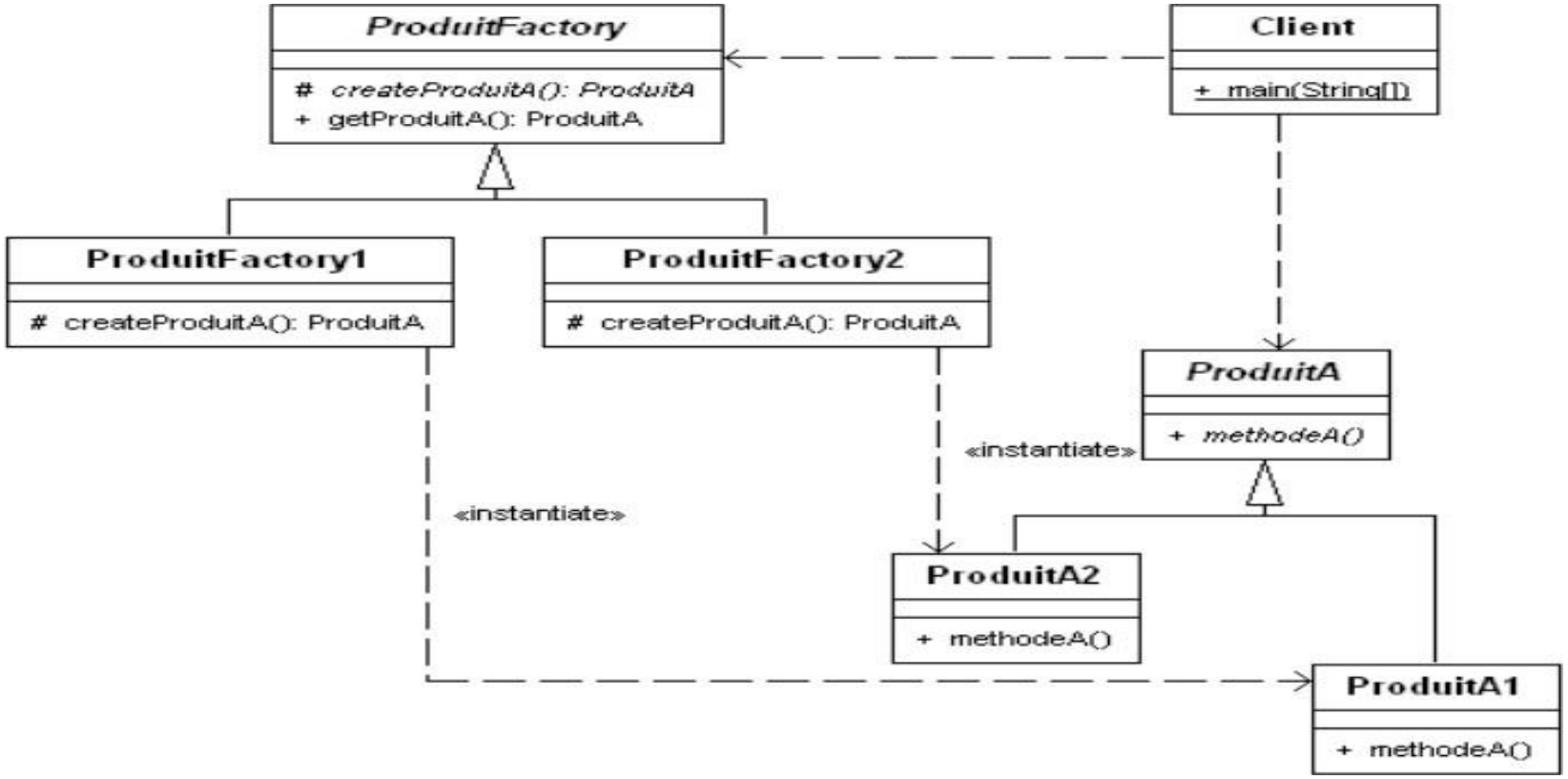
La fabrique permet de créer un objet dont le type dépend du contexte :

cet objet fait partie d'un ensemble de sous-classes. L'objet retourné par la fabrique est donc toujours du type de la classe mère mais grâce au polymorphisme les traitements exécutés sont ceux de l'instance créée.

