MI3/GL
Cours de Génie Logiciel
Programmation Orientée Objet (Avancée)

Daniel Le Berre

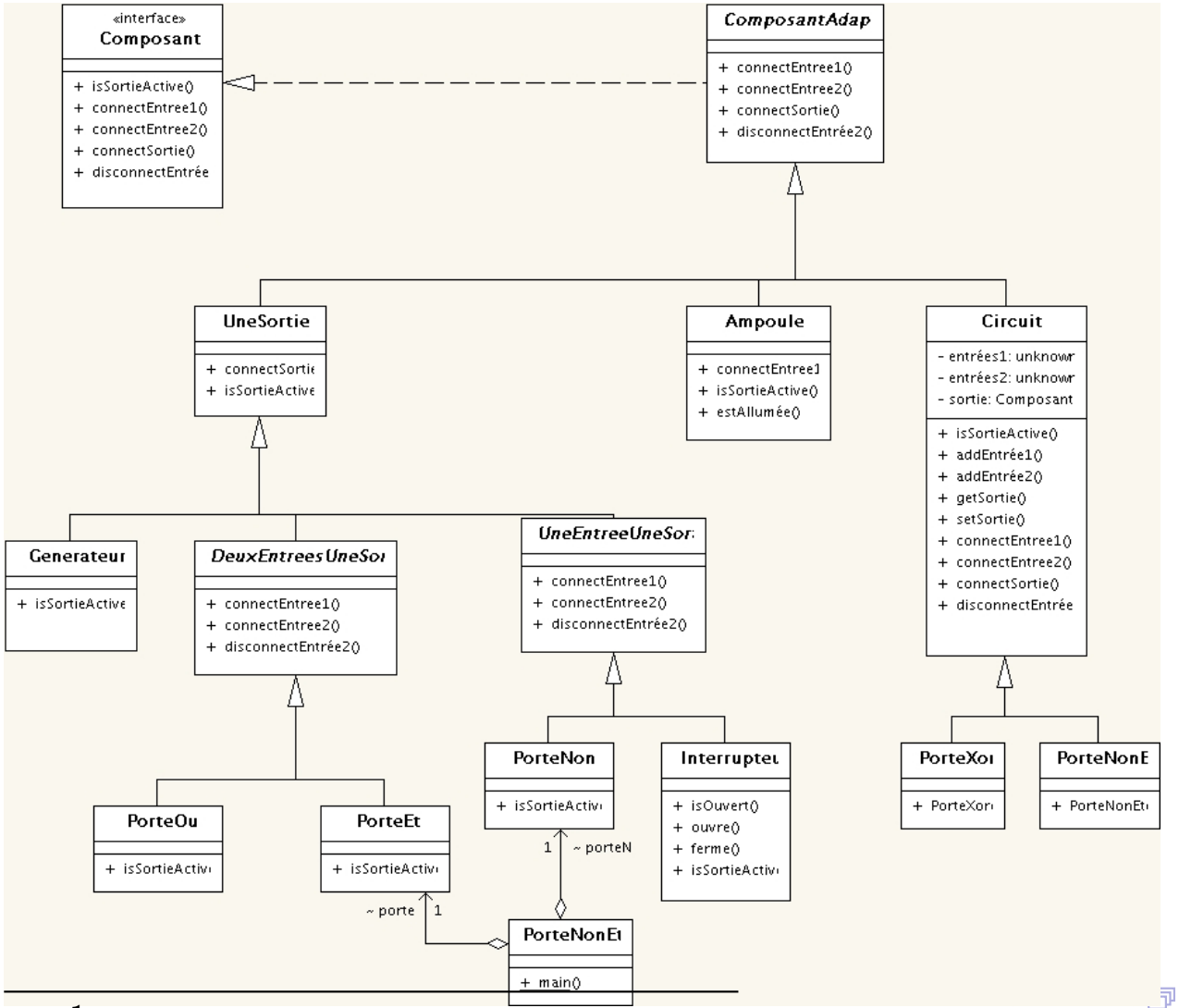
CRIL-CNRS FRE 2499, Université d'Artois, Lens, FRANCE
{leberre}@cril.univ-artois.fr

26 novembre 2004

Introduction aux design patterns

- Principes
- Patrons de création
- Patrons structuraux
- Patrons comportementaux

Avant de commencer ...



[1] DOD — Department of Defense

Patrons de conception (Design Pattern)

- Mouvement né en architecture
- Adapté au développement de logiciels par [7].

Définition (Christophe Alexander)

Chaque patron décrit un problème qui apparaît encore et encore dans notre environnement, et décrit la base de la solution à ce problème, de manière telle que vous pouvez utiliser cette solution des millions de fois sans jamais le faire deux fois de la même manière.

Autres définitions

- Des solutions récurrentes à des problèmes de conception ... Un ensemble de règles décrivant comment accomplir certaines tâches dans le cadre du développement de logiciels. (Pree, 1994)
- Les patrons s'intéressent plus à la réutilisation de thèmes de conception architecturale récurrents tandis que les Frameworks s'intéressent plus à la conception détaillée ... et à l'implantation. (Copien et Schmidt, 1995)
- Un patron adresse un problème récurrent qui apparait dans des situations de conception spécifiques et représente une solution à ce problème. (Bushmann et al 1996)
- Les patrons identifient et spécifient des abstractions qui sont au dessus des simples classes ou instances, ou composants. (Gamma et al, 1993).

La bande des quatre

Quatre auteurs :

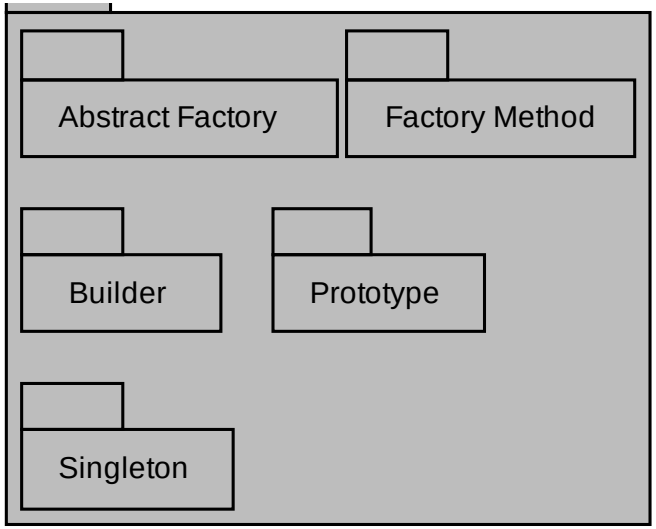
- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

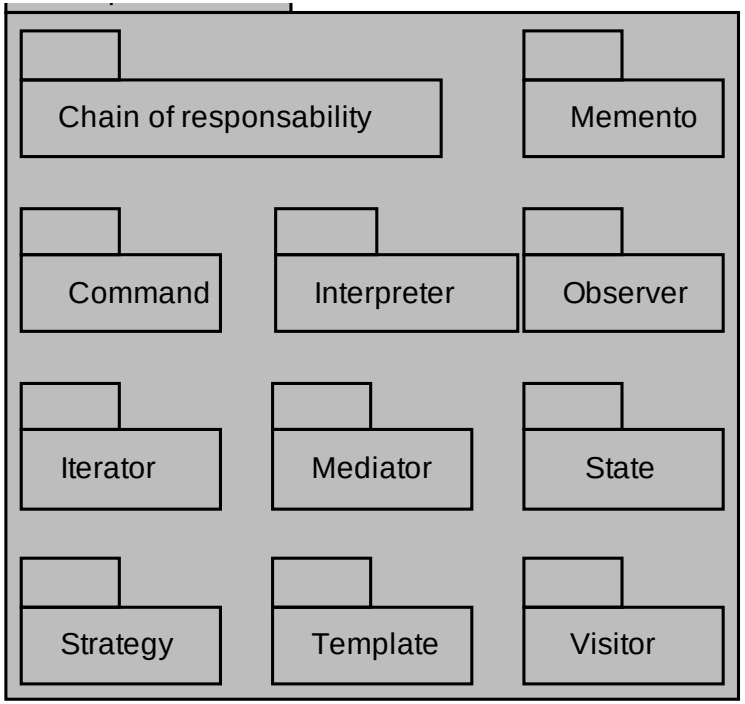
Pour un livre indispensable : Design Patterns, Elements of Reusable Object-Oriented Software

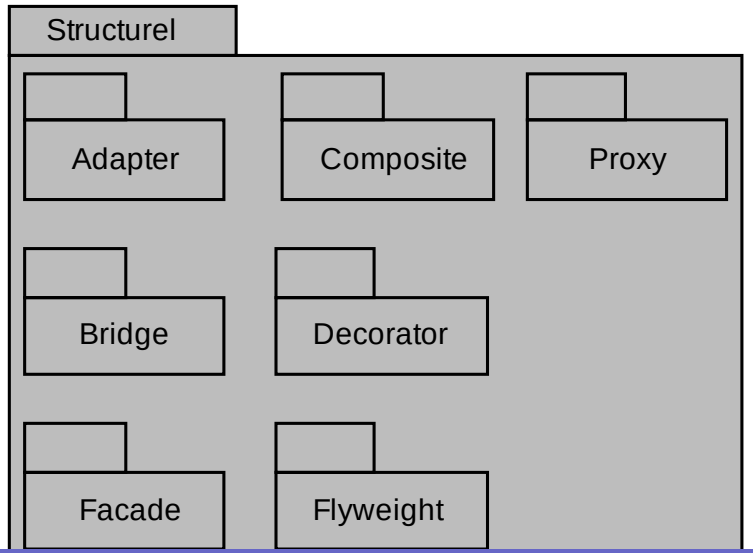
Principes

- Classe vs Objet
- Héritage vs Délégation
- Abstrait vs Concret
- Interface vs Réalisation

Types de patrons







L'espace des design patterns

	création	structure	comportement
classe	Fabrication	Adaptateur	Interprete Patron de méthode
objet	Fabrique abstraite Constructeur Prototype Singleton	Adaptateur Pont Composé Décorateur Façade Flyweight Proxy	Chaine de responsabilité Commande Itérateur Médiateur Mememto Observer Etat Stratégie Visiteur

Les patrons de création

- Cachent le type concret des objets
- Cachent comment les objets sont construits
- Permet beaucoup de flexibilité concernant qui construit quoi, comment et quand.
- Deux types de patrons de création :
  - Les patrons de création de classe utilisent l'héritage pour faire varier les classes instanciées.
  - Les patrons de création objets délèguent l'instanciation à un autre objet.

