Patrons de conception : *Iterator*, *Builder*, fabriques

Pratique de la programmation orientée-objet

Michel Schinz – 2014-04-14

Patrons de conception

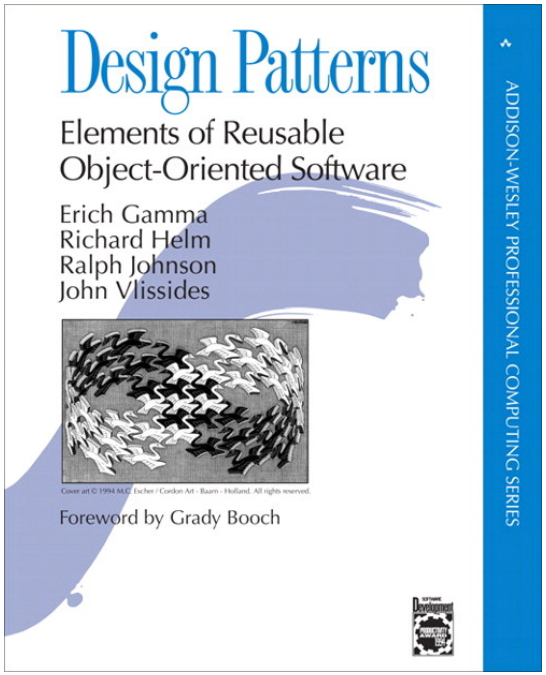
Problèmes récurrents

En programmation, comme dans toute discipline, certains problèmes sont récurrents. Un programmeur expérimenté sait identifier de tels problèmes, et connaît généralement leur solution.

Pourquoi ne pas répertorier ces problèmes et leur solution, afin de faciliter le travail des programmeurs ?

Patron de conception

Un **patron de conception** (*design pattern*) est une solution à un problème de conception récurrent.
Un tel patron est nommé et décrit en détail dans un répertoire de patrons – p.ex. le livre *Design Patterns, Elements of Reusable Object-Oriented Software* de Gamma, Helm, Johnson et Vlissides.



Exemple : itérateur

Problème récurrent :

Un objet possède une collection de valeurs et désire y donner accès, sans révéler la manière dont cette collection est représentée en interne.

Solution (patron de conception *Iterator*) :

Fournir un itérateur, à savoir un objet qui permet d'examiner les valeurs les unes après les autres, dans un ordre donné.

Attributs d'un patron

Les principaux composants d’un patron de conception sont :

- son **nom**,
- une description du **problème** résolu,
- une description de la **solution** à ce problème,
- une présentation des **conséquences** liées à l’utilisation du patron.

Intérêt des patrons

Les patrons de conception permettent de diffuser largement les meilleures solutions connues à différents problèmes récurrents.
De plus, ces solutions sont nommées, ce qui permet de raisonner et de communiquer à un plus haut niveau d'abstraction que lorsqu'on se concentre sur les détails de mise en œuvre.

Inconvénients des patrons

Malgré la récente excitation liée à leur « découverte », les patrons de conception ne sont pas une panacée. Une utilisation systématique des patrons ne saurait garantir qu'un programme soit bien conçu.
Il est donc important de n'utiliser un patron que lorsque cela est justifié, et que les avantages liés à son utilisation compensent les inconvénients – p.ex. l'augmentation de la complexité du programme qui en résulte fréquemment.

Patrons et langages

Un patron de conception n'est normalement pas lié à un langage de programmation donné.
Toutefois, beaucoup de patrons ont été inventés dans le contexte de langages orienté-objets. Ils font une utilisation intensive des concepts de ce type de langages – classes, objets, polymorphisme d'inclusion, etc. – et sont donc difficilement utilisables dans d'autres contextes.
Il faut aussi noter que certains langages possèdent des concepts qui rendent certains patrons obsolètes ! Par exemple, le langage Scala possède le filtrage de motifs (*pattern matching*) qui rend l'utilisation du patron *Visitor* inutile.

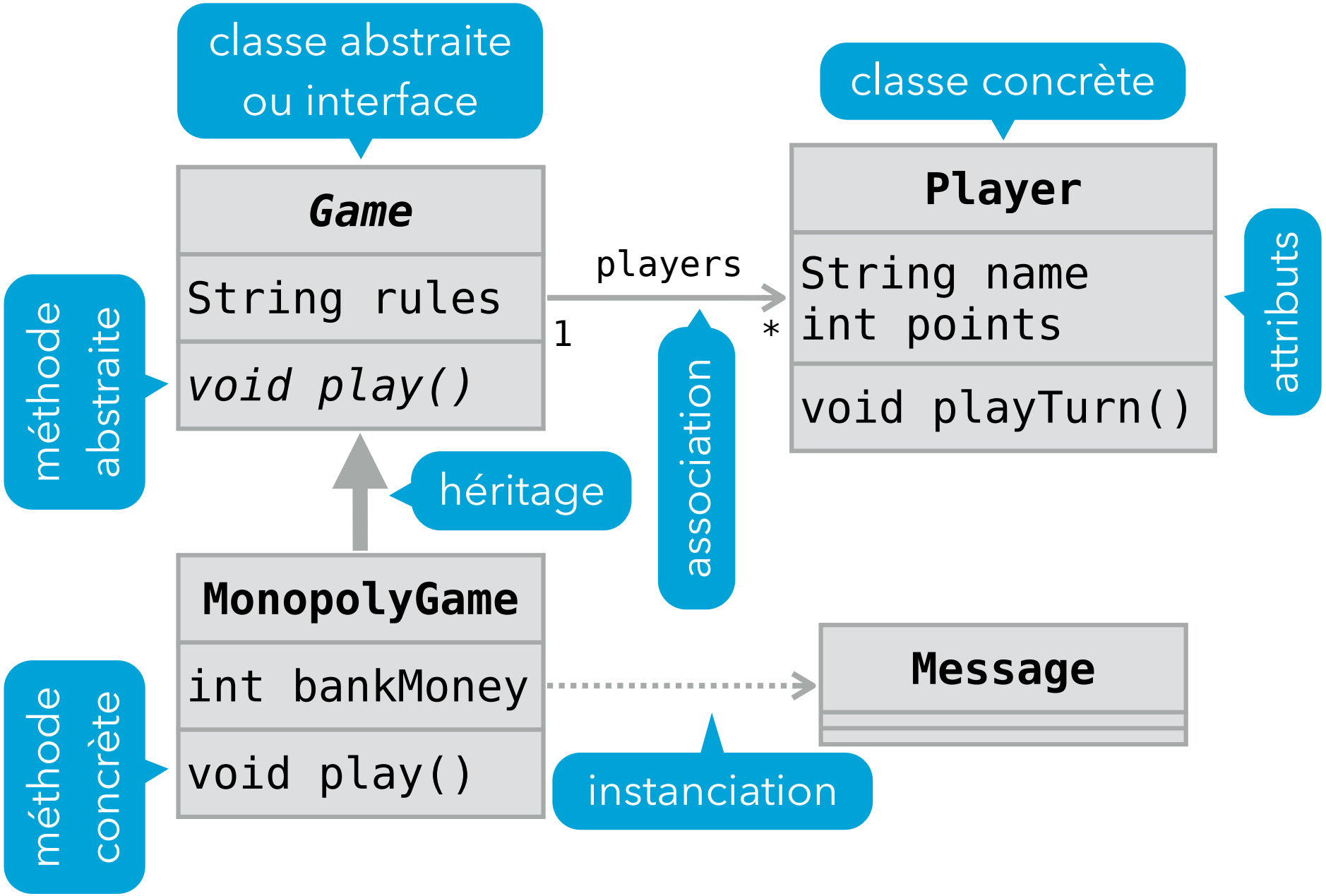
Diagramme de classe (digression)