- ✓ Il est utilisable lorsque :
  - Le client ne peut déterminer le type d'objet à créer qu'à l'exécution
  - Il y a une volonté de centraliser la création des objets
- ✓ Ce design pattern peut être implémenté sous deux formes principales sont :
  - Déclarer la fabrique abstraite et laisser une de ses sous-classes créer l'objet
  - Déclarer une fabrique dont la méthode de création de l'objet attend les données nécessaires pour déterminer le type de l'objet à instancier

Patterns de création

- Il est possible d'implémenter la fabrique sous la forme d'une classe abstraite et de définir des sous-classes chargées de réaliser les différentes instanciations.



Patterns de création

Fabrique abstraite (abstract Factory):

Le motif de conception Abstract Factory (fabrique abstraite) permet de fournir une interface unique pour instancier des objets d'une même famille sans avoir à connaître les classes à instancier.

✓ **Il est utilisable lorsque** :

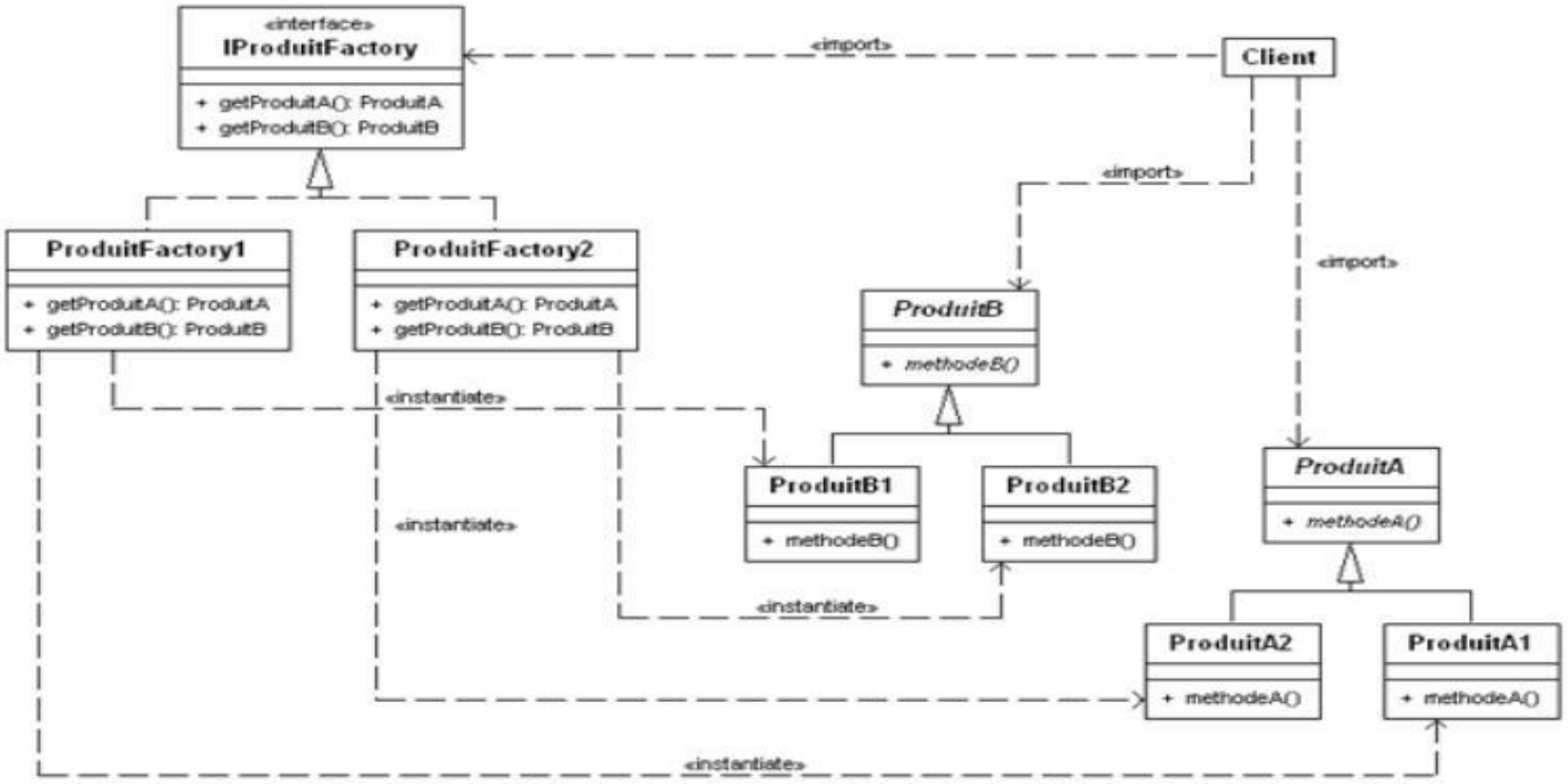
- Le système doit être indépendant de la création des objets qu'il utilise
- Le système doit être capable de créer des objets d'une même famille