Liste des patrons de création

Fabrique abstraite Fournit une interface pour construire des familles d'objets semblables ou dépendants sans spécifier leurs classes contrètes.

Builder Sépare la construction d'un objet complexe de sa représentation afin d'utiliser le même processus de construction pour différentes représentations.

Méthode fabrique Définit une méthode pour créer une instance mais laisse les sous classes gérer l'instanciation.

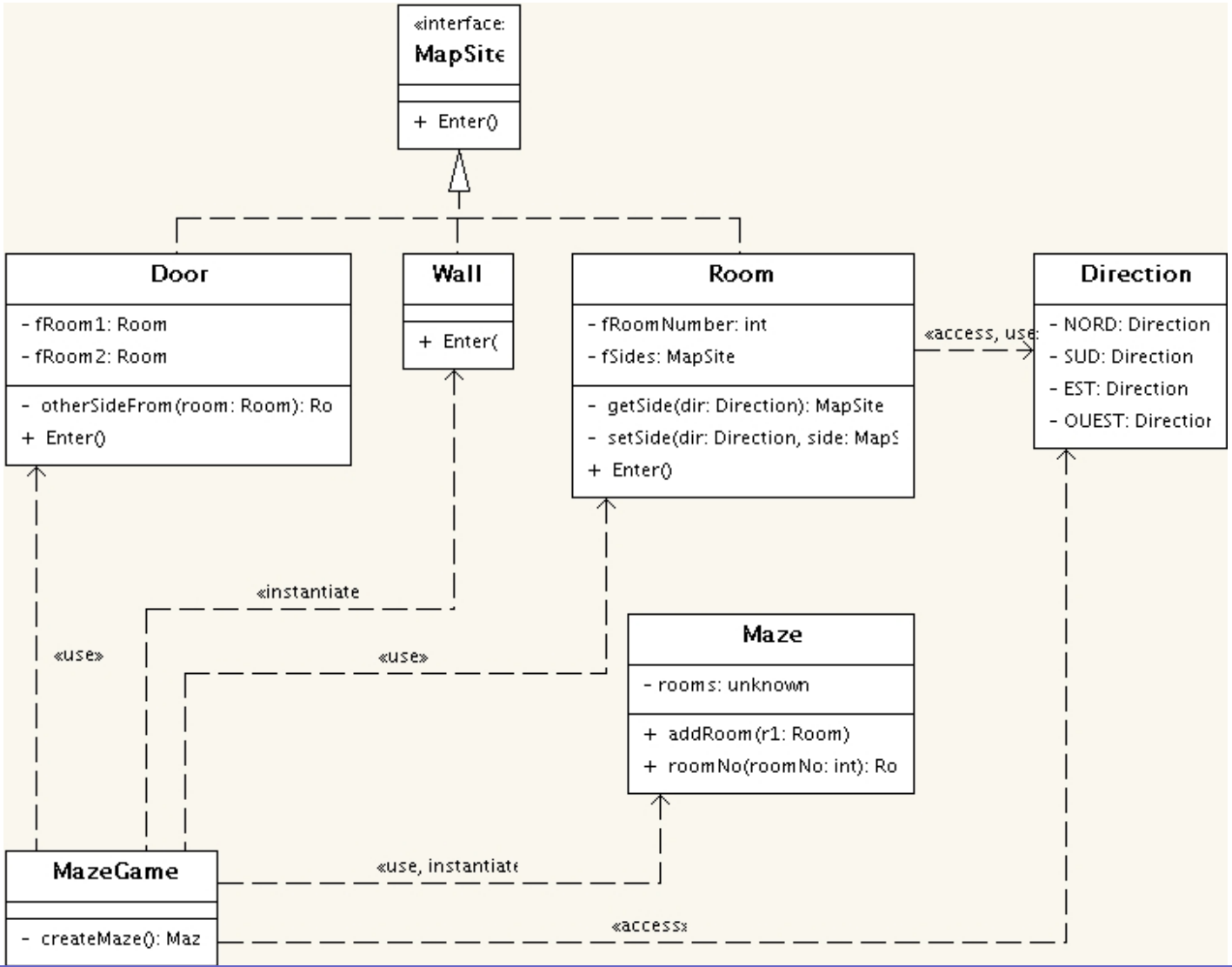
Prototype Création d'une instance à partir d'un modèle, par copie.

Singleton Assure qu'une classe à une seule instance et fournit un accès à cette méthode.

Exemple utilisé dans le GoF : le labyrinthe

- Un labyrinthe Maze est une collection de pièces Room.
- Chaque pièce à quatre cotés, désignés par leur Direction.
- Chaque coté peut contenir soit un mur Wall soit une porte Door.
- Une porte peut relier deux pièces.

Diagramme de classe du labyrinthe



Exemple de création de labyrinthe

```
Maze createMaze() {
  Maze aMaze = new Maze();
  Room r1 = new Room(1);
  Room r2 = new Room(2);
  Door theDoor = new Door(r1,r2);
  aMaze.addRoom(r1);
  aMaze.addRoom(r2);
  r1.setSide(NORD,new Wall());
  r1.setSide(EST,theDoor);
  r1.setSide(SUD,new Wall());
  r1.setSide(OUEST,new Wall());
  r2.setSide(NORD,new Wall());
  r2.setSide(EST,new Wall());
  r2.setSide(SUD,new Wall());
  r2.setSide(OUEST,theDoor);
  return aMaze;
}
```

Gestion de différents “Look and Feel”

- Un système graphique utilise plusieurs type de widgets : ascenseurs, boutons, cases à cocher, etc.
- Chaque L&F à sa propre façon de dessiner ses widgets.
- Un système supportant différents L&F ne doit pas coder directement l'instanciation des classes spécifiques à un L&F, sinon il sera diffcile d'en changer à l'exécution.
- Créer une classe abstraite qui réunit la création de tous les widgets et spécialiser cette classe pour chaque L&F.

La fabrique abstraite

Objectif Utiliser des familles ou des jeux d'objets pour des clients (ou des cas) donnés.

Problème Les familles d'objets doivent être instanciées.

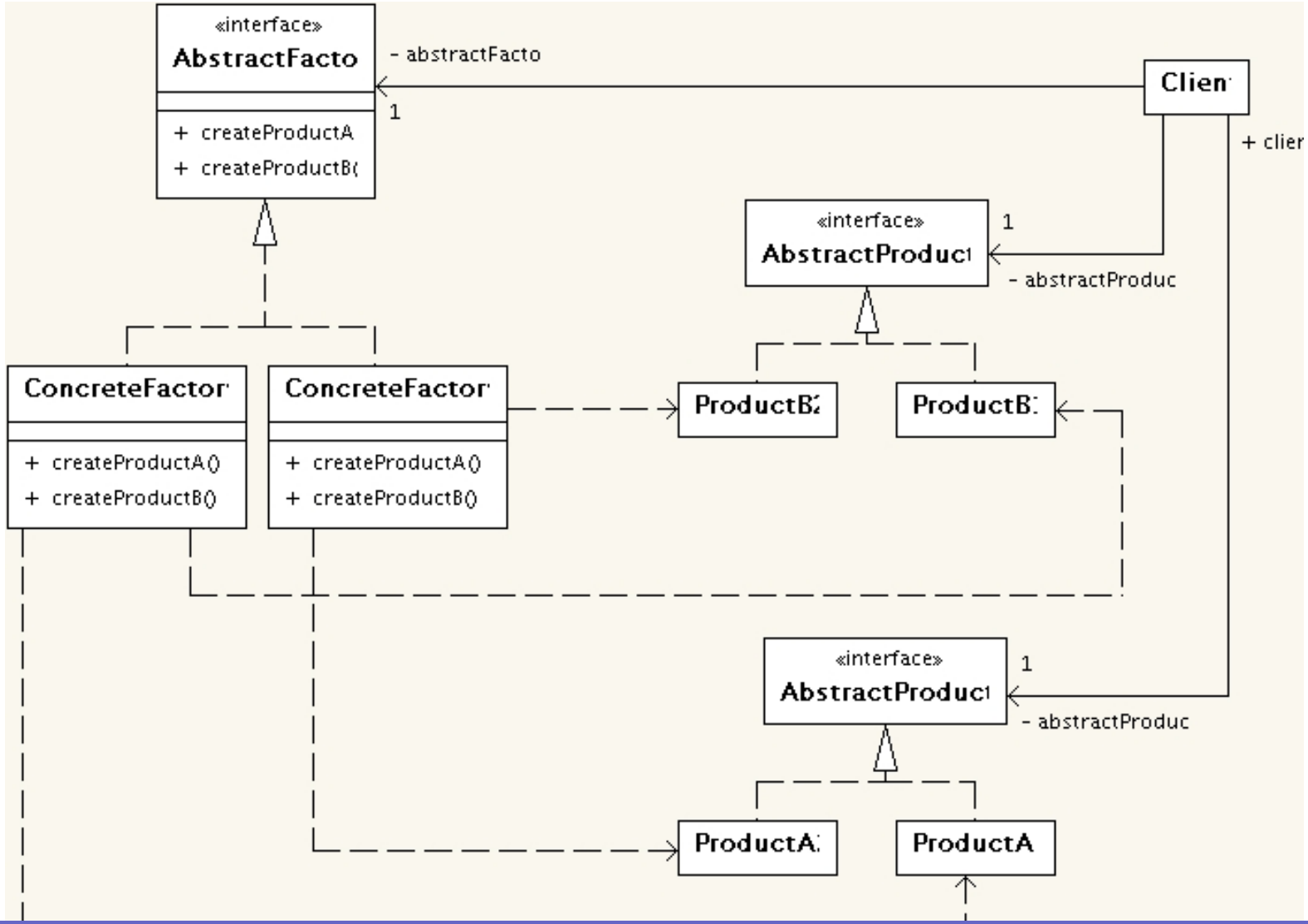
Solution Coordonner la création de familles d'objets. Méthode consistant à extraire les règles d'instanciation de l'objet client qui utilise les objets crées.

