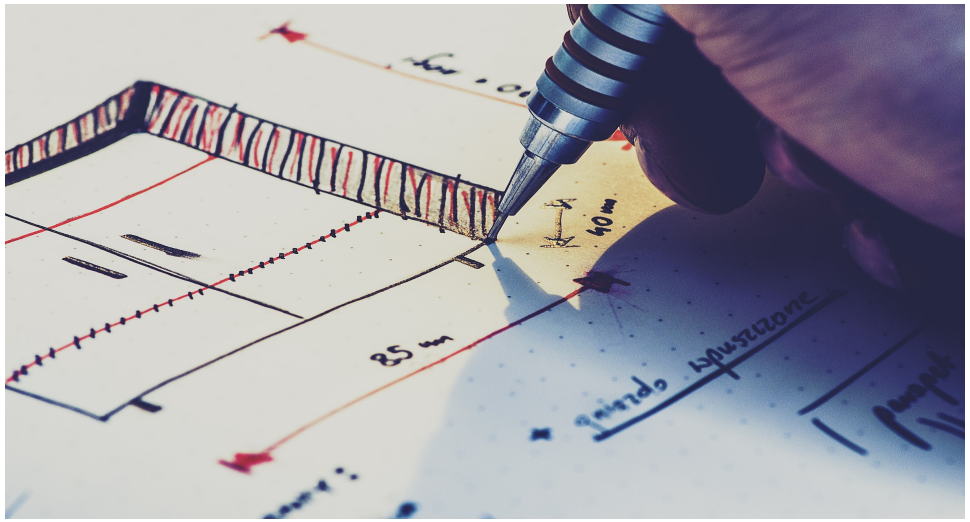
Partie 1

Patrons (et anti-patrons)



de conception

Plan

- Introduction
- Premier exemple
- Principes et classification
- Les patrons les plus significatifs
- Quelques anti-patrons

Motivations

- Les besoins pour une bonne conception et du bon code :
  - Extensibilité
  - Flexibilité
  - Maintenabilité
  - Réutilisabilité
  - ☞ Les qualités *internes*
    - ☞Meilleure spécification, construction, documentation

Motivations (suite)

- Augmenter la cohésion du code
  - La cohésion est le degré pour lequel les différentes données gérées dans une classe ou une fonction sont reliées entre elles. C'est le degré de corrélation des données entre elles.
- Diminuer le couplage
  - Le couplage est le nombre de liens entre les données de classes ou fonctions différentes. On peut faire un compte du couplage en comptant le nombre de références ou d'appels fait à l'objet d'un autre type.
- On minimise le couplage en maximisant la cohésion, et en créant des interfaces qui seront les points centraux d'accès aux autres données.