Diagramme de classe

Un diagramme de classe décrit visuellement un ensemble de classes ou interfaces et leurs relations, qui peuvent être de différente nature :

- **héritage** : lorsqu'une classe hérite d'une autre ou implémente une interface,
- **association** : lorsqu'une classe utilise une ou plusieurs instances d'une autre classe,
- **instanciation** : lorsqu'une classe créée des instances d'une autre classe.

Diagramme de classe



Patron n°1 :
Iterator (ou *Cursor*)

Illustration du problème

Les classes qui implémentent l'interface `List<E>`, qui représente une liste d'éléments, doivent fournir un moyen de parcourir ces éléments les uns après les autres.
Une possibilité serait d'exposer la représentation interne au client – p.ex. en rendant la classe des nœuds visible dans le cas des listes chaînées. Mais l'encapsulation serait alors violée, et le parcours dépendrait du type de liste parcouru.
Comment faire ?

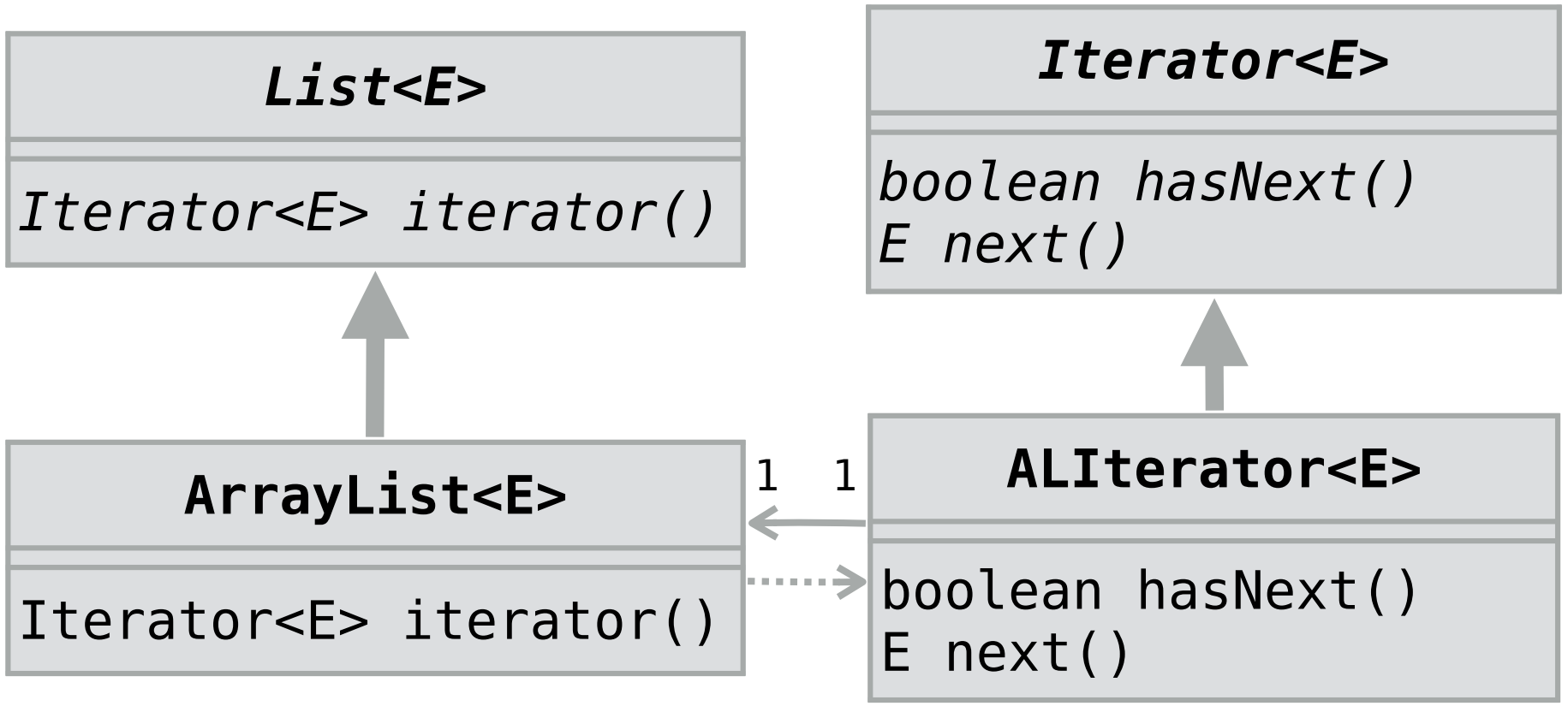
Solution

Pour permettre le parcours des éléments d'une liste sans exposer sa représentation interne, on peut utiliser un objet qui désigne à tout moment un élément de la liste. Cet objet possède des opérations permettant d'obtenir l'élément actuellement désigné et de passer à l'élément suivant, voire au précédent.
Un tel objet s'appelle un **itérateur**, ou un **curseur**.