Participants FabriqueAbstraite détermine l'interface déterminant le mode de création de chaque membre de la famille d'objets utilisée. Généralement, chaque famille est créée à l'aide de sa propore classe FabriqueConcrete.

Conséquence Le patron sépare les règles de sélection des objets de leur logique d'utilisation.

Référence GoF Pages 87 à 96.

Fabrique Abstraite : Diagramme de classe



Fabrique abstraite pour le labyrinthe 1/2

```
public interface MazeFactory {

   Maze makeMaze();

   Wall makeWall();

   Room makeRoom(int no);

   Door makeDoor(Room r1,Room r2);
}
```

Fabrique abstraite pour le labyrinthe 2/2

```
Maze createMaze(MazeFactory factory) {
  Maze aMaze = factory.makeMaze();
  Room r1 = factory.makeRoom(1);
  Room r2 = factory.makeRoom(2);
  Door theDoor = factory.makeDoor(r1,r2);
  aMaze.addRoom(r1);
  aMaze.addRoom(r2);
  r1.setSide(NORD,factory.makeWall());
  r1.setSide(EST,theDoor);
  r1.setSide(SUD,factory.makeWall());
  r1.setSide(OUEST,factory.makeWall());
  r2.setSide(NORD,factory.makeWall());
  r2.setSide(EST,factory.makeWall());
  r2.setSide(SUD,factory.makeWall());
  r2.setSide(OUEST,theDoor);
  return aMaze;
}
```

Convertisseur de document

- On dispose d'un lecteur de fichier RTF.
- On souhaite convertir des fichiers RTF en ASCII, latex, OpenOffice, HTML, etc.
- On souhaite pouvoir facilement ajouter de nouveaux formats de conversion sans avoir à toucher au lecteur.

Le constructeur (Builder)

Objectif Sépare la construction d'un objet complexe de sa représentation afin d'utiliser le même processus de construction pour différentes représentations.

Problème Plusieurs types d'objets complexes peuvent être construits avec la même processus général de construction, mais il y a des variantes à chaque étape individuelle de la construction.

Solution Utiliser une classe

Participants Builder spécifie une interface pour créer les parties. ConcreteBuilder construit les différentes parties du produit et permet de la récupérer une fois terminé.

Conséquence Permet de changer la représentation interne du produit. Isole le code de la construction du code de la représentation.

Référence GoF Pages 97 à 106

Diagramme de classe

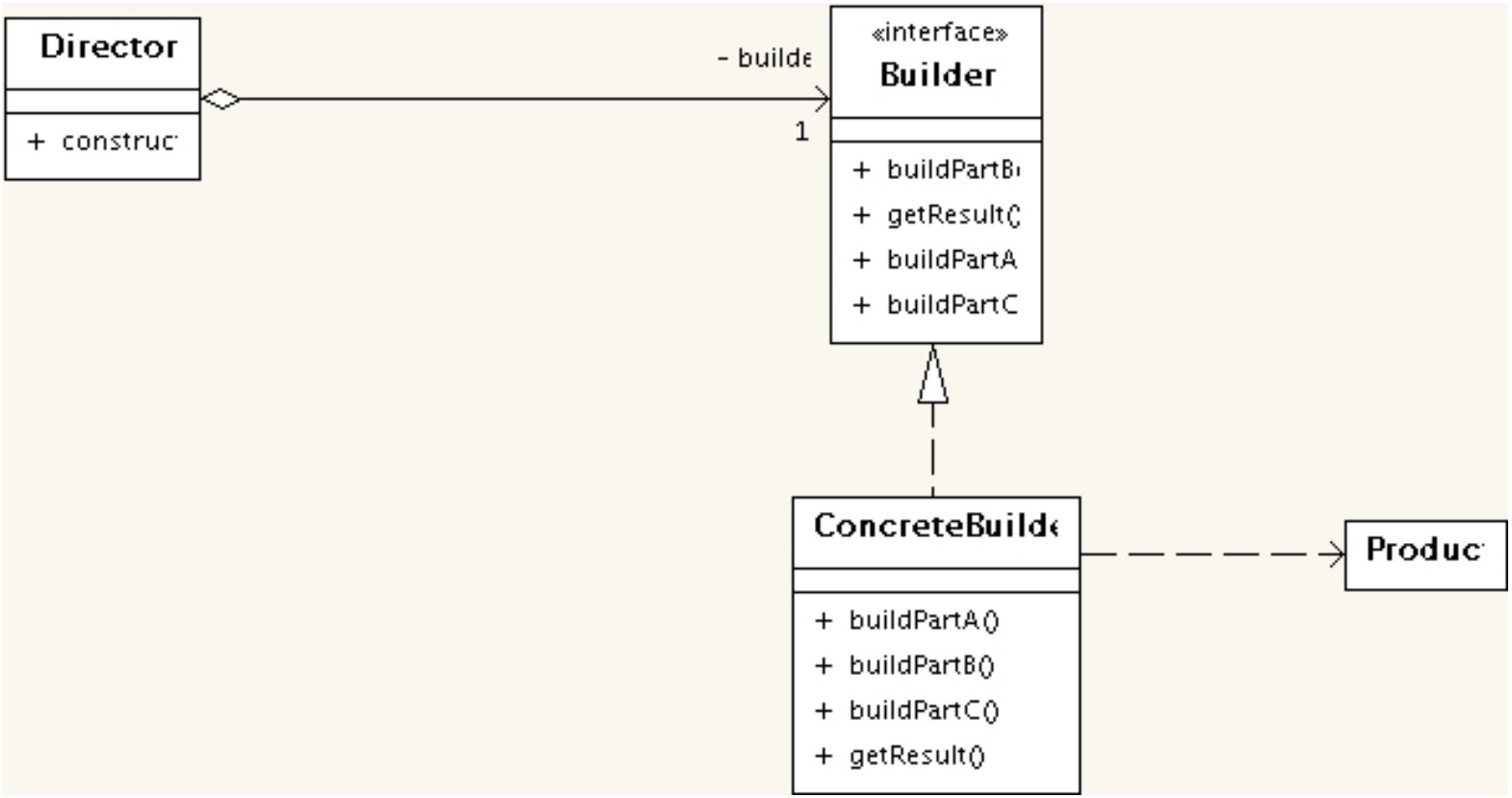
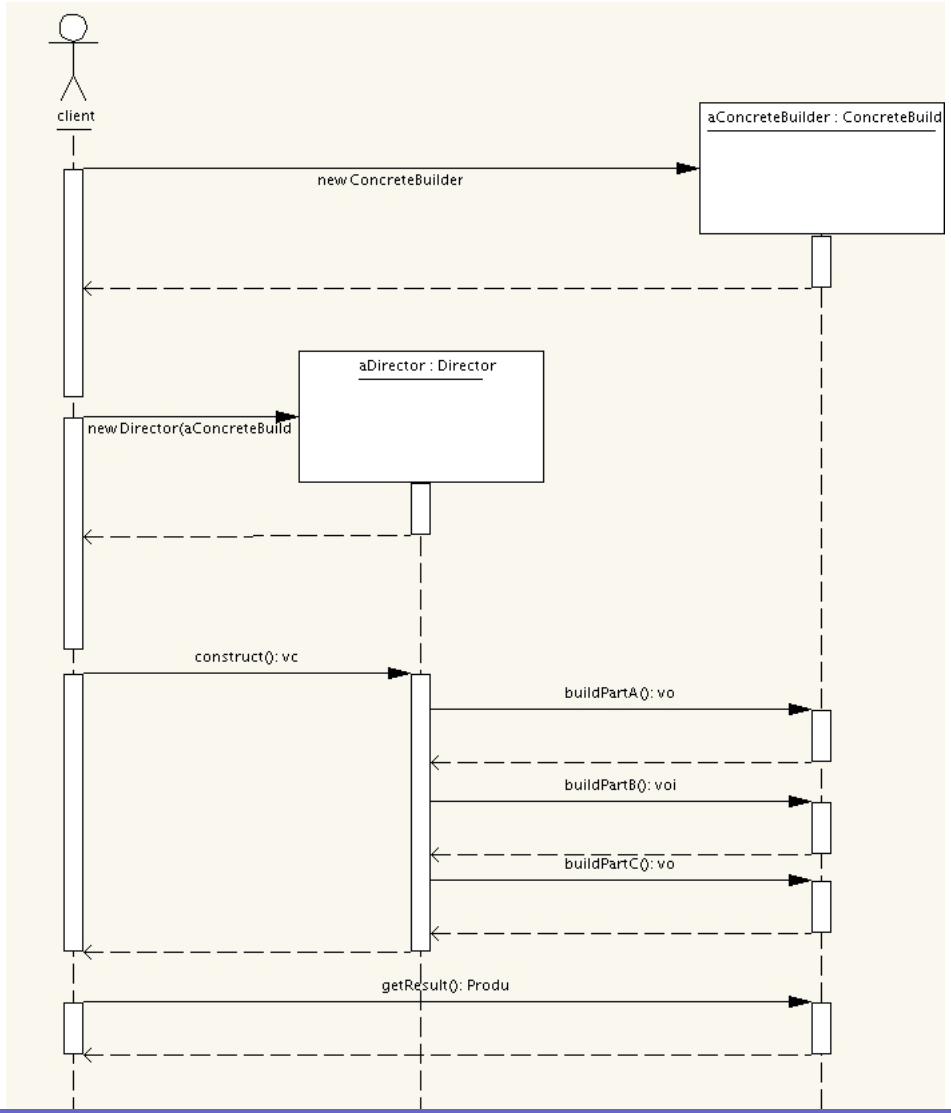