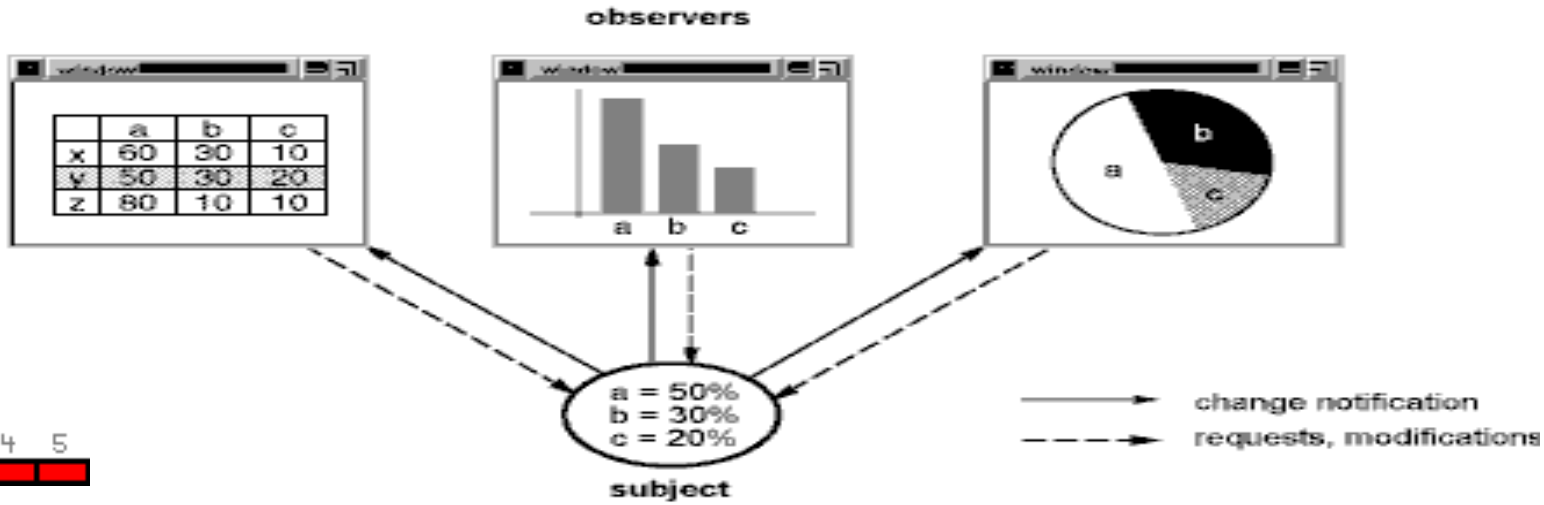
Design Pattern : principe

- Description d'une solution à un problème général et récurrent de conception dans un contexte particulier
  - Description des objets communicants et des classes
  - Indépendant d'une application ou spécifique
  - Certains patterns sont relatifs à la concurrence, à la programmation distribuée, temps-réel
- *Traduction : patron de conception*
- Tous les patrons visent à renforcer la cohésion et à diminuer le couplage

Un exemple :

Observer (comportement)

- **Intention**
  - Définir une dépendance 1-N de telle façon que si l'objet change d'état tous ses dépendants sont prévenus et mis à jour automatiquement
- **Synonymes :** Dépendants, Publier/Abonner, Publish/Subscribe
- **Motivation**
  - Un effet secondaire de découper un logiciel en classes coopératives est la nécessité de maintenir la cohérence des objets associés. Le faire par des classes **fortement couplées** entrave leur réutilisabilité



Fréquence :

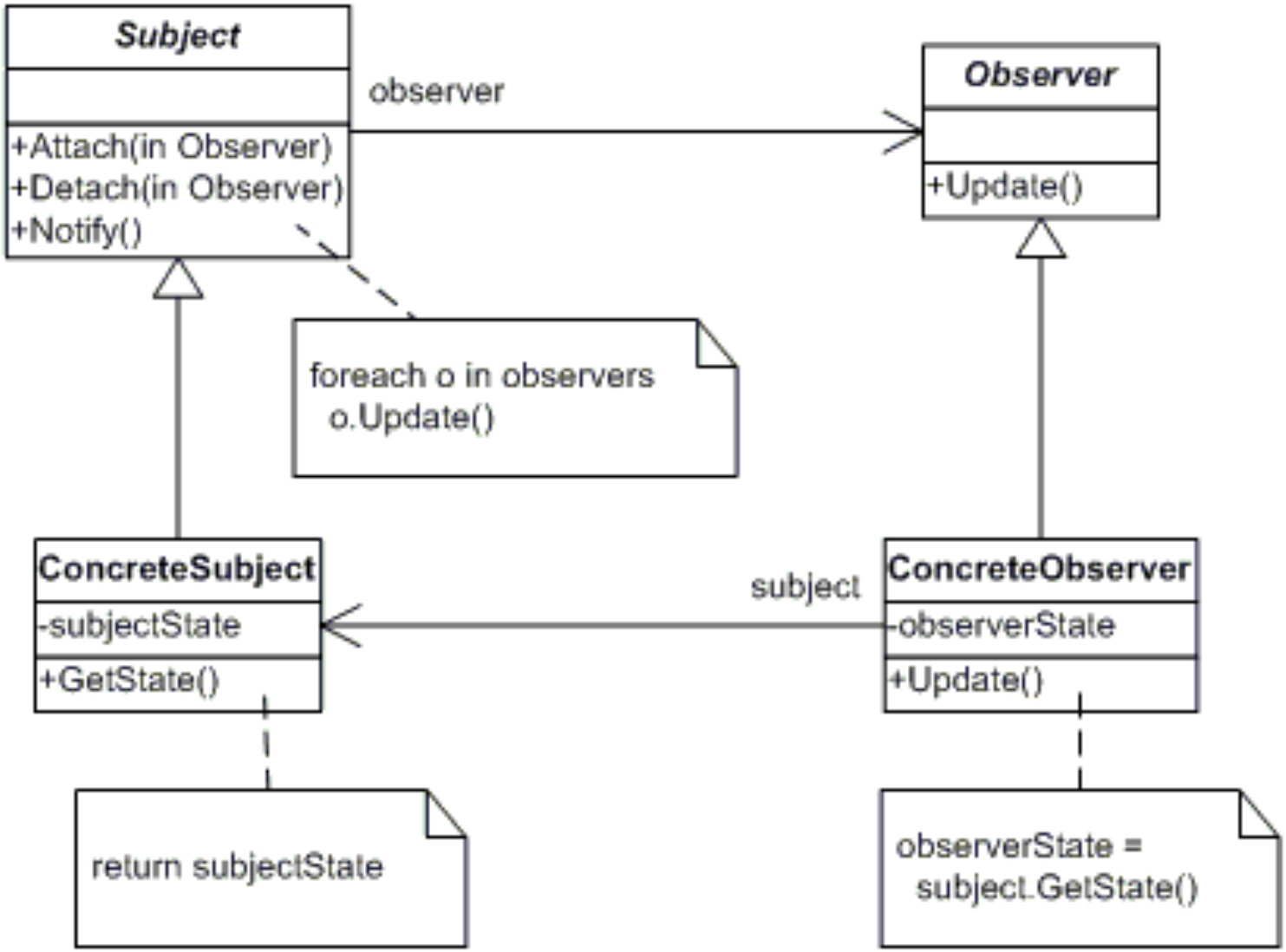
1 2 3 4 5

Observer (2)