Exemple : ArrayList

```
public interface Iterator<E> {
  // … méthodes hasNext, next, etc.
}
public final class ArrayList<E>
    implements List<E> {
  // … autres méthodes de List
  public Iterator<E> iterator() {
    return new ALIterator(this);
  }
  private final static class ALIterator<E>
      implements Iterator<E> {
    // … méthodes hasNext, next, etc.
  }
}
```

Diagramme de classes



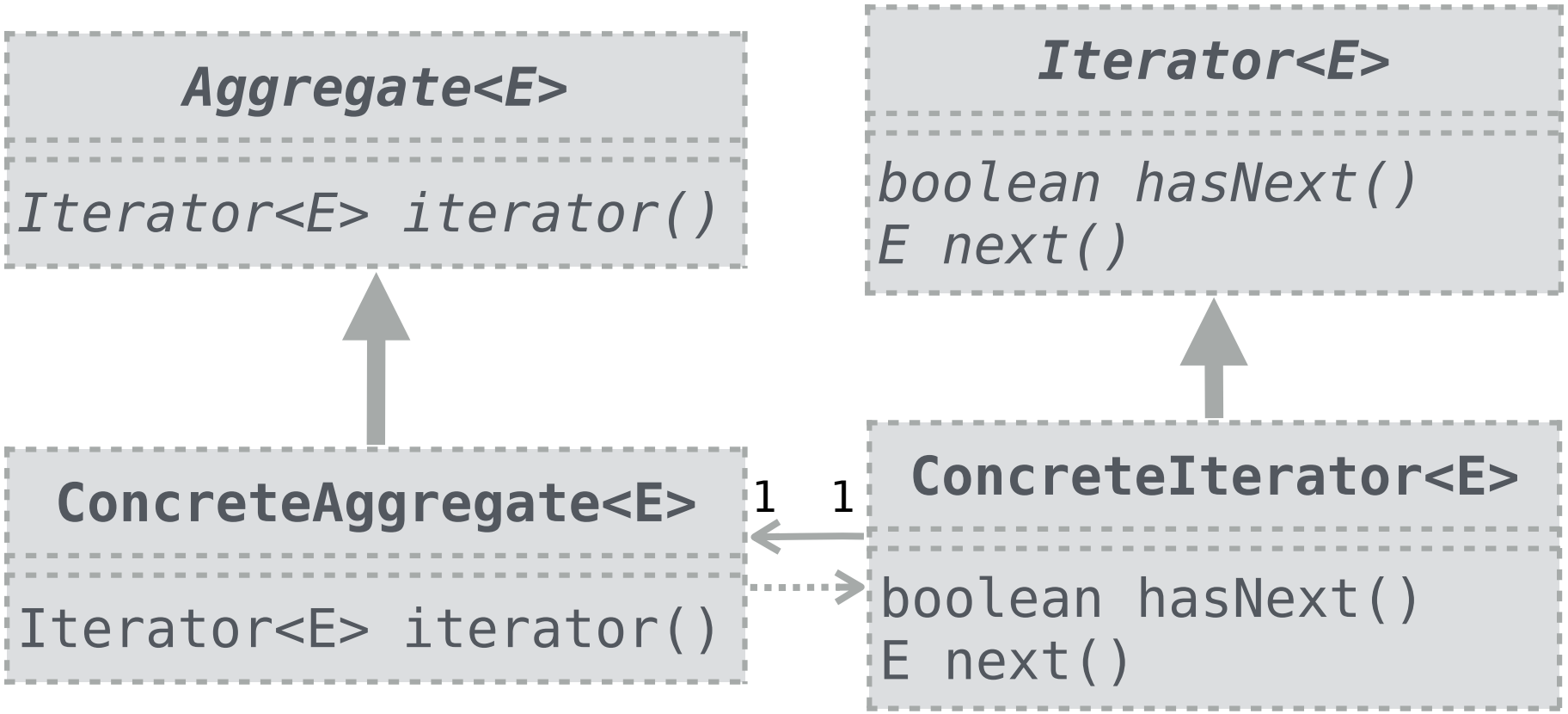
Généralisation

Le concept d'itérateur peut être utilisé chaque fois qu'un objet possède une collection d'éléments qu'un client doit pouvoir parcourir sans connaître la représentation interne de la collection.

Exemples : parcours du contenu d'un répertoire dans un système de fichiers, parcours des résultats d'une requête à une base de données, etc.

Diagramme de classes

Le diagramme de classes ci-dessous illustre les classes (imaginaires) impliquées dans l'itération sur les éléments d'une classe `ConcreteAggregate` au moyen de l'itérateur `ConcreteIterator`.



Raffinements

Les itérateurs présentés ici sont très simples. Selon les besoins, on peut imaginer les augmenter avec des méthodes permettant de :

- passer à l'élément précédent,
- aller au début ou à la fin de la collection,
- supprimer ou ajouter des éléments à la position désignée par l'itérateur,
- etc.

Types de parcours

Les itérateurs définis sur les listes jusqu'à présent effectuent un parcours du premier au dernier élément.
On peut imaginer d'autres sortes de parcours, p.ex. en sens inverse. Pour des structures de données plus complexes – p.ex. les arbres – les possibilités sont plus nombreuses : en largeur ou profondeur d'abord, en pré-ordre, post-ordre, etc.
Chacun de ces types de parcours peut être mis en œuvre par un itérateur différent.

Intérêt du patron *Iterator*

Un itérateur permet de parcourir les éléments d'une collection en faisant abstraction de sa représentation.
Ainsi, on peut écrire exactement le même code pour parcourir les éléments d'un ensemble représenté par un arbre de recherche que ceux d'une liste représentée par chaînage.
Il est même possible de définir des itérateurs sur des structures infinies, p.ex. la liste des nombres premiers !

Exemples réels

La bibliothèque Java utilise des itérateurs – représentés par l'interface `java.util.Iterator` – pour permettre le parcours des collections (listes, ensembles, etc.) et de beaucoup d'autres objets : composantes d'un chemin de répertoire, documents XML, etc.

Le concept d'itérateur est très répandu et se retrouve dans beaucoup de langages orienté-objets : C++, Python, etc.

Patron n°2:
Builder

Illustration du problème

Admettons que l'on désire écrire une bibliothèque de calcul matriciel. Un composant central d'une telle bibliothèque est la (ou les) classe(s) modélisant les matrices.
Comme toutes les classes représentant des entités mathématiques, ces classes devraient être immuables. Cela implique que la totalité des éléments d'une matrice doivent être spécifiés lors de sa construction. Pour les grosses matrices, ou les matrices creuses, cela peut s'avérer pénible.
Comment résoudre ce problème ?

