Patrons de conception

- A.k.a. « Design Patterns »

  Un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.
  Wikipedia

- Omniprésents dans les logiciels commerciaux
  - Par exemple : modèle-vue-contrôleur pour les UI, le Web

- Concept issu du livre « Gang of 4 » ou « GoF »

  Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. ISBN 0-201-63361-2.

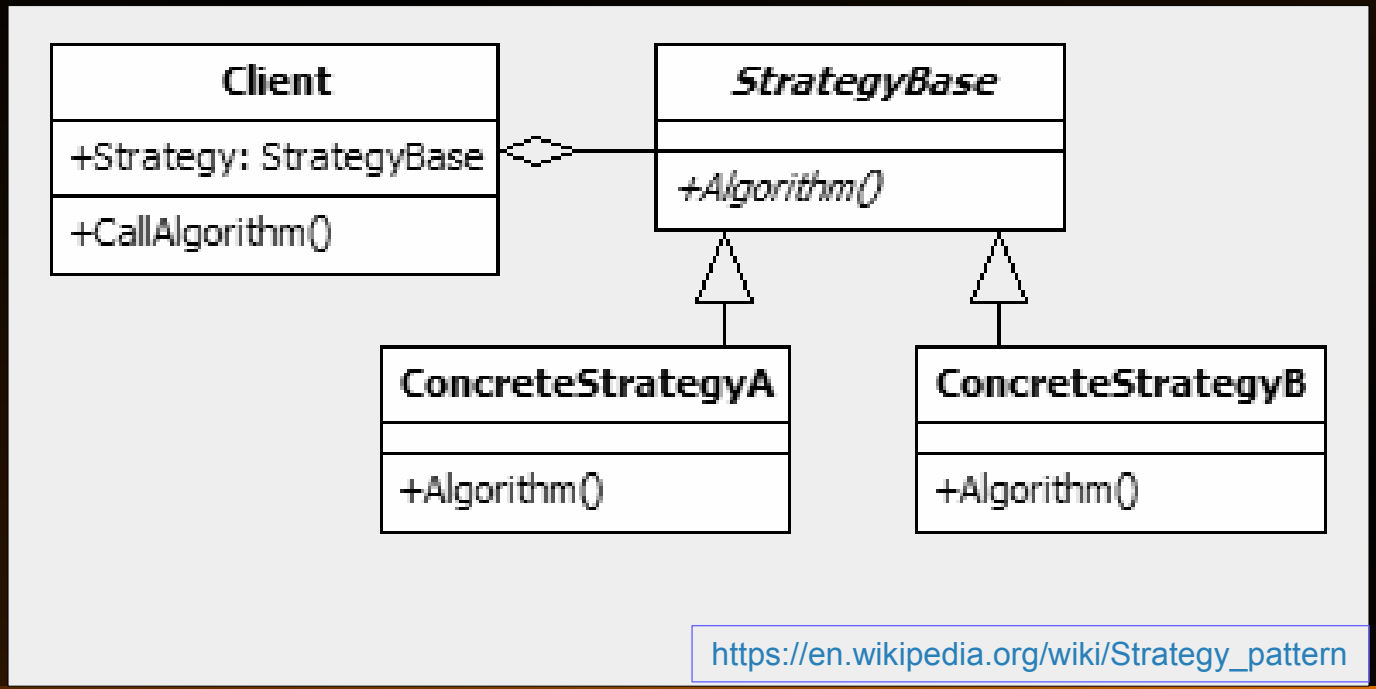
Patrons de conception

- Conçus pour la programmation orientée objet
  - Modèles de classe UML
- Plusieurs catégories
  - Patrons de création :
    - Création, initialisation d'objets
    - Par exemple : *Factory*, *Singleton*, ...
  - Patrons d'architecture :
    - Organisation des parties d'un programme
    - Par exemple : *Proxy*, *Adapter*, ...
  - Patrons comportementaux :
    - Communication entre les parties d'un programme
    - Par exemple : *Iterator*, *Observer, ...*

« Strategy »

"Allows a set of similar algorithms to be defined and encapsulated in their own classes. The algorithm to be used for a particular purpose may then be selected at run-time according to your requirements."