✓ **Le principal avantage de ce motif de conception est**

- d'isoler la création des objets retournés par la fabrique.
- L'utilisation d'une fabrique abstraite permet de remplacer facilement une fabrique par une autre selon les besoins

Patterns de création

- Le motif de conception fabrique abstraite peut être interprété et mis en œuvre de différentes façons.
- Le diagramme UML ci-dessous propose une mise en œuvre possible avec deux familles de deux produits.



Patterns de comportement

- **Stratégie (Strategy):**

- Le pattern de comportement stratégie est a pour but d'adapter le comportement et les algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec les clients de cet objet

  - Il est utilisable lorsque :

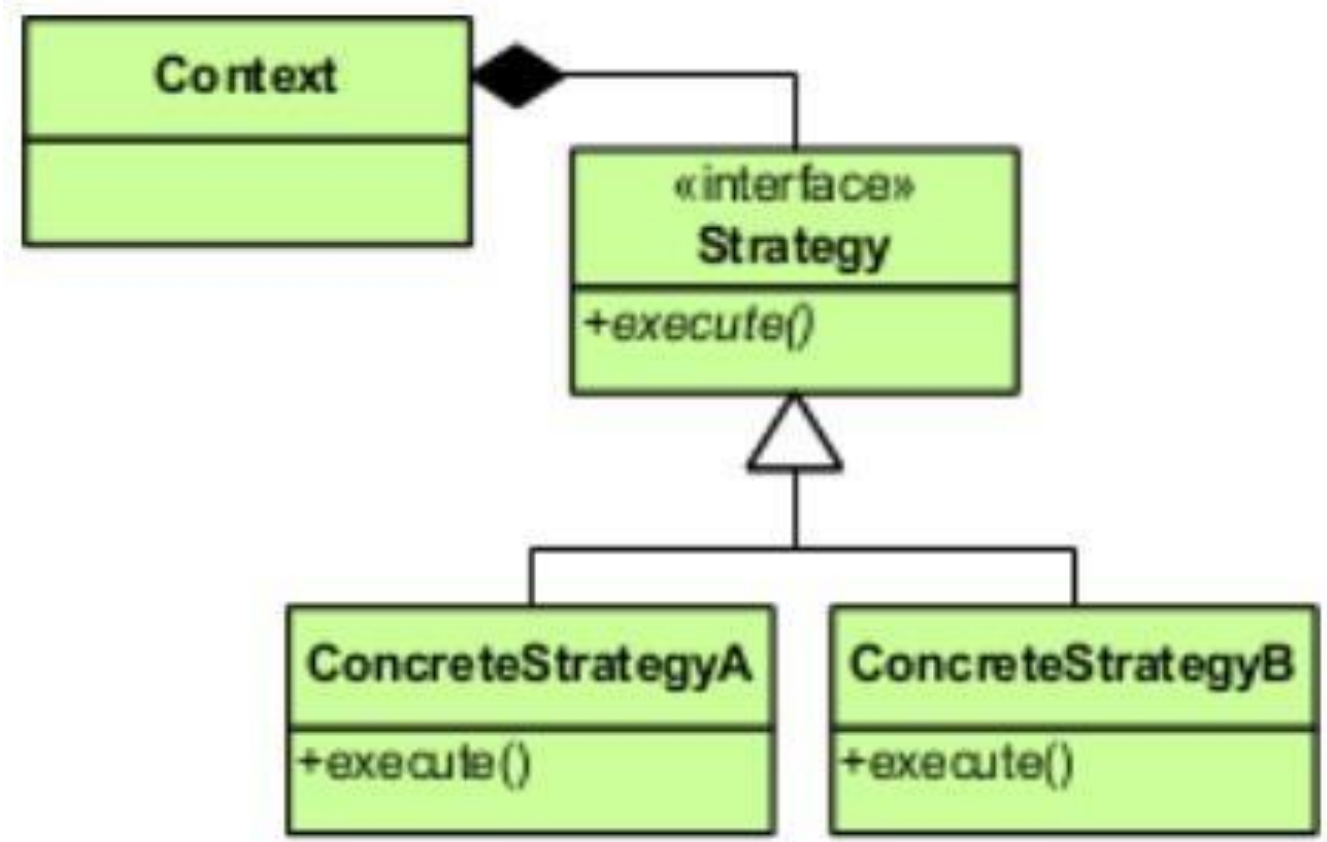
    - ✓ il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application.
    - ✓ Adapte le comportement et les algorithmes d'un objet en fonction d'un besoin, sans changer les interactions avec les clients de cet objet