Solution

Une solution consiste à offrir un bâtisseur de matrice, c'est-à-dire une classe représentant une matrice en cours de construction.
Le bâtisseur est modifiable, et possède des méthodes pour changer les différents éléments de la matrice en cours de construction. Il possède également une méthode pour construire la matrice.

Matrices

```
public interface Matrix {
  double get(int r, int c);
  Matrix transpose();
  Matrix inverse();
  Matrix add(Matrix that);
  Matrix multiply(double that);
  Matrix multiply(Matrix that);
  // … autres méthodes
}
public final class DenseMatrix
  implements Matrix { … }
public final class SparseMatrix
  implements Matrix { … }
```

Bâtisseur de matrice

```
public final class MatrixBuilder {
  private double[][] elements;
  public MatrixBuilder(int r, int c) {
    elements = new double[r][c];
  }
  public double get(int r, int c) {
    return elements[r][c];
  }
  public void set(int r, int c, double v) {
    elements[r][c] = v;
  }
  public Matrix build() {
    return new DenseMatrix(elements);
  }
}
```

Types de matrices

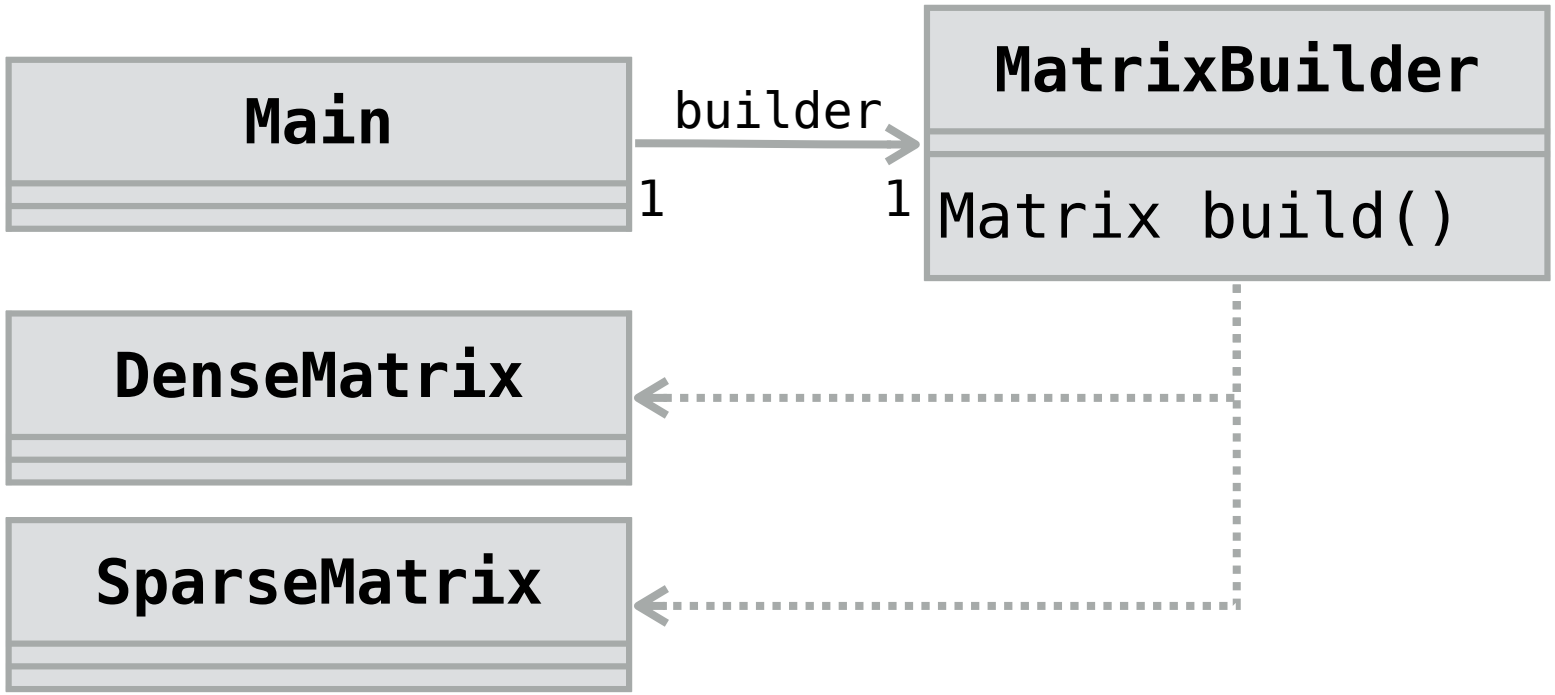
Outre le fait que le bâtisseur permet de construire une matrice en plusieurs étapes, il permet également de choisir une représentation pour la matrice qui soit appropriée à son contenu !
Par exemple, une matrice creuse – c-à-d dont la plupart des éléments valent 0 – peut être représentée au moyen d'une table associative contenant les éléments non-nuls. Une matrice dense peut être représentée par un tableau bi-dimensionnel.
Le choix entre les deux représentations peut être fait par le bâtisseur au moment de la création de la matrice.

Bâtisseur intelligent

```
public final class MatrixBuilder {
  // … comme avant
  public Matrix build() {
    if (density() > 0.5)
      return new DenseMatrix(elements);
    else
      return new SparseMatrix(elements);
  }
}
```

Diagramme de classes

Le diagramme de classes ci-dessous montre les classes impliquées dans la construction, par une classe `Main`, d'une matrice, au moyen d'un bâtisseur.