- **Champs d'application**
  - Quand une abstraction a deux aspects, un dépendant de l'autre. Encapsuler ces aspects dans des objets séparés permet de les faire varier et de les réutiliser indépendamment
  - Quand un changement sur un objet nécessite de modifier les autres et qu'on ne peut savoir combien d'objets doivent être modifiés
  - Quand un objet doit être capable de notifier d'autres objets sans faire de suppositions sur qui sont ces objets, c'est-à-dire que les objets ne doivent pas être fortement couplés

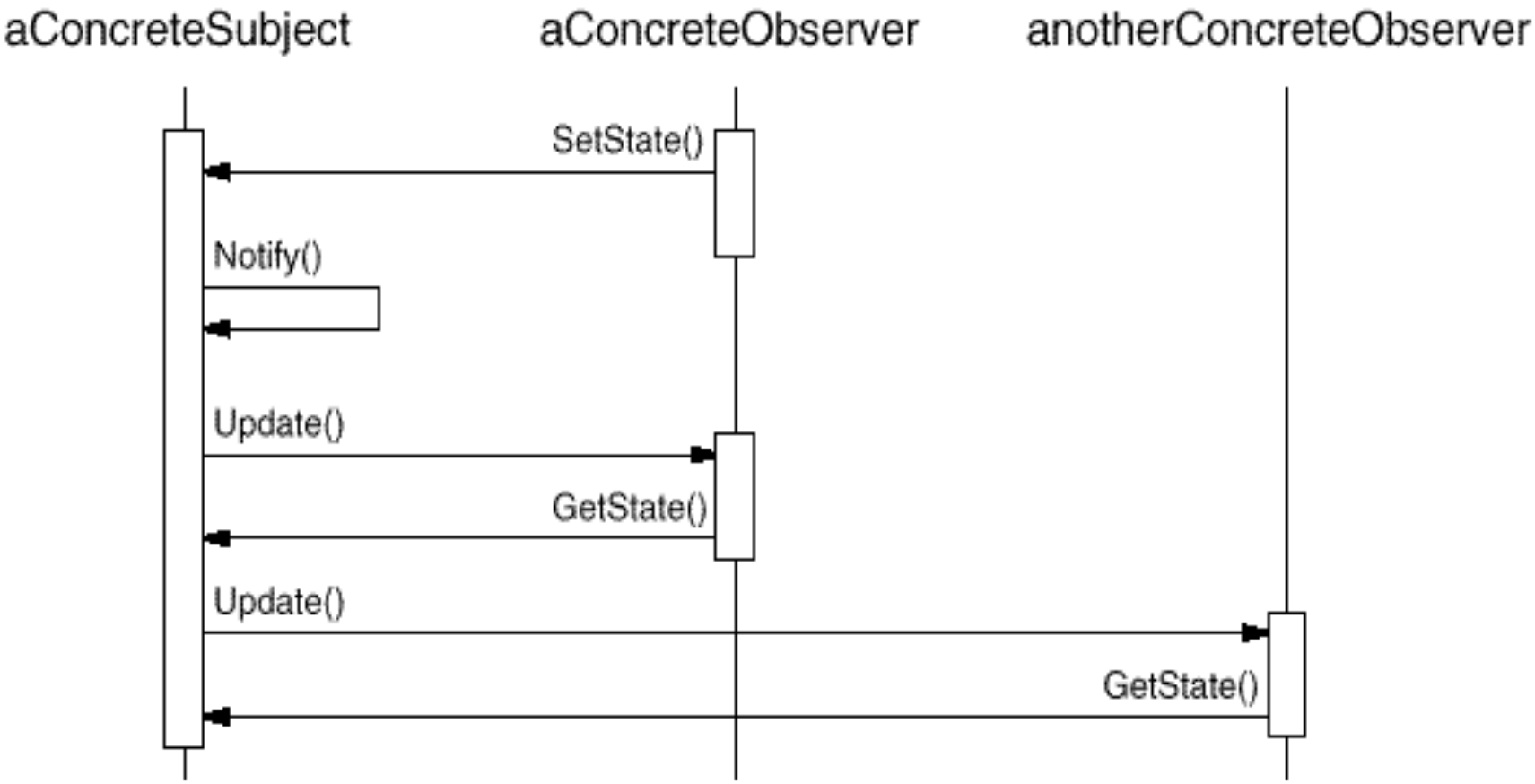
Observer (3)

- **Structure**



Observer (4)

- **Collaborations**



Observer (5)