  - Le principal avantage de ce motif est :

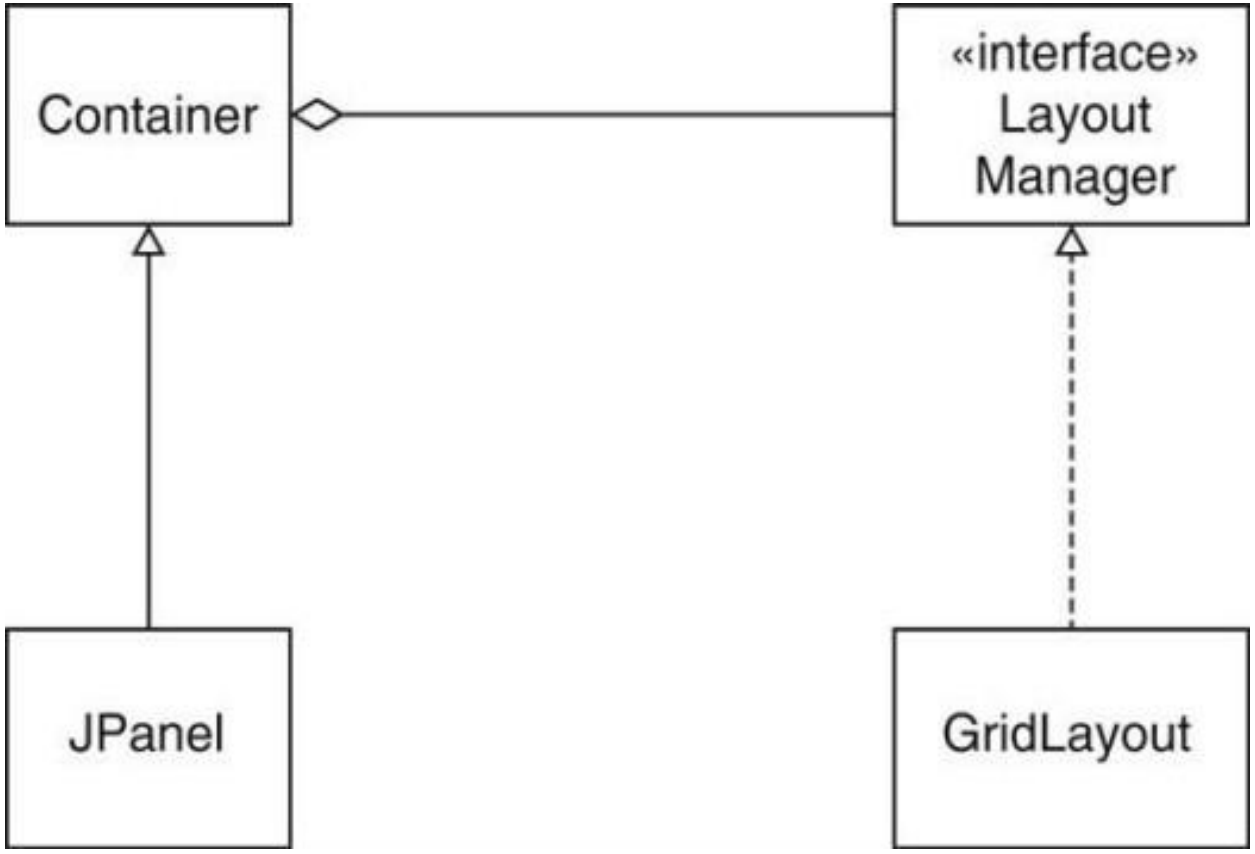
    - ✓ De fournir le moyen de définir une famille d'algorithmes, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables.
    - ✓ Ce patron laisse les algorithmes changer indépendamment des clients qui les emploient.