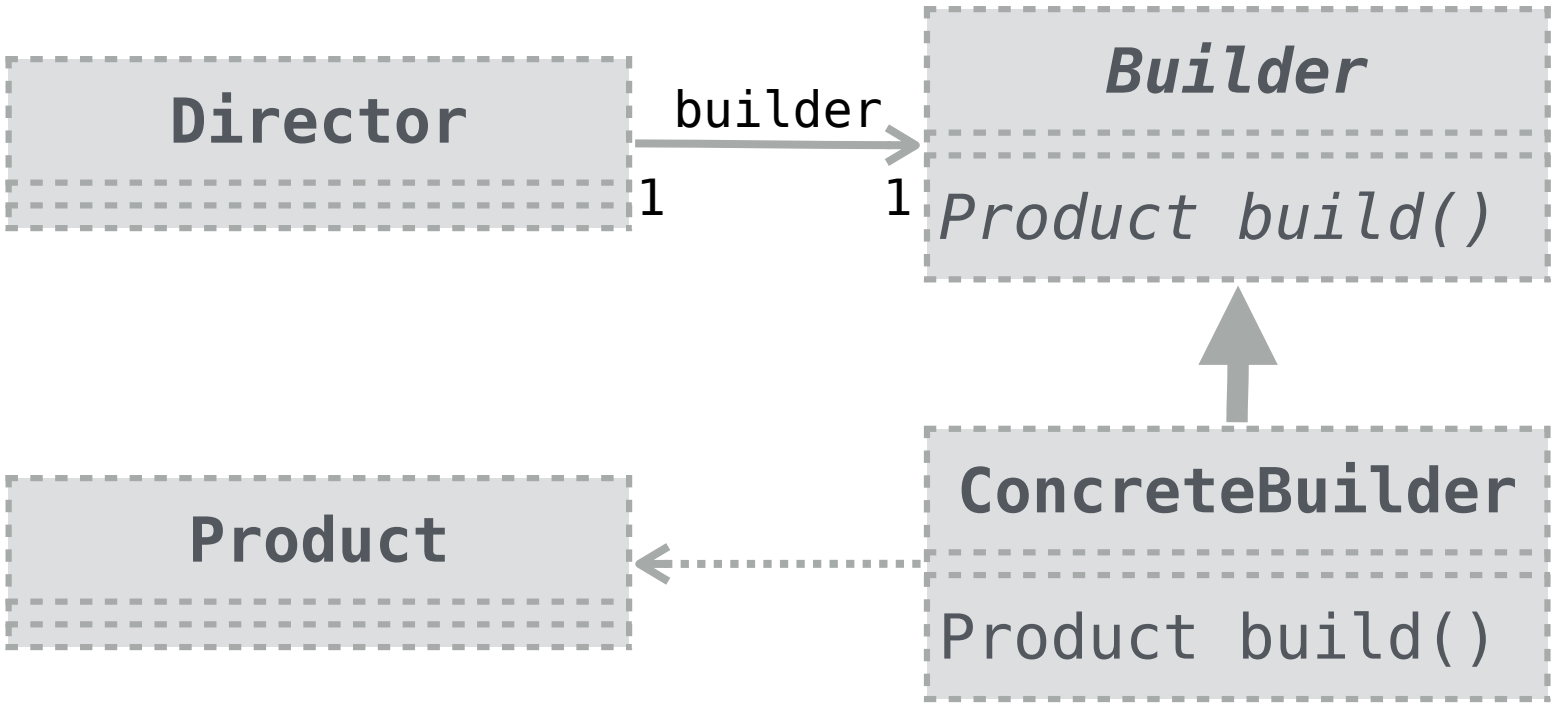


Généralisation

Chaque fois que le processus de construction d'une classe est assez difficile pour que l'on désire le découper en plusieurs étapes, on peut utiliser un objet pour stocker l'état de l'objet en cours de construction.
C'est l'idée du patron de conception *Builder*.

Diagramme de classes

Le diagramme de classes ci-dessous illustre les classes impliquées dans la construction d'une instance de la classe `Product` par une instance d'une classe `Director`, au moyen d'un bâtisseur de classe `ConcreteBuilder`.



Exemple réel

La classe `String` (de `java.lang`) modélise les chaînes de caractères, qui ne sont pas modifiables.
La classe `StringBuilder` sert de bâtisseur pour les chaînes de caractères. Elle possède entre autres une méthode `append` pour ajouter la représentation textuelle d'un objet à la chaîne en cours de construction, et la méthode `toString` pour obtenir la chaîne construite.
(Notez que la concaténation de chaînes de caractères au moyen de l'opérateur `+` est traduite via `StringBuilder`).

Patron n°3:
Factory Method

Illustration du problème

Plusieurs jeux récents sont extensibles par programmation : l'utilisateur a la possibilité de modifier ou améliorer certains de leurs aspects en écrivant du code.

Dans un langage orienté-objets, une manière naturelle de permettre ce genre d'extensions consiste à laisser l'utilisateur définir ses propres sous-classes des classes principales du jeu.

Illustration du problème

Pour rendre le problème plus concret, imaginons un jeu dans lequel chaque joueur possède un bateau, modélisé par une classe `Ship`.
Afin d'étendre le jeu, un utilisateur pourrait définir sa propre sous-classe de `Ship`, p.ex. `MagicShip`.
Une question se pose alors : comment s'assurer que tous les bateaux créés lors du jeu soient des instances de `MagicShip` et pas de `Ship` ?

Illustration du problème

```
public class Game {
  public Game(int pCount) {
    Player[] players = new Player[pCount];
    Ship[] ships = new Ship[pCount];
    for (int i = 0; i < pCount; ++i) {
      players[i] = new Player();
      ships[i] = new Ship();
      ships[i].setPosition(…);
      ships[i].setOwner(players[i]);
    }
    // … initialisation complète du jeu
  }
}
```

problème

Solution

Afin de résoudre ce problème, une solution est de définir une méthode dans la classe `Game` pour créer un nouveau bateau, puis de l'utiliser systématiquement à la place de l'opérateur `new`.
Cette méthode peut être redéfinie dans une sous-classe afin de créer un autre type de bateau.
Une telle méthode est appelée une **méthode fabrique** (*factory method*) ou **constructeur virtuel** (*virtual constructor*).

Jeu extensible

```
public class Game {
  Ship newShip() { return new Ship(); }
  public Game(int pCount) {
    Player[] players = new Player[pCount];
    Ship[] ships = new Ship[pCount];
    for (int i = 0; i < pCount; ++i) {
      players[i] = new Player();
      ships[i] = newShip();
      ships[i].setPosition(…);
      ships[i].setOwner(players[i]);
    }
    // … initialisation complète du jeu
  }
}
```