- **Conséquences**
  - Variations abstraites des sujets et observateurs
  - Couplage abstrait entre le sujet et l'observateur
  - Prise en charge de communication broadcast
  - Mises à jour non prévues
    - Protocole additionnel pour savoir ce qui a précisément changé

Observer (6)

- **Implémentation**
  - Référence sur les observateurs, *hash-table*
  - Un observateur peut observer plusieurs sujets
    - Étendre le *update* pour savoir qui notifie
  - Attention lors de la suppression des observés
  - Attention à la consistance de l'observé avant notification
  - Attention aux informations de notification
- **Patterns associés**
  - Mediator, Singleton

Observer (Code Example)

```java
public class Hobbits implements WeatherObserver {

  private static final Logger LOGGER = LoggerFactory.getLogger(Hobbits.class);

  @Override
  public void update(WeatherType currentWeather) {
    switch (currentWeather) {
      case COLD:
        LOGGER.info("The hobbits are shivering in the cold weather.");
        break;
      case RAINY:
        LOGGER.info("The hobbits look for cover from the rain.");
        break;
      case SUNNY:
        LOGGER.info("The happy hobbits bade in the warm sun.");
        break;
      case WINDY:
        LOGGER.info("The hobbits hold their hats tightly in the windy weather.");
        break;
      default:
        break;
    }
```

```java
public interface WeatherObserver {

  void update(WeatherType currentWeather);

}
```

Observer (Code Example)

```java
public class Weather {
  private WeatherType currentWeather;
  private List<WeatherObserver> observers;

  public Weather() {
    observers = new ArrayList<>();
    currentWeather = WeatherType.SUNNY;
  }

  public void addObserver(WeatherObserver obs) { observers.add(obs); }

  public void removeObserver(WeatherObserver obs) { observers.remove(obs); }

  /**
   * Makes time pass for weather
   */
  public void timePasses() {
    WeatherType[] enumValues = WeatherType.values();
    currentWeather = enumValues[(currentWeather.ordinal() + 1) % enumValues.length];
    LOGGER.info("The weather changed to {}.", currentWeather);
    notifyObservers();
  }

  private void notifyObservers() {
    for (WeatherObserver obs : observers) {
      obs.update(currentWeather);
    }
  }
```

```java
public static void main(String[] args) {

  Weather weather = new Weather();
  weather.addObserver(new Orcs());
  weather.addObserver(new Hobbits());

  weather.timePasses();
```

Design Patterns : bénéfices globaux

- Standardisation de certains concepts en modélisation
- *Capture* de l'expérience de conception
- Réutilisation de solutions élégantes et efficaces vis-à-vis du problème
- Amélioration de la documentation
- Facilité de maintenance

Historique & définition

- *Gang of Four* : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison Wesley, 1994

Un Design Pattern nomme, abstrait et identifie les aspects essentiels d'une structuration récurrente, ce qui permet de créer une modélisation orientée objet réutilisable

Classification des Patterns

Objectif				
		Création	Structure	Comportement
Portée	Classe	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Objet	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Facade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Quelques patrons de création

Factory Method (création)

- **Intention**
  - Définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier
  - Permet à une classe de déléguer l'instanciation à des sous-classes
- **Motivation**
  - instancier des classes, en connaissant seulement les classes abstraites
- **Synonymes**
  - Constructeur virtuel
- **Champs d'application**
  - une classe ne peut anticiper la classe de l'objet qu'elle doit construire
  - une classe délègue la responsabilité de la création à ses sous-classes, tout en concentrant l'interface dans une classe unique

Fréquence :

Factory Method (2)