GoF



https://en.wikipedia.org/wiki/Strategy_pattern

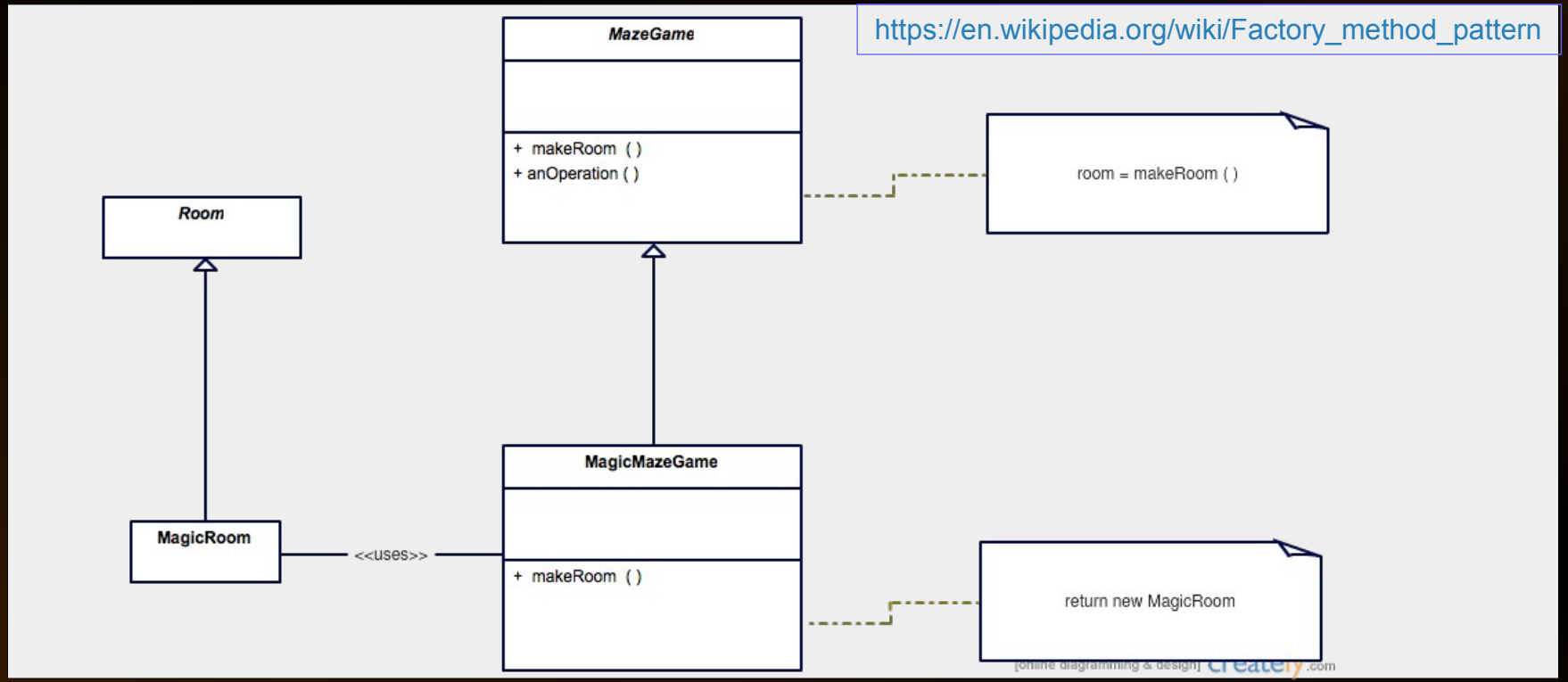
« Strategy » Utilisation

- Exemple dans une simulation numérique
  - Plusieurs implémentations possibles
    - Différents schémas numériques (FDM, FEM, …)
    - Différentes architectures (CPU, GPU, …)
  - Mais aussi : choix de la stratégie de stockage
    - HDF5, ASCII, en mémoire, envoi via MPI…
- Mais attention à la granularité
  - **/!** Surcoût appel fonction virtuelle (~10ns)
  - Sur un point (0,1…1ns) => **NON**
  - Sur un tableau => **OK**

« Factory method »

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses."