Diagramme de séquence



Application au labyrinthe 1/2

```
public interface MazeBuilder {

   void buildMaze();
   void buildRoom(int room);
   void buildDoor(int roomFrom,int roomTo);
   Maze getMaze();
}
```

Application au labyrinthe 2/3

```
public class StandardMazeBuilder implements MazeBuilder

  private Maze fCurrentMaze;

  public void buildMaze() {
   fCurrentMaze = new Maze();
  }

  public void buildRoom(int roomNo) {
   Room room = fCurrentMaze.roomNo(roomNo);
   if (room != null) {
      room = new Room(roomNo);
      fCurrentMaze.addRoom(room);
      for (Direction dir : Direction.values()) {
          room.setSide(dir, new Wall());
      }
```

Application au labyrinthe 3/3

```
    // methode utilitaire pour connaitre le mur commun
    private Direction commonWall(Room a, Room b) { ...}

    public void buildDoor(int roomFrom, int roomTo) {
     Room r1 = fCurrentMaze.roomNo(roomFrom);
     Room r2 = fCurrentMaze.roomNo(roomTo);
     Door d = new Door(r1,r2);
     r1.setSide(commonWall(r1,r2),d);
     r2.setSide(commonWall(r2,r1),d);
    }

    public Maze getMaze() {
     return fCurrentMaze;
    }
}
```

Création du labyrinthe

```
Maze createMaze(MazeBuilder builder) {
  builder.buildMaze();
  builder.buildRoom(1);
  builder.buildRoom(2);
  builder.buildDoor(1,2);

  return builder.getMaze();
}
```

Fabrication (factory method)

Objectif Définir une interface de création d'un objet, mais laisser les sous classes décider des classes à instancier.

Problème Une classe doit instancier une spécialisation d'une autre classe, sans savoir laquelle.

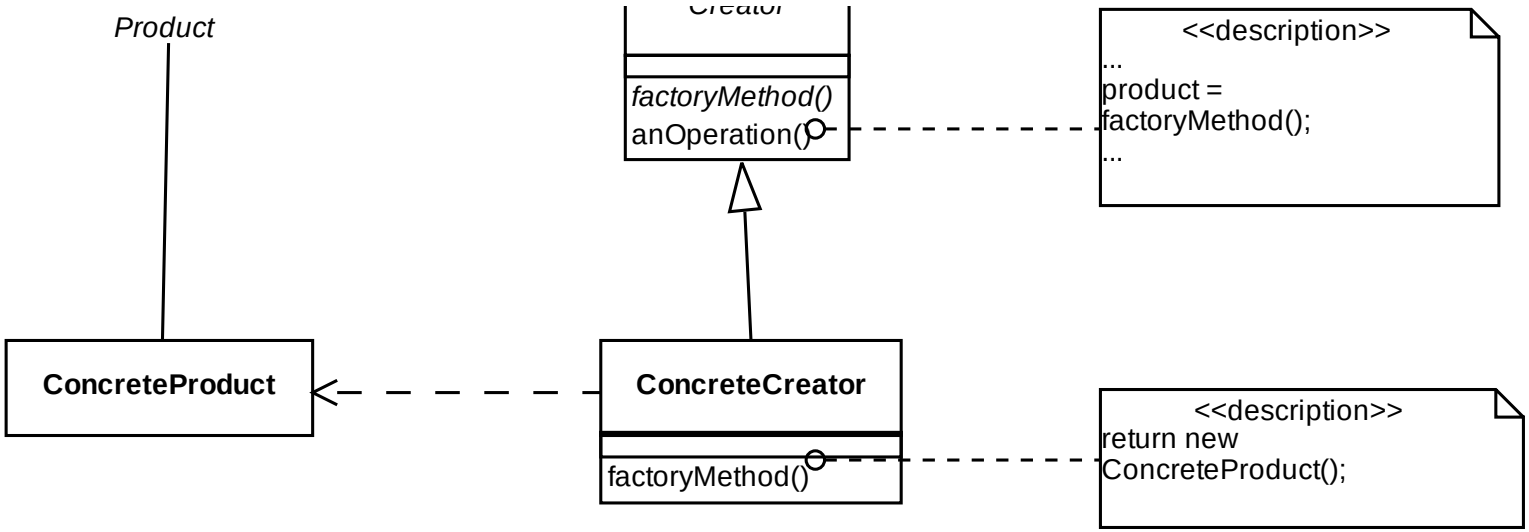
Solution Une classe spécialisée de prendre cette décision.

Participants Product est l'interface du type d'objet créé par la fabrication. Creator est l'interface qui définit la fabrication.

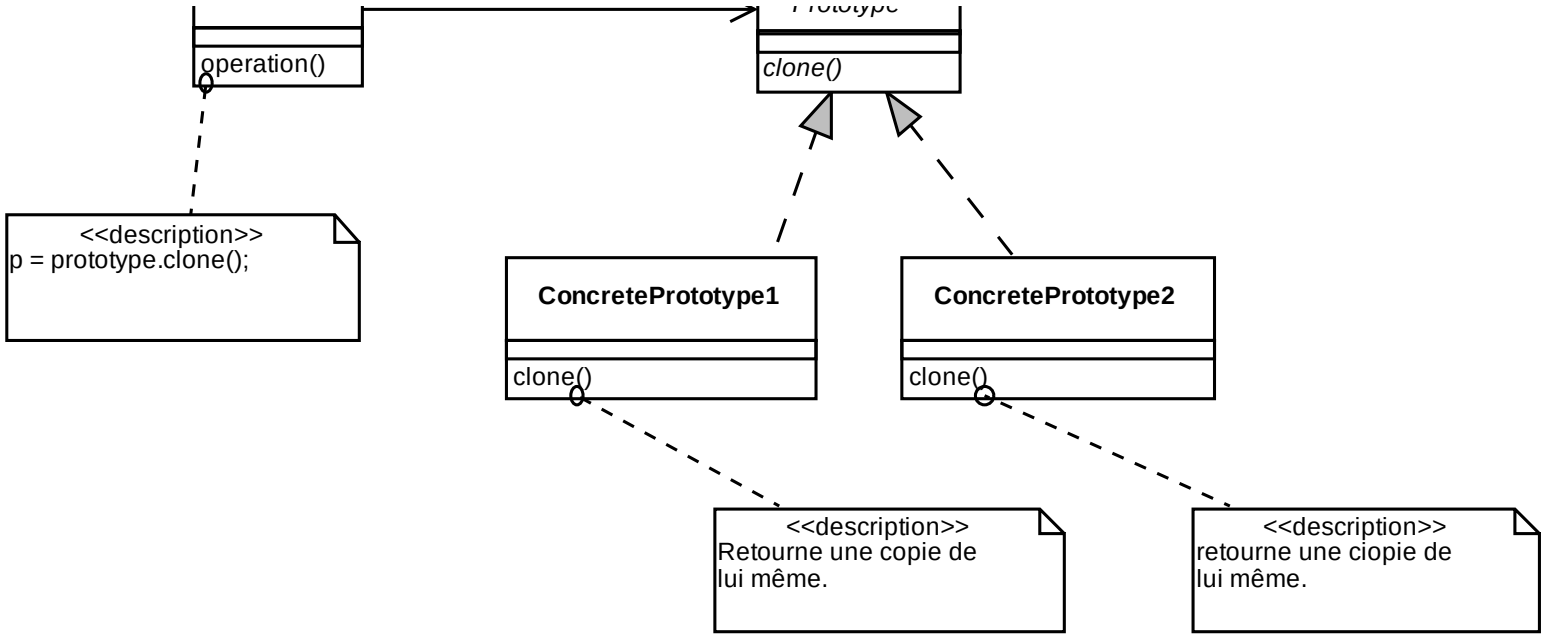
Conséquence Les clients auront besoin de créer une sous classe de la classe créateur pour créer un ProduitConcret donné.

Référence GoF Pages 87 à 96.

Fabrication



Prototype


operation()
clone()
<<description>>
p = prototype.clone();
ConcretePrototype1
clone()
ConcretePrototype2
clone()
<<description>>
Retourne une copie de
lui même.
<<description>>
retourne une ciopie de
lui même.

Le singleton

Objectif On souhaite disposer d'un seul objet sans devoir en créer un global destiné à contrôler son instanciation.

Problème Divers objets client doivent se référer à la même chose et l'on veut être sûr qu'il n'existe qu'une seule instance.

Solution S'assurer qu'il n'existe qu'une seule instance.

Participants Les clients créent une seule instance de Singleton via la méthode getInstance().

Conséquence Les clients n'ont pas besoin de chercher si une instance existe, la classe s'en charge.

Référence GoF Pages 127 à 134.

Exemple de singleton

```
public class StandardMazeFactory implements MazeFactory

 private static MazeFactory factory;

 private StandardMazeFactory() {
 }

 public static MazeFactory getInstance() {
  if (factory==null) {
   factory = new StandardMazeFactory();
  }
  return factory;
 }

 [...]
```

Exemple de singleton, version multithread

```
public class StandardMazeFactory implements MazeFactory

 private static MazeFactory factory;

 private StandardMazeFactory() {
 }
 private synchronized static void syncMethod() {
  if (factory==null) {
   factory = new StandardMazeFactory();
  }
 }
 public static MazeFactory getInstance() {
   if (factory==null) {
    syncMethod();
   }
   return factory;
```

Adapteur

Objectif Faire correspondre à une interface donnée un objet ou une classe que l'on ne contrôle pas.

Problème Un système a les bonnes données et les bonnes méthodes, mais pas les bonnes interfaces.