méthode fabrique

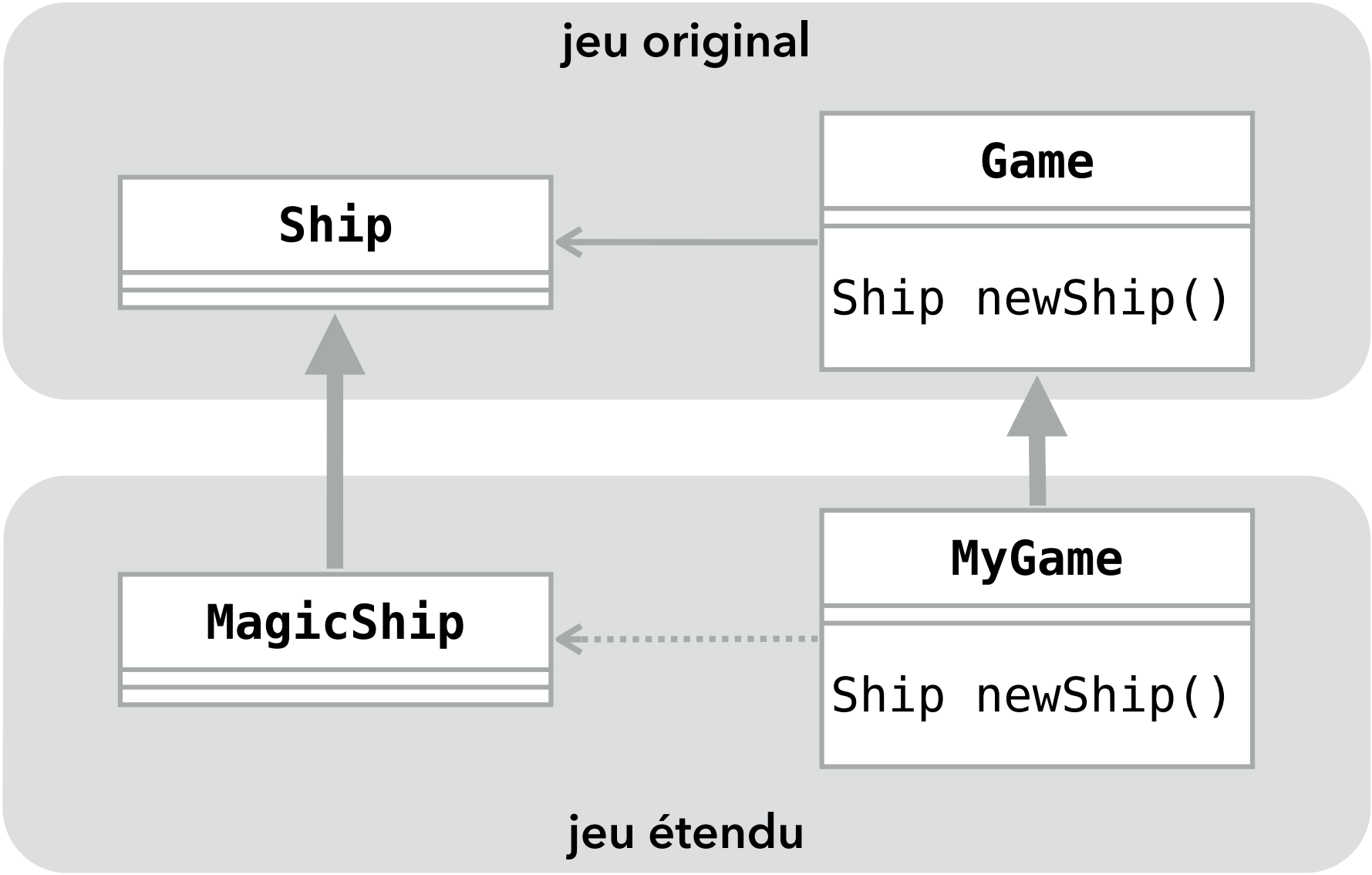
tous les bateaux sont créés via la fabrique

Jeu étendu

```
public class MagicShip extends Ship {
  // …
}
public class MyGame extends Game {
  @Override
  Ship newShip() {
    return new MagicShip();
  }
  public MyGame(int playersCount) {
    super(playersCount);
  }
}
```

nouvelle version de la méthode fabrique

Diagramme de classes

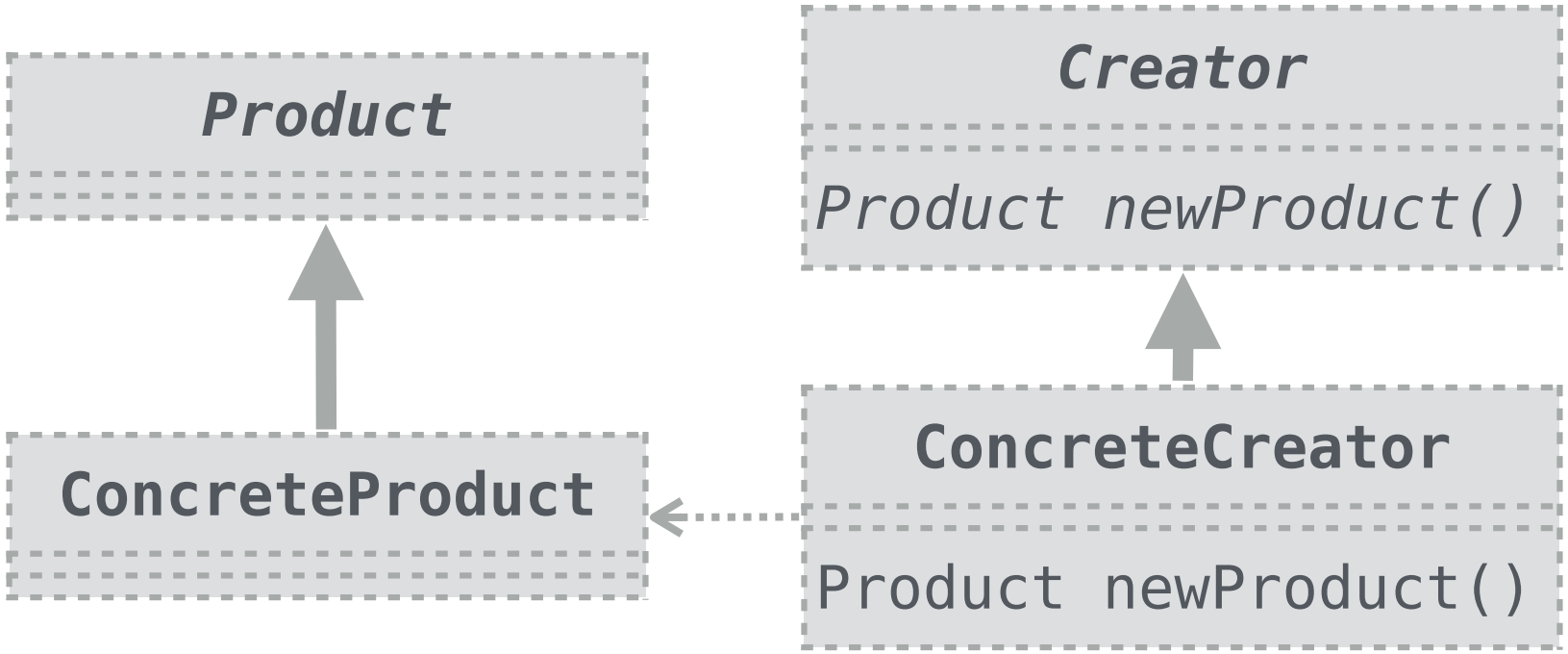


Généralisation

Chaque fois qu'une classe doit créer une instance d'une autre classe qui n'est pas connue à l'avance, elle peut utiliser une méthode fabrique.
En (re)définissant la méthode fabrique dans une sous-classe, il est possible de spécifier la classe exacte à créer.
Le patron de conception *Factory Method* décrit cette solution.