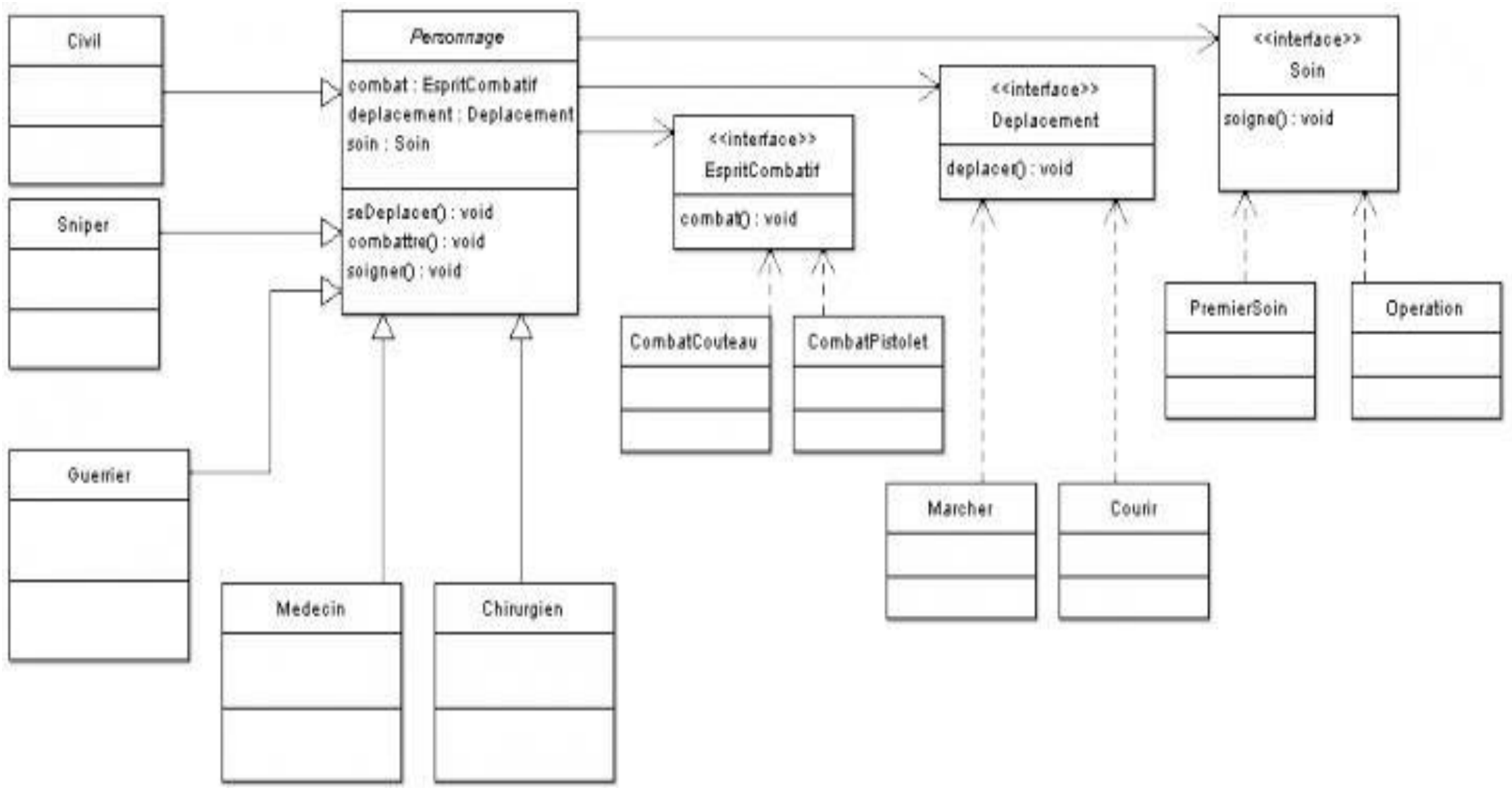
Patterns de comportement

- Il est possible d'implémenter le patron stratégie sous la forme suivante:



Exemple :