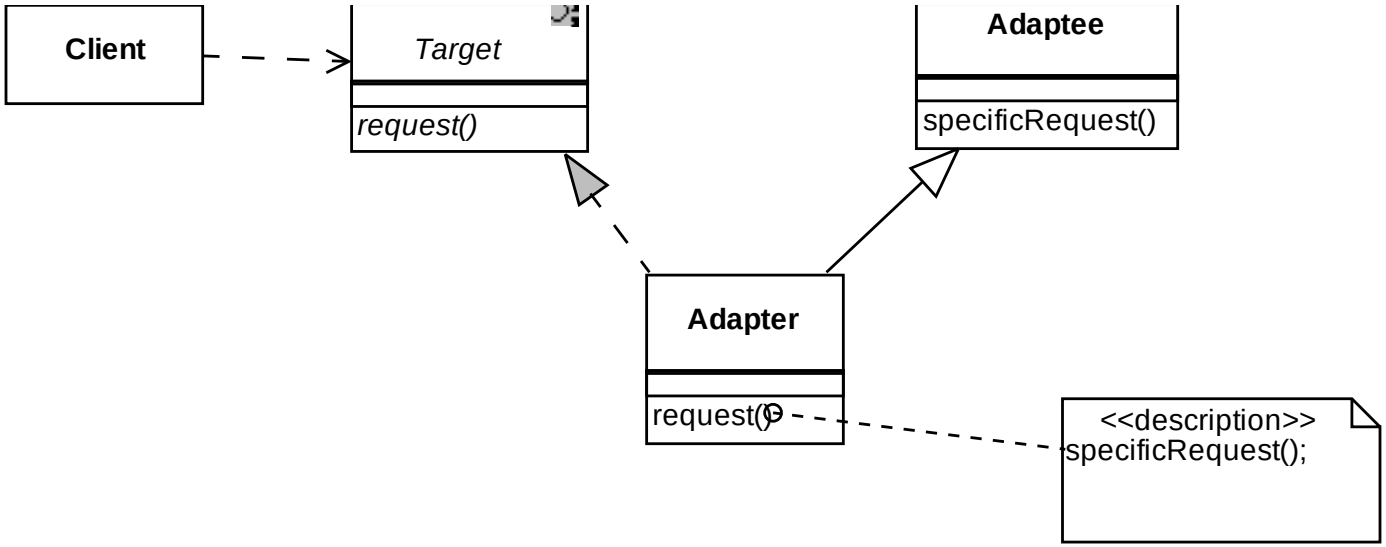
Solution Héritage (classe) ou délégation (objet).

Participants La classe `Adapter` adapte l'interface de la classe `Adaptee` pour qu'elle corresponde à la cible de l'`Adapter`

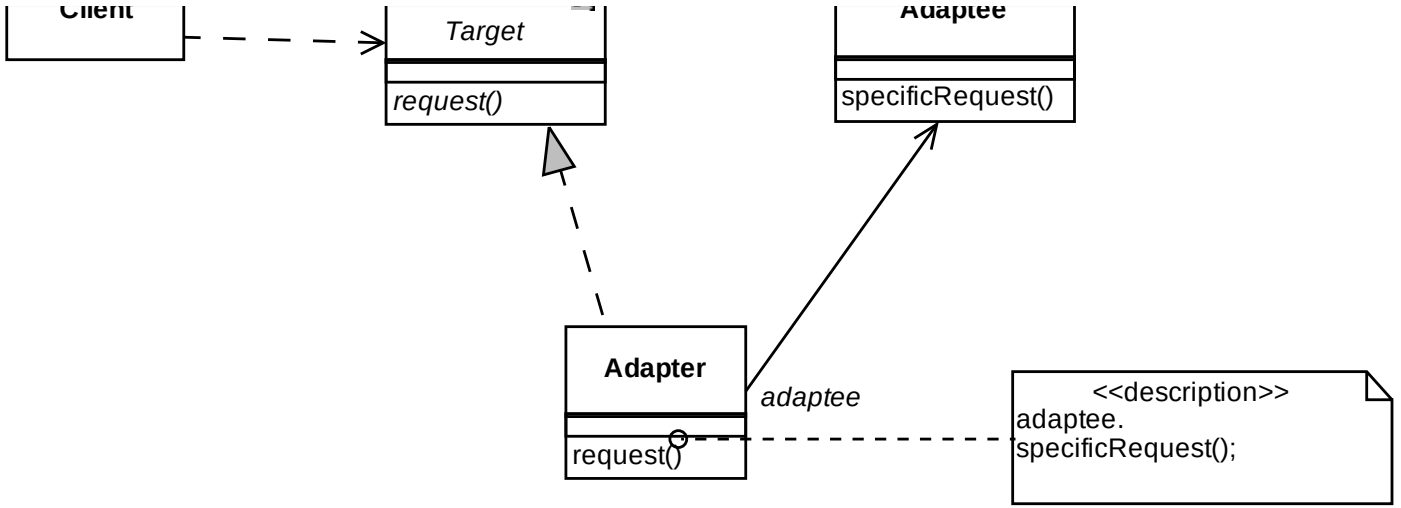
Conséquence Des objets existants peuvent être intégrés à de nouvelles structures de classes sans être limités par leur interface.

Référence GoF Pages 139 à 150.

Adapteur de classe



Adapteur Objet



Le pont

Objectif Découpler un jeu d'implantations à partir d'un jeu d'objets qui l'utilise.

Problème Les dérivations d'une classe abstraite doivent utiliser plusieurs implantations sans provoquer une augmentation du nombre de classes.

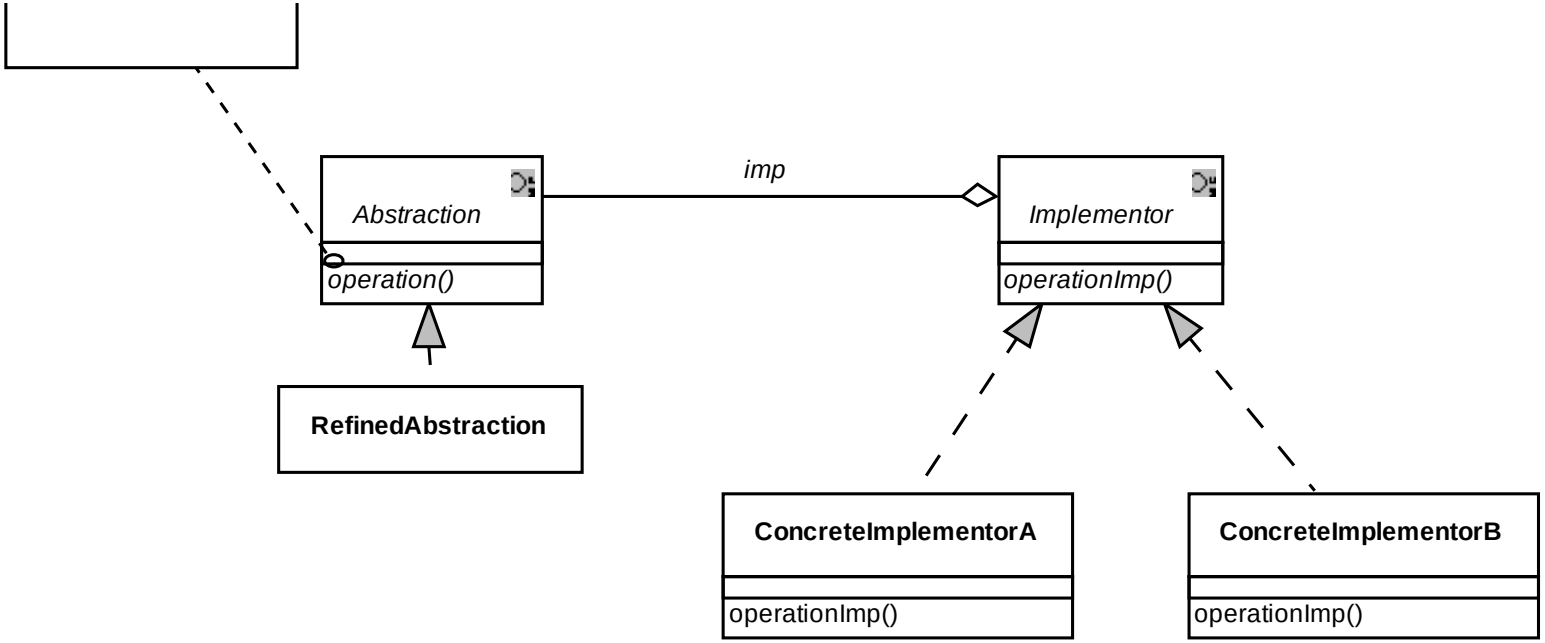
Solution Définir une interface pour toutes les implantations voulues et la faire utiliser par les dérivations de la classe abstraite.

Participants `Abstraction` définit l'interface des objets en cours d'implantation. `Implementor` définit l'interface des classes d'implantation.