Diagramme de classes

Le diagramme de classes ci-dessous illustre la création, par un object `Creator`, d'une instance d'une classe (inconnue par lui) implémentant l'interface `Product`.



Exemple réel

Les classes de test des collections (`ListTest`, `SetTest`, etc.) sont écrites en fonction de l'interface qu'elles testent (`List` pour `ListTest`, `Set` pour `SetTest`, etc.). Elles n'ont pas connaissance de la classe précise des objets qu'elles testent.

Etant donné que ces classes de test doivent néanmoins pouvoir créer des instances des classes à tester, elles possèdent une méthode fabrique abstraite (`newList` pour `ListTest`, `newSet` pour `SetTest`, etc.). Cette méthode est définie dans les sous-classes de test concrètes pour créer une instance de la classe à tester. Ainsi, `ArrayListTest` définit `newList` pour lui faire créer une instance de `ArrayList`.

Patron n°3:
Abstract Factory

Illustration du problème

Continuons avec notre exemple de jeu extensible, mais en imaginant une situation plus complexe. Plutôt que de n'avoir que la seule classe `Ship` qui soit extensible par l'utilisateur, imaginons que nous en ayons trois: `Ship`, `Treasure` et `Island`.

Posons-nous la même question qu'avant : comment s'assurer que chaque fois qu'un bateau, trésor ou île est créé, la bonne classe est utilisée ?

Solution

Une solution consiste bien entendu à définir trois méthodes fabriques dans la classe `Game`.
Toutefois, lorsque le nombre de méthodes fabriques commence à devenir important et que les classes dont elles créent des instances sont liées, il peut devenir préférable de les extraire de la classe `Game` pour les mettre dans une interface séparée.
Une telle interface, qui ne contient que des méthodes fabriques, est appelée **fabrique abstraite** (*abstract factory*)

Fabrique abstraite

Pour notre exemple, la fabrique abstraite se présente ainsi :

```
interface GameFactory {
  Ship newShip();
  Island newIsland();
  Treasure newTreasure();
}
```

On peut utiliser un objet implémentant cette interface pour créer tous les bateaux, îles et trésors nécessaires à un jeu.

Jeu extensible

```
class Game {
  private final Ship[] ships;
  private final Island[] islands;
  private final Treasure[] treasures;
  public Game(GameFactory factory) {
    ships = new Ship[] {
      factory.newShip(), … };
    islands = new Island[] {
      factory.newIsland(), … };
    treasures = new Treasure[] {
      factory.newTreasure(), … };
  }
}
```

toutes les instances d'objets extensibles sont créées via la fabrique

Fabrique concrète

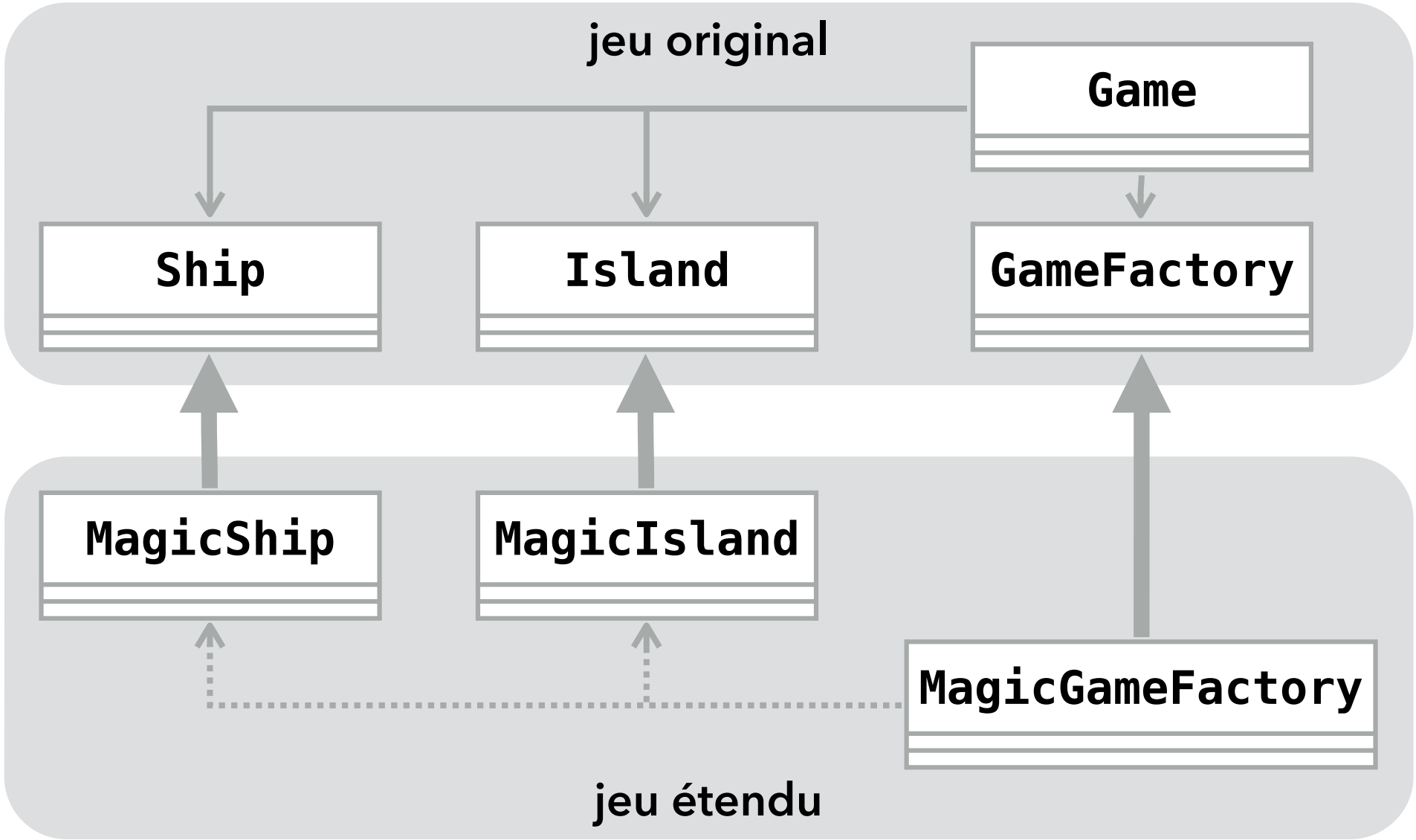
```
public final class MagicShip
    extends Ship { … }
public final class MagicIsland
    extends Island { … }
public final class MagicTreasure
    extends Treasure { … }
public final class MagicGameFactory
    implements GameFactory {
  public Ship newShip()
    { return new MagicShip(); }
  public Island newIsland()
    { return new MagicIsland(); }
  public Treasure newTreasure()
    { return new MagicTreasure(); }
}
```