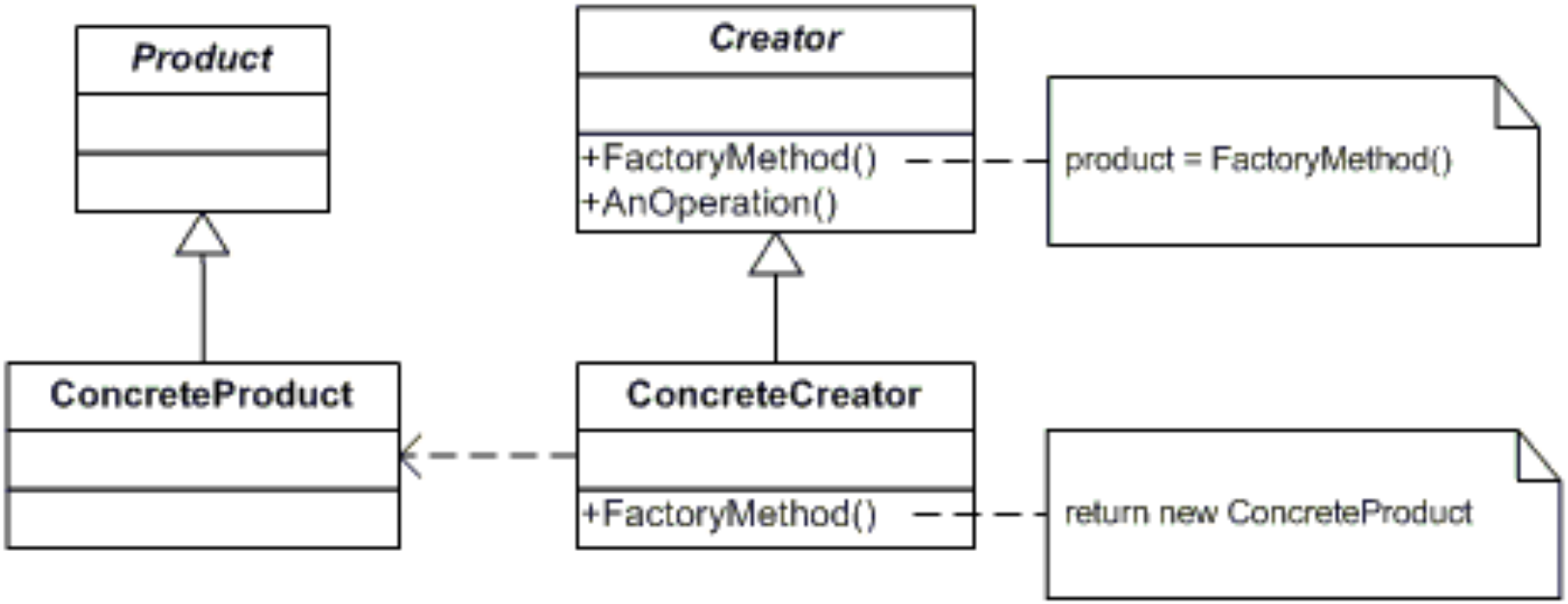
```
public interface Blacksmith {
  Weapon manufactureWeapon(WeaponType weaponType);
}

public class ElfBlacksmith implements Blacksmith {
  public Weapon manufactureWeapon(WeaponType weaponType) {
    return new ElfWeapon(weaponType);
  }
}

public class OrcBlacksmith implements Blacksmith {
  public Weapon manufactureWeapon(WeaponType weaponType) {
    return new OrcWeapon(weaponType);
  }
}
```

```
Blacksmith blacksmith = new ElfBlacksmith();
blacksmith.manufactureWeapon(WeaponType.SPEAR);
blacksmith.manufactureWeapon(WeaponType.AXE);
```

https://github.com/iluwatar/java-design-patterns/tree/master/factory-method

Factory Method (3)

- **Structure**



- **Participants**
  - **Product** (Document) définit l'interface des objets créés par la fabrication
  - **ConcreteProduct** (MyDocument) implémente l'interface Product

Factory Method (4)

- **Creator** (Application) déclare la fabrication; celle-ci renvoie un objet de type Product. Le Creator peut également définir une implémentation par défaut de la fabrication, qui renvoie un objet ConcreteProduct par défaut. Il peut appeler la fabrication pour créer un objet Product
- **ConcreteCreator** (MyApplication) surcharge la fabrication pour renvoyer une instance d'un ConcreteProduct

- **Conséquences**
  - La fabrication dispense d'avoir à incorporer à son code des classes spécifiques de l'application. Le code ne concerne que l'interface Product, il peut donc fonctionner avec toute classe ConcreteProduct définie par l'utilisateur
  - La création d'objets à l'intérieur d'une classe avec la méthode fabrication est toujours plus flexible que la création d'un objet directement

Factory Method (5)

- **Implémentation**
  - 2 variantes principales :
    - La classe Creator est une classe abstraite et ne fournit pas d'implémentation pour la fabrication qu'elle déclare (les sous-classes définissent obligatoirement une implémentation)
    - La classe Creator est une classe concrète qui fournit une implémentation par défaut pour la fabrication
  - Fabrication paramétrée :
    - La fabrication utilise un paramètre qui identifie la variété d'objet à créer
- **Utilisations connues**
  - Applications graphiques, un peu partout…
- **Patterns associés**
  - Abstract Factory, Template Method, Prototype