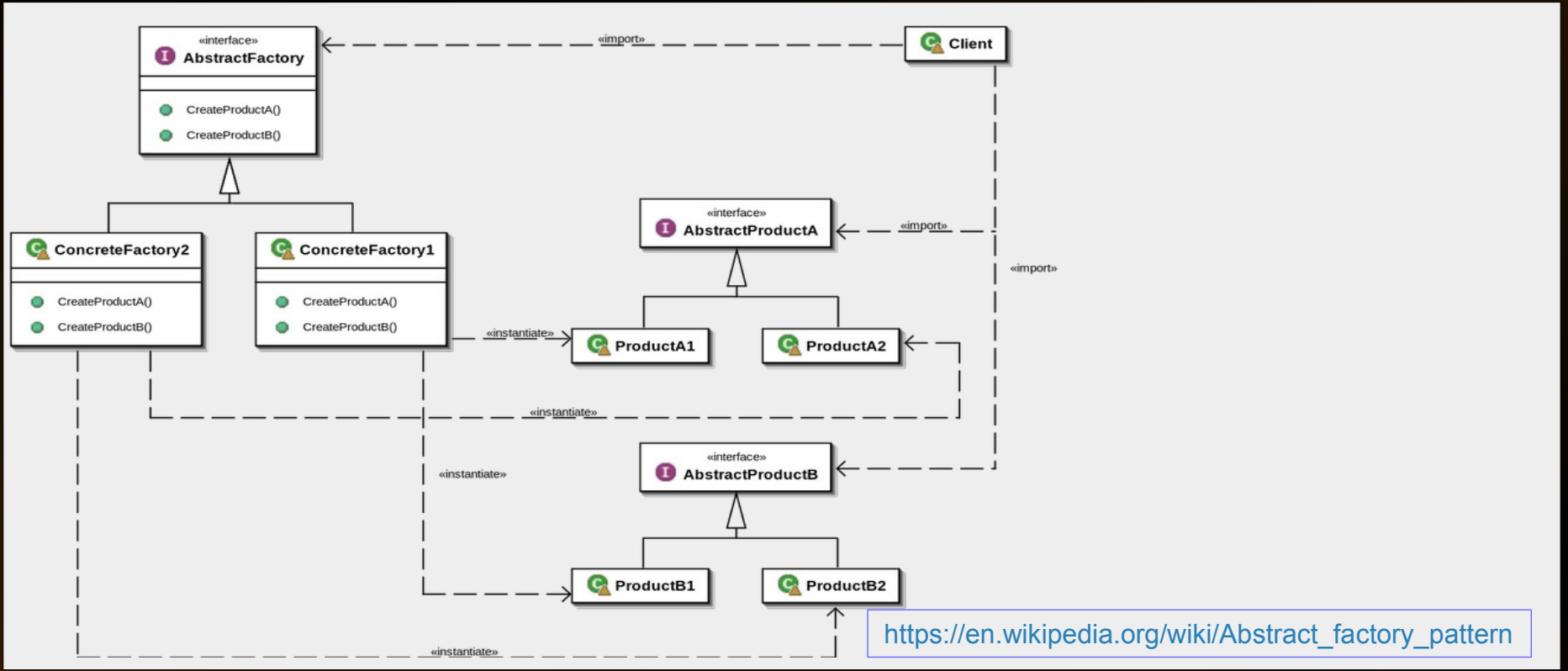
GoF



« Abstract factory »

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

GoF



« Factory » Utilisation (1/2)

- Combinaison avec « Strategy » intéressante
  - Construction de l’objet stratégie à utiliser
- Exemple simulation numérique
  - Construction de la stratégie de stockage
  - Peut varier en fonction
    - De la machine
    - Du pas de temps
    - …
  - Possibilité avec *abstract factory* :
    - chargement depuis un fichier de configuration