Jeu étendu

Pour obtenir une version magique du jeu, il suffit de créer une instance de la classe `Game` en lui passant la fabrique correspondante :

```
// Classe principale de la version magique
// du jeu.
public final class MagicGameMain {
  public static void main(String[] args) {
    // crée et démarre le jeu
    new Game(new MagicGameFactory());
  }
}
```

Diagramme de classes

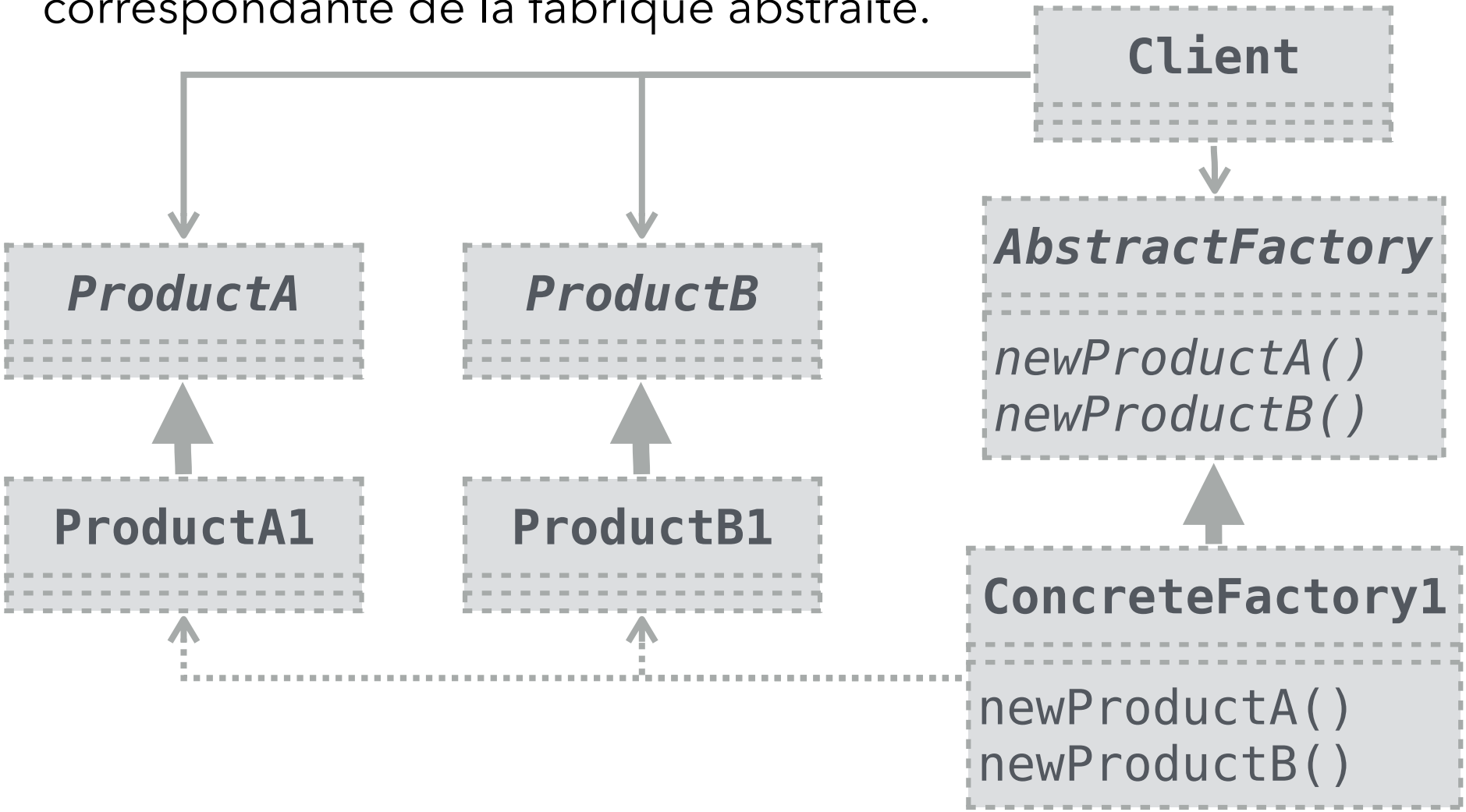


Généralisation

Chaque fois qu'une classe doit créer des instances d'une famille de classes apparentées mais non connues d'avance, elle peut utiliser une fabrique abstraite.
Le patron de conception *Abstract Factory* décrit cette solution.

Diagramme de classes

Lorsque le client `Client` désire créer une instance d'un produit `Product`?, il utilise la méthode `newProduct`? correspondante de la fabrique abstraite.



Types de fabriques

Les patrons *Factory Method* et *Abstract Factory* sont très proches. En fait, une fabrique abstraite n'est rien d'autre qu'un ensemble de méthodes fabriques regroupées dans un objet.
On utilise en général une fabrique abstraite dès le moment où il doit être possible de créer des instances d'une famille de classes liées.

Résumé

Un patron de conception est une solution, nommée et documentée, à un problème de conception récurrent. Un bon programmeur se doit de connaître un certain nombre de patrons importants et de savoir quand les utiliser.

Le patron *Iterator* permet à un objet conteneur de donner accès à ses éléments sans révéler la manière dont ils sont représentés en interne.

Le patron *Builder* permet de construire petit-à-petit un objet complexe.

Les patrons *Factory Method* et *Abstract Factory* permettent de construire des instances de classes dont le type exact est inconnu.