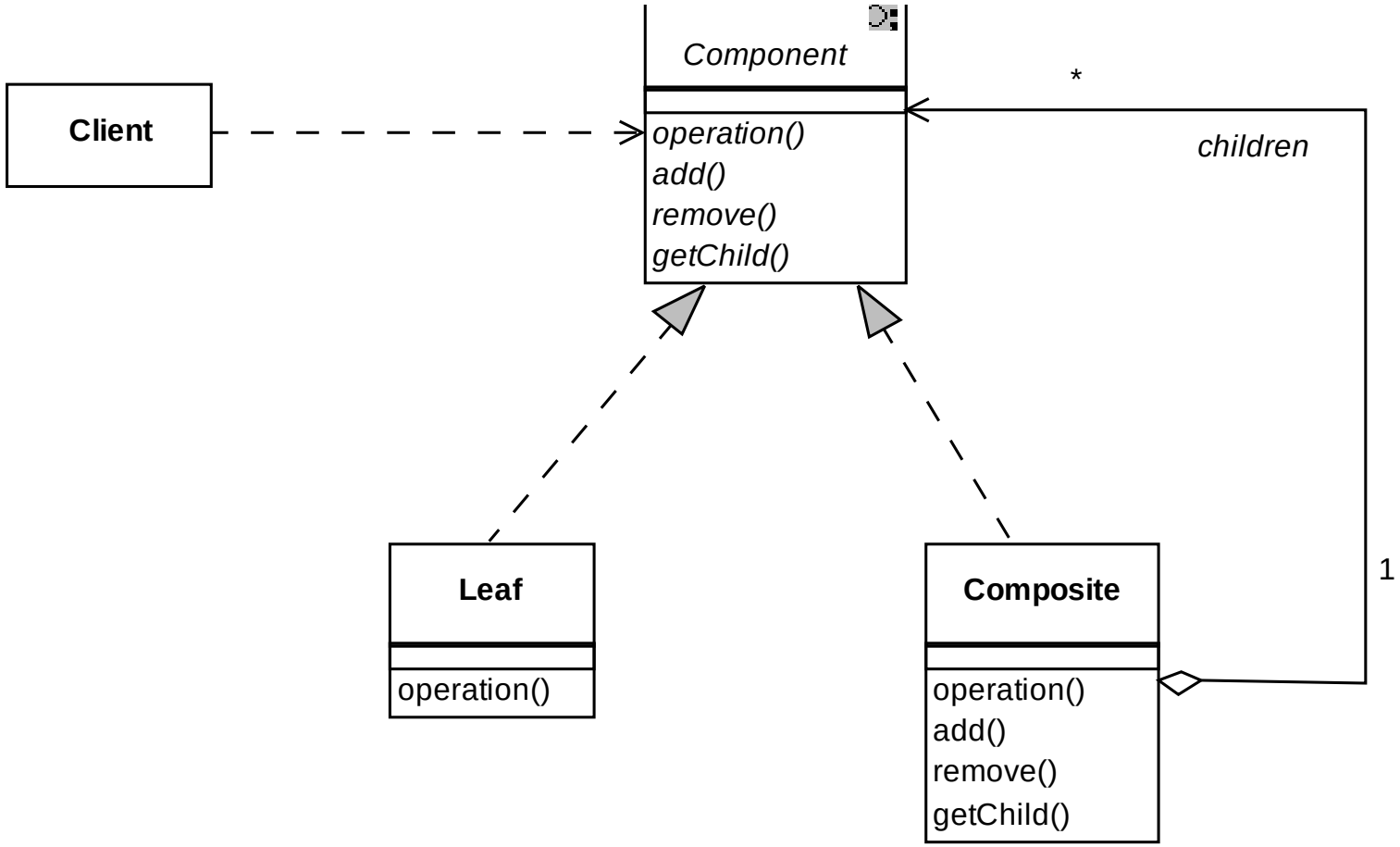
Conséquence Encapsuler les implantations dans une classe abstraite référencée dans la classe de base de l'abstraction.

Référence GoF 151 à 162

Pont


Abstraction
operation()
imp
Implementor
operationImp()
RefinedAbstraction
ConcreteImplementorA
operationImp()
ConcreteImplementorB
operationImp()

Composite


Component
operation()
add()
remove()
getChild()
Client
*
children
1
Leaf
operation()
Composite
operation()
add()
remove()
getChild()

Décorateur

Objectif Associer dynamiquement des responsabilités supplémentaires à un objet.

Problème L'objet que l'on souhaite utiliser possède les fonctions de base dont vous avez besoin. Cependant, on devra probablement lui ajouter des fonctionnalités supplémentaires qui s'appliqueront avant ou après la fonctionnalité de base.

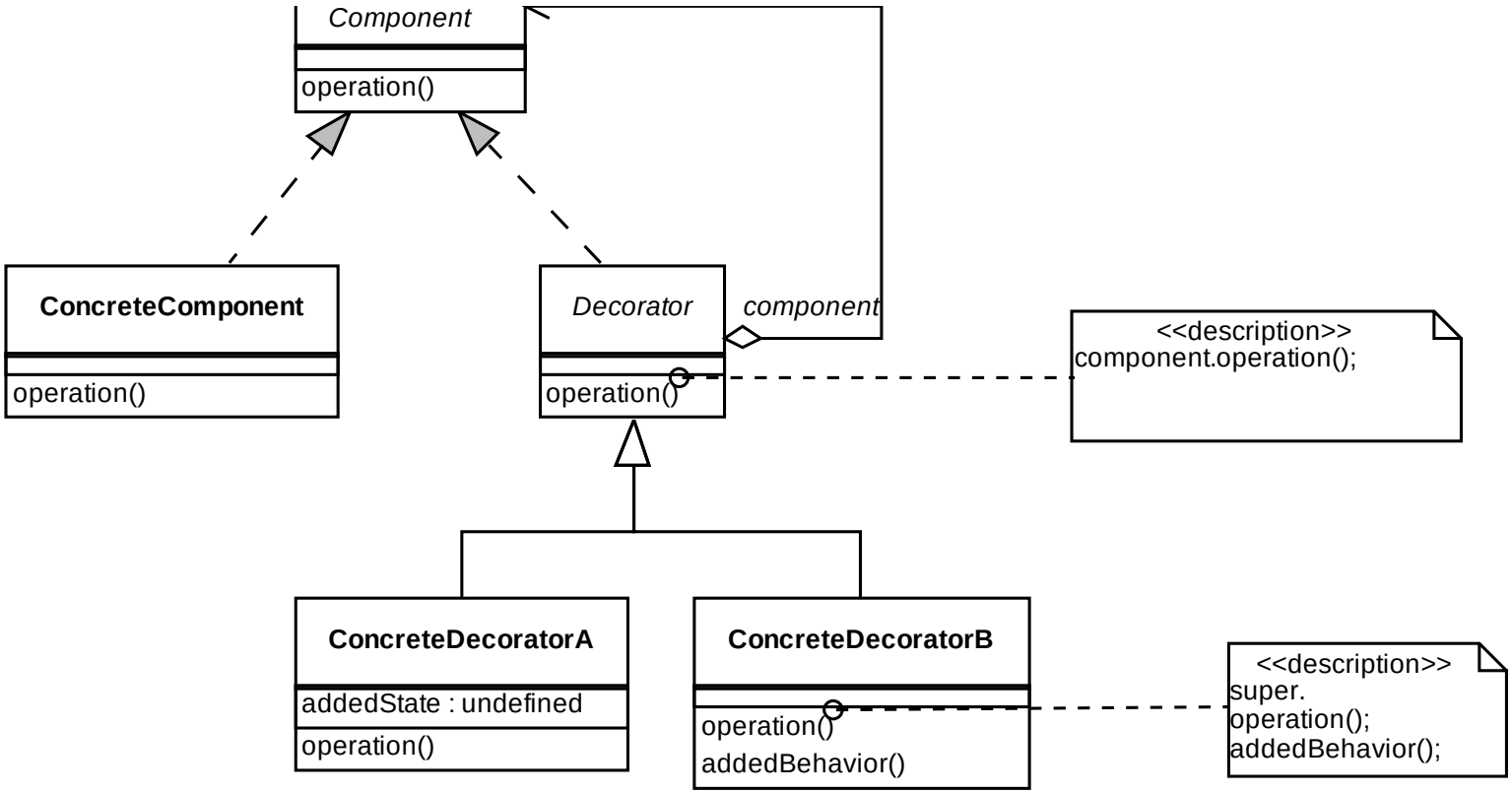
Solution Permettre l'extension de la fonctionnalité d'un objet sans avoir recours à des sous classes.

Participants

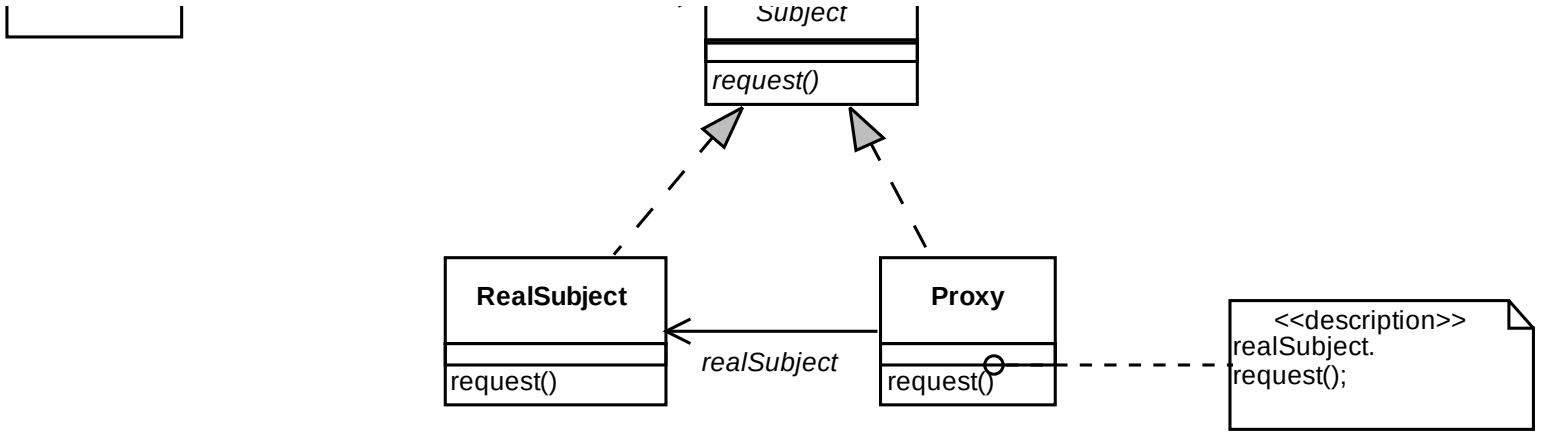
Conséquence La fonctionnalité à ajouter se trouve dans des composants de petite taille, ce qui permet au décorateur de l'ajouter dynamiquement avant ou après la fonctionnalité d'un ComposantConcret.