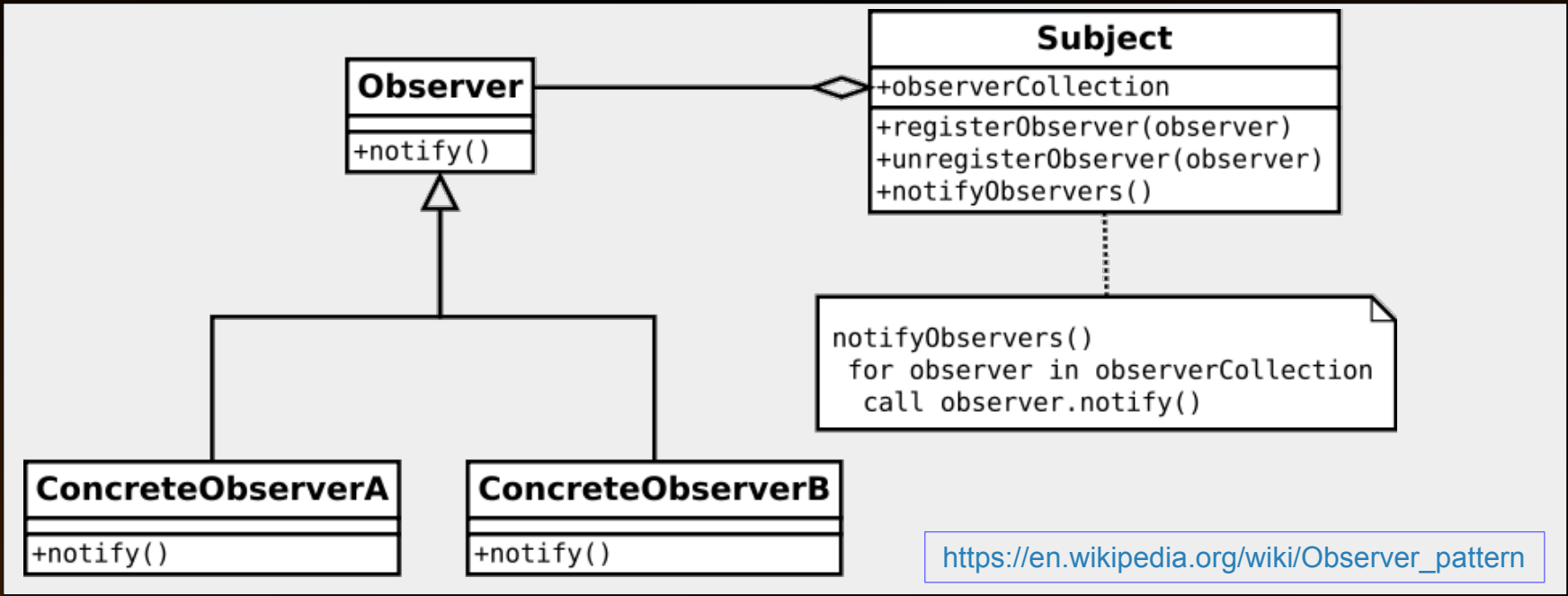
« Factory » Utilisation (2/2)

- Attention à ne pas en abuser
  - Seulement en cas de création dynamique d’objets
  - P.ex. pas pour les singletons
    - Instance peut être configurée de l’extérieur
    - Sinon, pourquoi pas une « factory de factory » ?
  - Pas à grain trop fin
    - Création d’un objet tableau ?
      - P.ex. Column vs. Row Major
    - **NON !!!** => accès à un élément trop coûteux !

« Observer »

"Defines a link between objects so that when one object's state changes, all dependent objects are updated automatically. This pattern allows communication between objects in a loosely coupled manner."