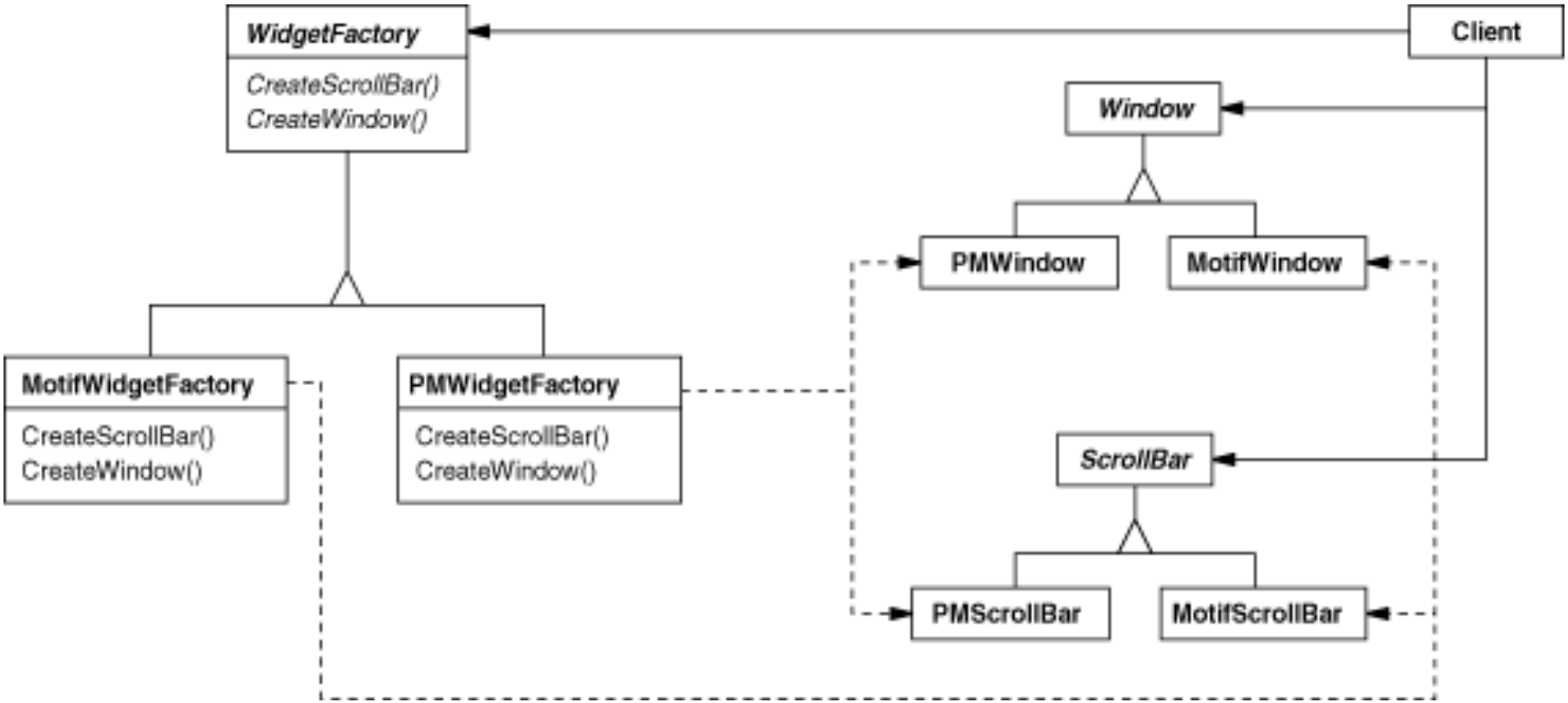
Fréquence :

Abstract Factory (création)

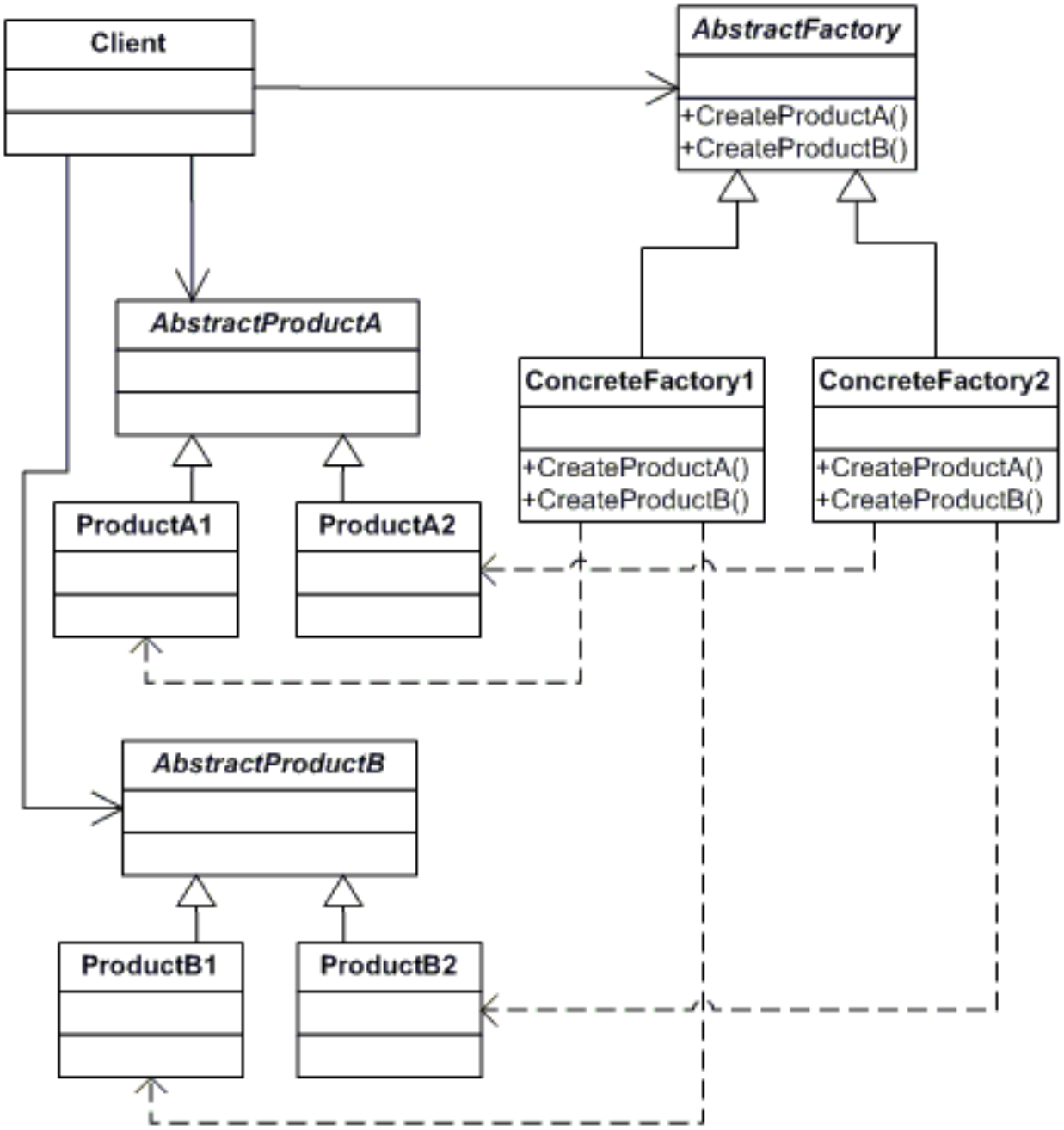
- **Intention**
  - Fournir une interface pour créer des familles d'objets dépendants ou associés sans connaître leur classe réelle
- **Synonymes :** Kit, Fabrique abstraite, Usine abstraite
- **Motivation**
  - Un éditeur qui va produire plusieurs représentations d'un document
- **Champs d'application**
  - Indépendance de comment les objets/produits sont créés, composés et représentés
  - Configuration d'un système par une instance d'une multitude de familles de produits