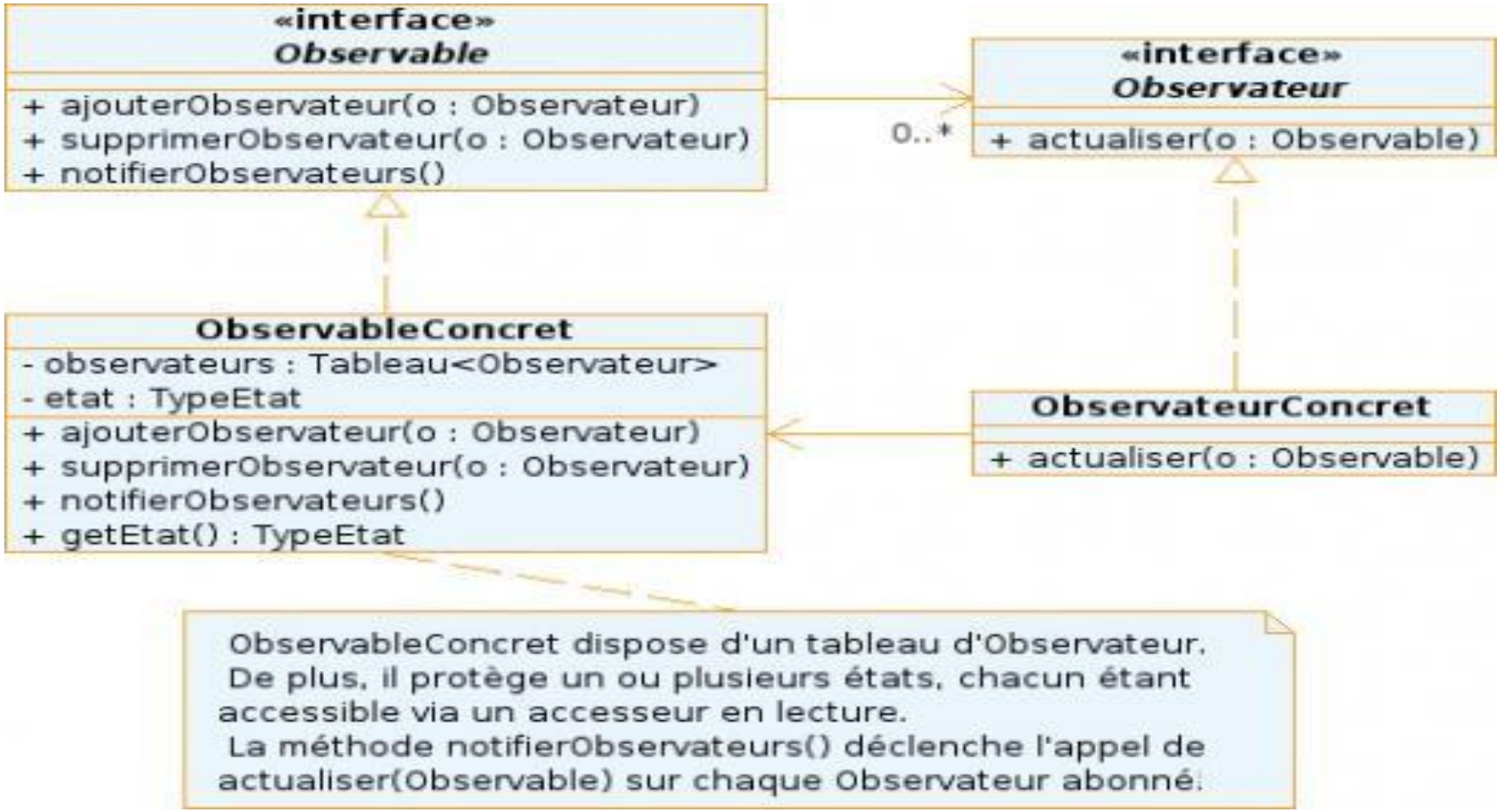



Civil
Personnage
combat : EspritCombatif
deplacement : Deplacement
soin : Soin
seDeplacer() : void
combattre() : void
soigner() : void
<<interface>>
Soin
soigne() : void
<<interface>>
Deplacement
deplacer() : void
<<interface>>
EspritCombatif
combat() : void
Sniper
Guerrier
Medecin
Chirurgien
CombatCouteau
CombatPistolet
Marcher
Courir
PremierSoin
Operation

Patterns de comportement

- **Observer :**
  - Le pattern de comportement Observer permet de construit une dépendance entre un sujet et des observateurs de façon à ce que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.
    - ✓ Il est utilisable lorsque:
      - Construit une dépendance entre un objet et des observateurs
      - Pour que chaque modification de l'objet soit notifiée aux observateurs
      - Afin qu'ils puissent mettre à jour leurs états
      - Souvent utilisé pour les IHM (modèle événementiel en Java)
    - ✓ Le principal avantage de ce motif est :
      - Il permet de construire une dépendance entre un sujet et des observateurs de sorte que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.

le diagramme UML du pattern Observateur



Exercice sur les DP

- Préparer un document qui va contenir les 23 DP du GOF (problème et solution sous forme de diagramme de classes)
- Créer un projet java qui contiendra les 23 exemples.
- A vérifier et à rendre la semaine prochaine