Référence GoF 175 à 184

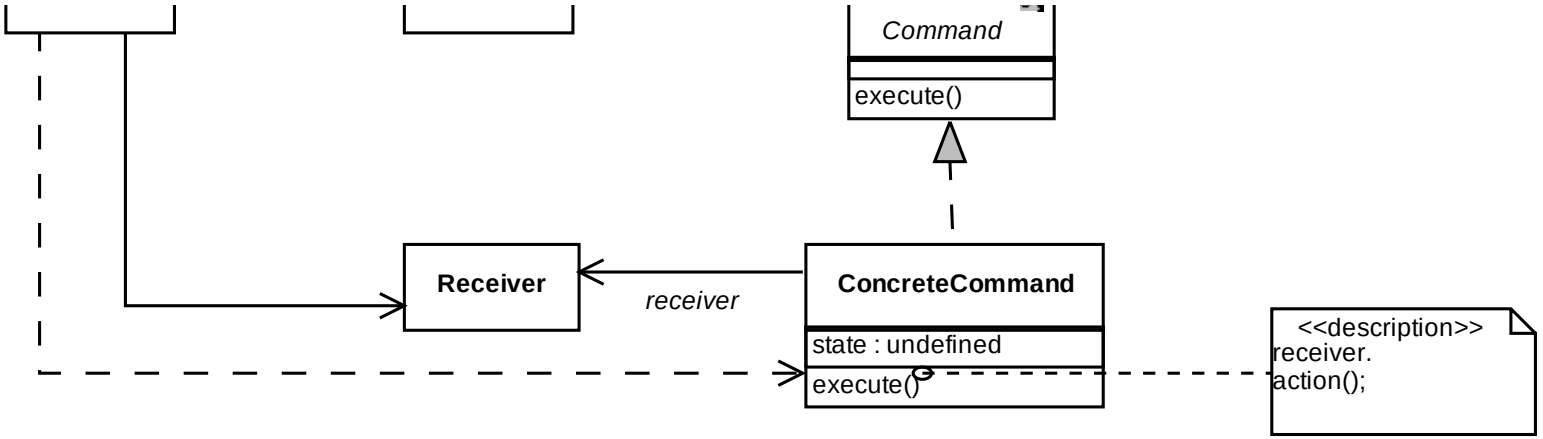
Décorateur


Component
operation()
ConcreteComponent
operation()
Decorator
component
operation()
<<description>>
component.operation();
ConcreteDecoratorA
addedState : undefined
operation()
ConcreteDecoratorB
operation()
addedBehavior()
<<description>>
super.
operation();
addedBehavior();

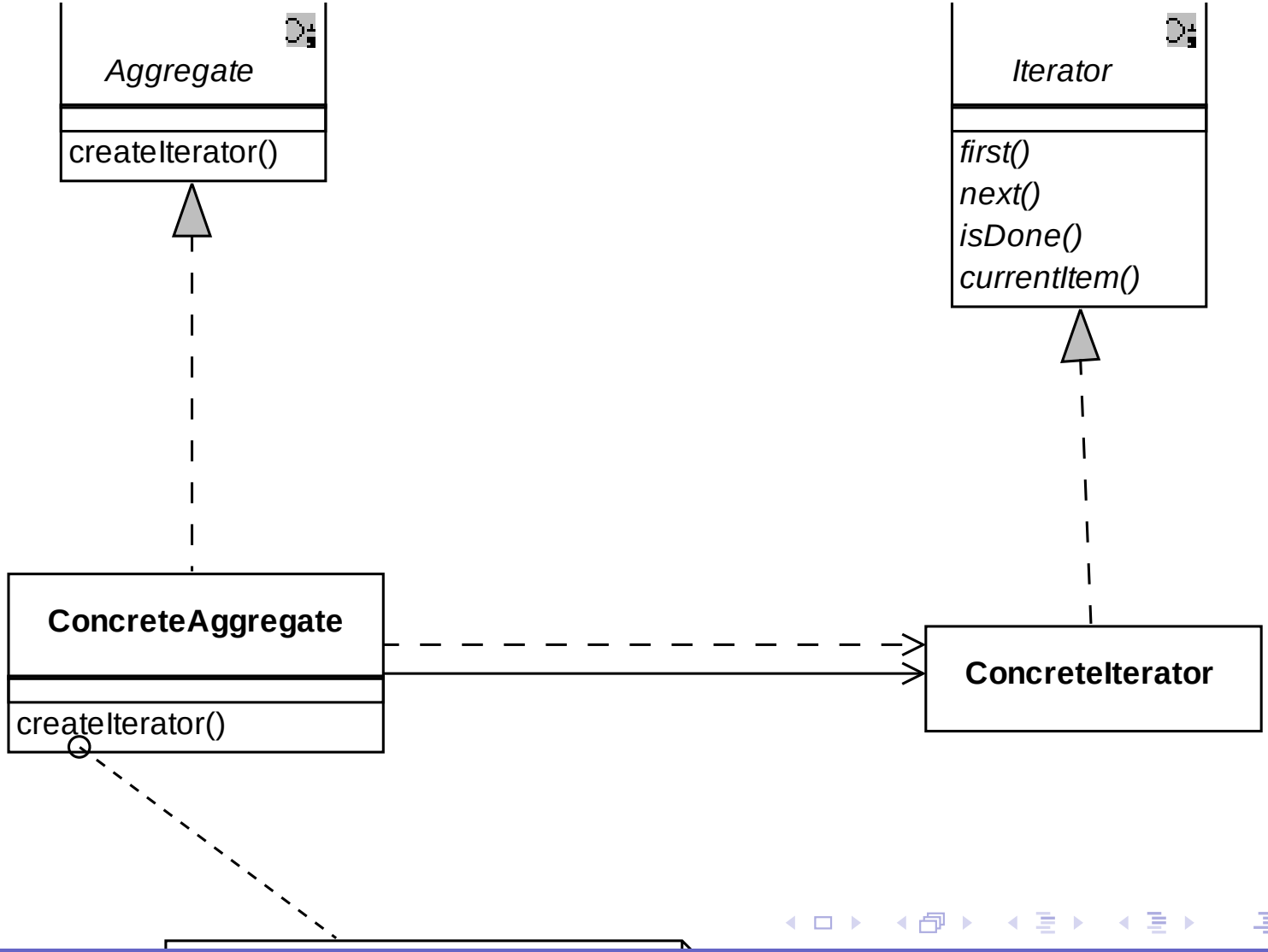
Proxy


Subject
request()
RealSubject
request()
realSubject
Proxy
request()
<<description>>
realSubject.
request();

Commande


Command
execute()
Receiver
receiver
ConcreteCommand
state : undefined
execute()
<<description>>
receiver.
action();

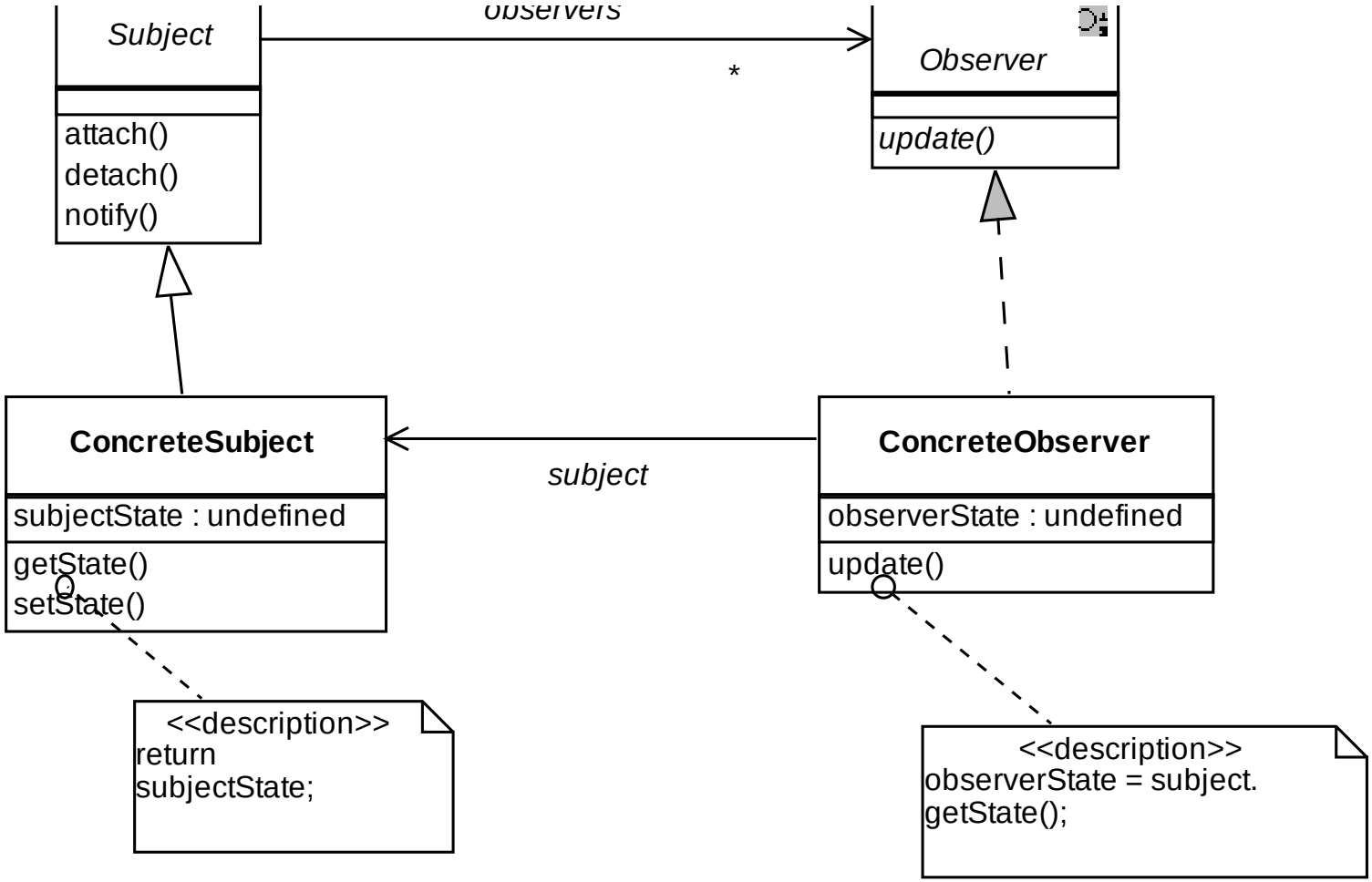
Iterateur


Aggregate
createIterator()
Iterator
first()
next()
isDone()
currentItem()
ConcreteAggregate
createIterator()
ConcreteIterator