Abstract Factory (2)

- Conception d'une famille d'objets pour être utilisés ensemble et contrôle de cette contrainte
- Bibliothèque fournie avec seulement leurs interfaces, pas leurs implémentations



- **Structure**


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

Abstract Factory (4)

- **Participants**
  - *AbstractFactory* déclare l'interface pour les opérations qui créent des objets abstraits
  - *ConcreteFactory* implémente les opérations qui crée les objets concrets
  - *AbstractProduct* déclare une interface pour un type d'objet
  - *ConcreteProduct* définit un objet qui doit être créé par la fabrique concrète correspondante et implémente l'interface *AbstractProduct*
  - *Client* utilise seulement les interfaces déclarée par *AbstractFactory* et par les classes *AbstractProduct*

Abstract Factory (5)

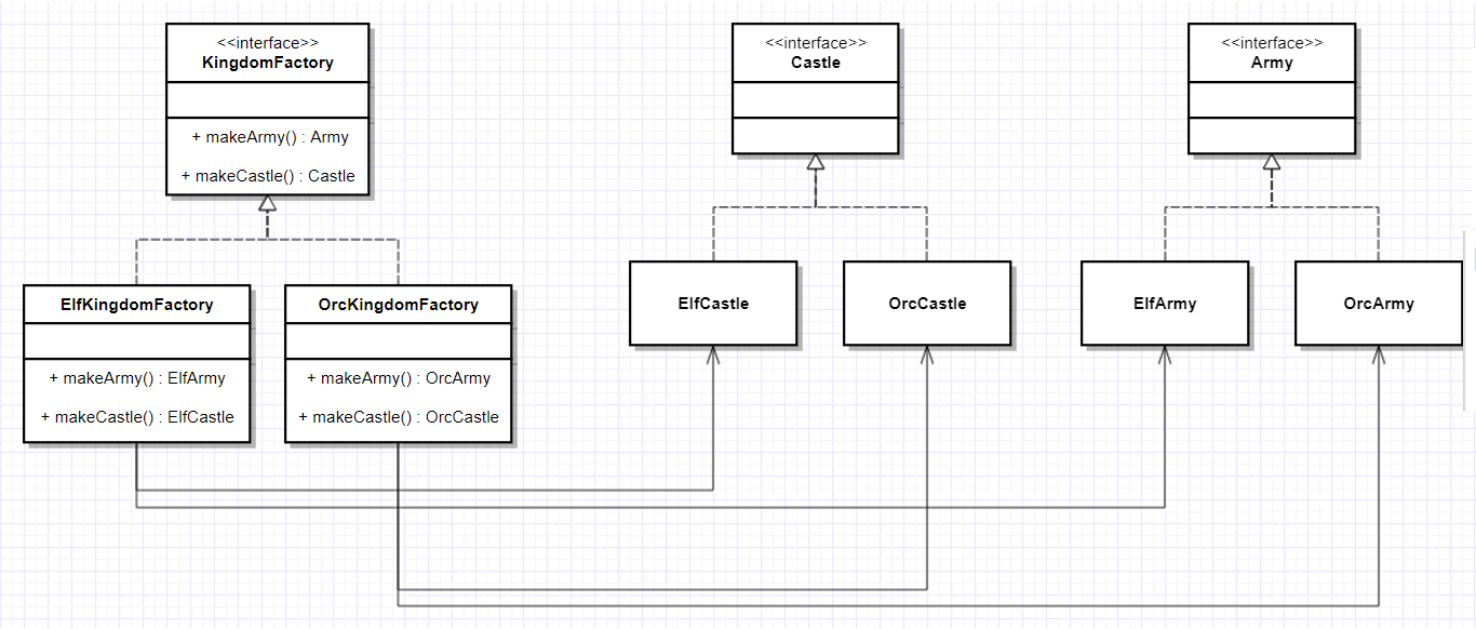
- **Collaborations**
  - Normalement, une seule instance de fabrique **concrète** est créée à l'exécution. Cette fabrique crée les objets avec une implémentation spécifique. Pour créer différents sortes d'objets, les clients doivent utiliser différentes fabriques concrètes.
  - La fabrique abstraite défère la création des objets à ses sous-classes concrètes
- **Conséquences**
  1. Isolation des classes concrètes
  2. Échange facile des familles de produit