GoF

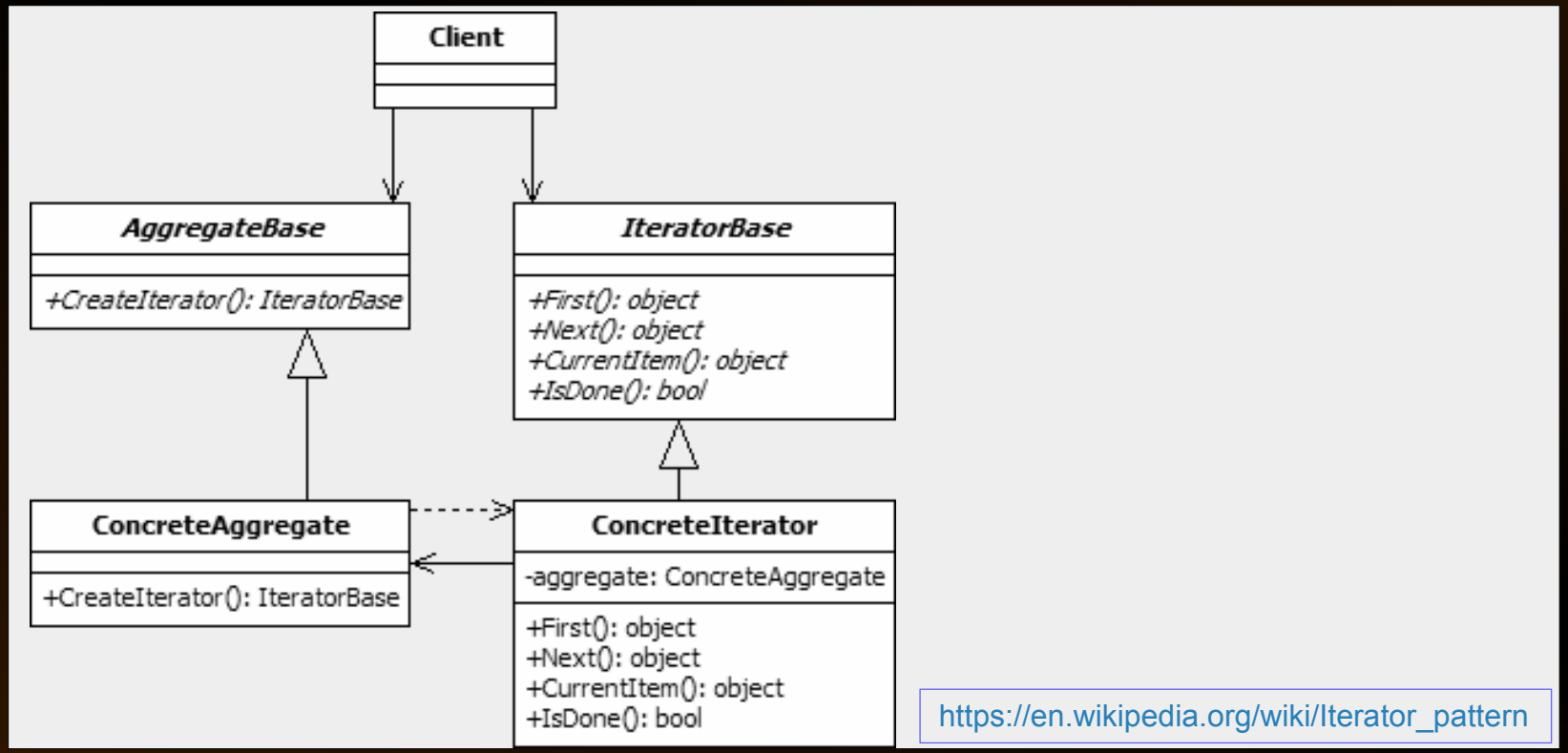


« Observer » Utilisation

- Exemple simulation numérique
  - Notification quand la donnée principale change
  - Plusieurs actions possibles
    - Écriture sur disque
    - Post-traitement local
    - Envoie via MPI
    - …
- Définition d’un changement
  - À chaque changement d’un point => **NON !**
    - Donnée incohérente + trop coûteux !
  - Quand le **tableau complet** est mis à jour

« Iterator »

"Provides a means for the elements of an aggregate object to be accessed sequentially without knowledge of its structure. This allows traversing of lists, trees and other structures in a standard manner."



GoF

https://en.wikipedia.org/wiki/Iterator_pattern

« Iterator » Utilisation

- Omniprésent en objet (Java, C#, …)
- Algorithme indépendant du stockage
  - Row major, Column major, Block based, …

- Mais…
  - **Accès fonction virtuelle pour chaque point !!!**

- Pourtant…
  - Iterator très utilisé en C++ HPC (p.ex. Kokkos)

Polymorphisme

Capacité dans certains langages de programmation d'obtenir un comportement différent pour le même code en fonction des types manipulés

- 2 types de polymorphismes
  - Polymorphisme dynamique : choix à l'exécution
    - p.ex. Fonctions virtuelles d'un objet
  - Polymorphisme statique : choix à la compilation
    - p.ex. Surcharge de fonctions, *templates* C++
- *Iterator* en C++ : basé sur les templates (Ø héritage)
  - Pas exactement le *design pattern* du ***GoF***
  - Choix à la compilation => pas de surcoût à l'exécution

Conclusion

- Design patterns
  - De bonnes pratiques de génie logiciel
  - Basés sur l'héritage
    - attention au surcoût des méthodes virtuelles
  - Savoir les utiliser, ne pas en abuser

- Objet & héritage
  - Pas la seule solution de génie logiciel
  - Parfois des inspirations design patterns mais implémentation différente