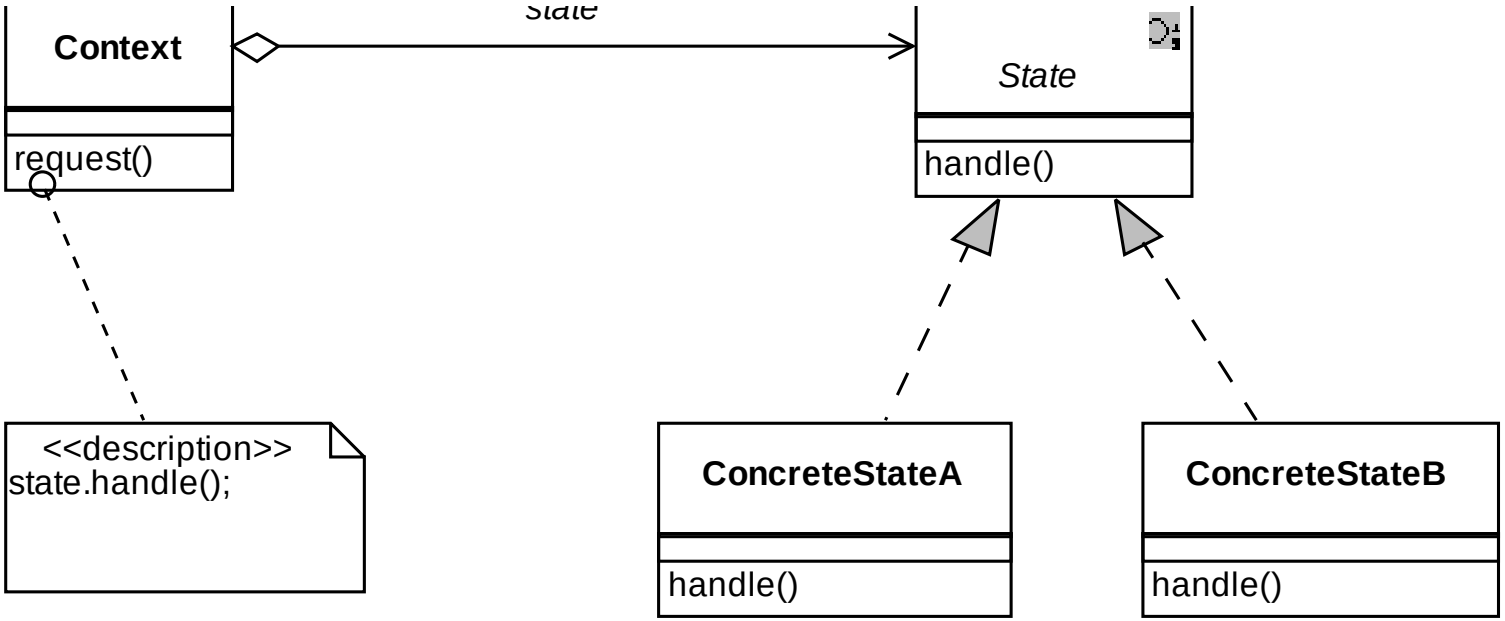
Observateur

Objectif	Défiir une dépendance “un à plusieurs” entre des objets pour que tous ceux qui dépendent d’un objet modifié soient avertis du changement d’état et mis à jour automatiquement.
Problème	il faut avertir une liste variable d’objets qu’un événement à eu lieu.
Solution	Les objets Observer délèguent la responsabilité de contrôle d’un événement à un objet central, le sujet.
Participants	le sujet connaît les observateurscar ils s’enregistrent auprès de lui.Il doit les avertir dès qu’un événement à lieu.
Conséquence	
Référence GoF	293 à 303

Observateur



Etat


state
Context
request()
State
handle()
<<description>>
state.handle();
ConcreteStateA
handle()
ConcreteStateB
handle()

Stratégie

Objectif Permettre d'utiliser différentes règles métier ou algorithmes identiques sur le plan conceptuel en fonction du contexte.

Problème La sélection de l'algorithme dépend du client à l'origine de la demande ou des données à traiter.

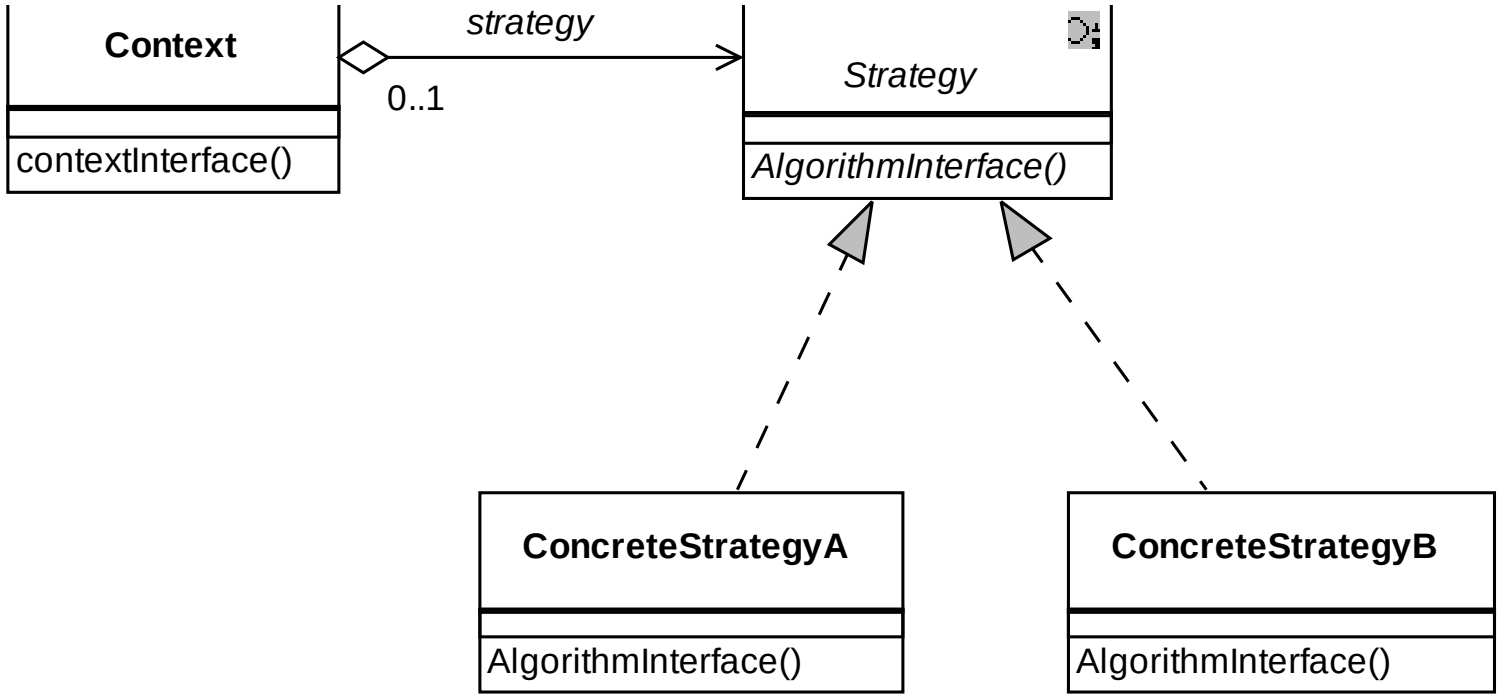
Solution Séparer la sélection de l'algorithme et son implantation.

Participants

Conséquence Le patron Stratégie définit une famille d'algorithmes.

Référence GoF

Strategie


Context
contextInterface()
strategy
0..1
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()


Kent Beck.
eXtreme Programming eXplained, Embrace Change.
Addison Wesley, 2000.


B. Boehm.
Software Engineering Project Management, chapter A spiral model of software development and enhancement.
1987.


Bernd Bruegge and Allen H. Dutoit.
Object Oriented Software Engineering: Conquering Complex and Changing Systems.
Prentice Hall, 2000.
ISBN: 0-13-017452-1.


Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Object Technology Series. Addison Wesley, 2000.

Indispensable !

Martin Fowler.
Patterns of Enterprise Application Architecture.
The Addison-Wesley Signature Series. Addison Wesley, 2003.

Martin Fowler.
UML distilled: A brief Guide to the Standard Object Modelling Language.
Object Technology series. Addison Wesley, 3rd edition, 2004.
Indispensable !

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns, Elements of Reusable Object-Oriented Software.
Addison Wesley, 1995.
A posséder absolument!


Marie-Claude Gaudel, Bruno Marre, Françoise Schienger, and Gilles Bernot.
Précis de Génie Logiciel.
Enseignement de l'Informatique. Masson, 1996.
ISBN : 2-225-85189-1.


Andrew Hunt and David Thomas.
The Pragmatic Programmer.
Addison Wesley, 2000.
Un livre interessant (mais en anglais) pour tous les programmeurs.


Robert C. Martin.
Agile Software Development: Principles, Patterns and Practices.
Prentice Hall, 2003.
Utile.


Robert C. Martin.
UML for Java Programmers.
Prentice Hall, 2003.


Bertrand Meyer.
Conception et programmation orientées objet.
Eyrolles, 1997.
Indispensable pour tout programmeur dans un langage objet, même si les exemples donnés sont plutôt en Eiffel.


P. G. Neumann.
Computer-Related Risks.
Reading. Addison-Wesley, MA, 1995.


W. W. Royse.
Managing the development of large software systems.

In IEEE Computer Society, editor, *Tutorial: Software Engineering Project Management*, pages 118–127, Washington, DC, 1970.

Jack Shirazi.
Java Performance Tuning, 2nd edition.
O'Reilly, 2003.