Abstract Factory (6)

3. Encouragement de la cohérence entre les produits
4. Prise en compte difficile de nouvelles formes de produit

- **Implémentation**
  - Les fabriques sont souvent des singletons
  - Ce sont les sous-classes concrètes qui font la création, en utilisant le plus souvent une Factory Method
  - Si plusieurs familles sont possibles, la fabrique concrète utilise Prototype
- **Patterns associés**
  - Singleton, Factory Method, Prototype

Abstract Factory (Code Example)




```java
public interface Army {

  String getDescription();
}
```

```java
public interface Castle {

  String getDescription();
}
```

```java
public interface KingdomFactory {

  Castle createCastle();

  King createKing();

  Army createArmy();

}
```

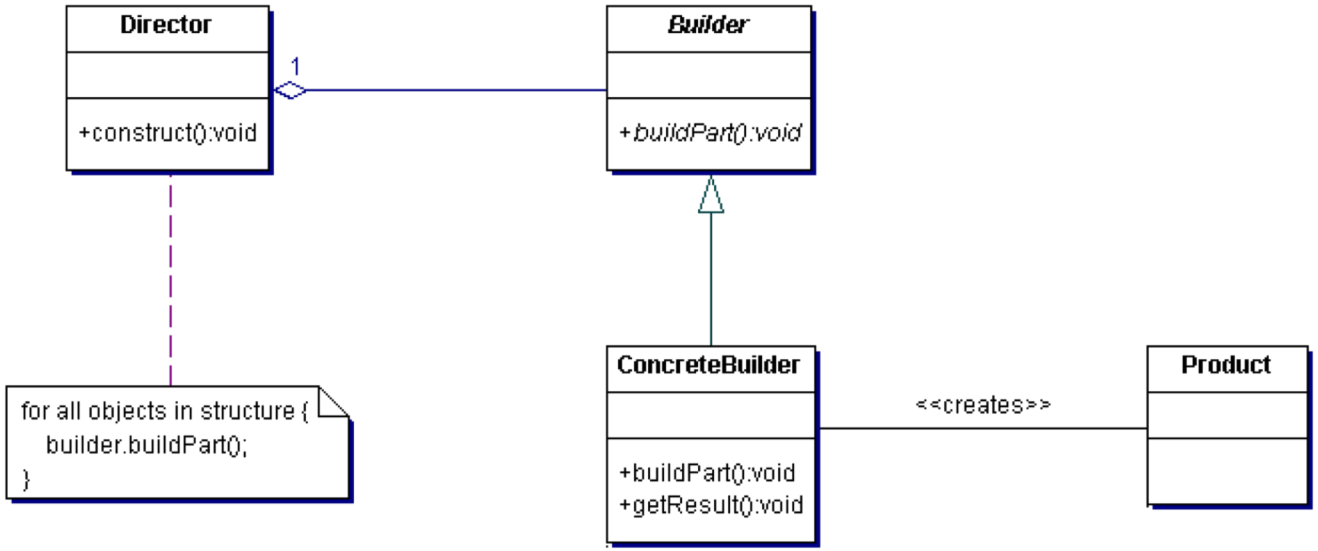
```java
public class ElfKingdomFactory implements KingdomFactory {

  @Override
  public Castle createCastle() { return new ElfCastle(); }

  @Override
  public King createKing() { return new ElfKing(); }

  @Override
  public Army createArmy() { return new ElfArmy(); }

}
```

Builder (Création)

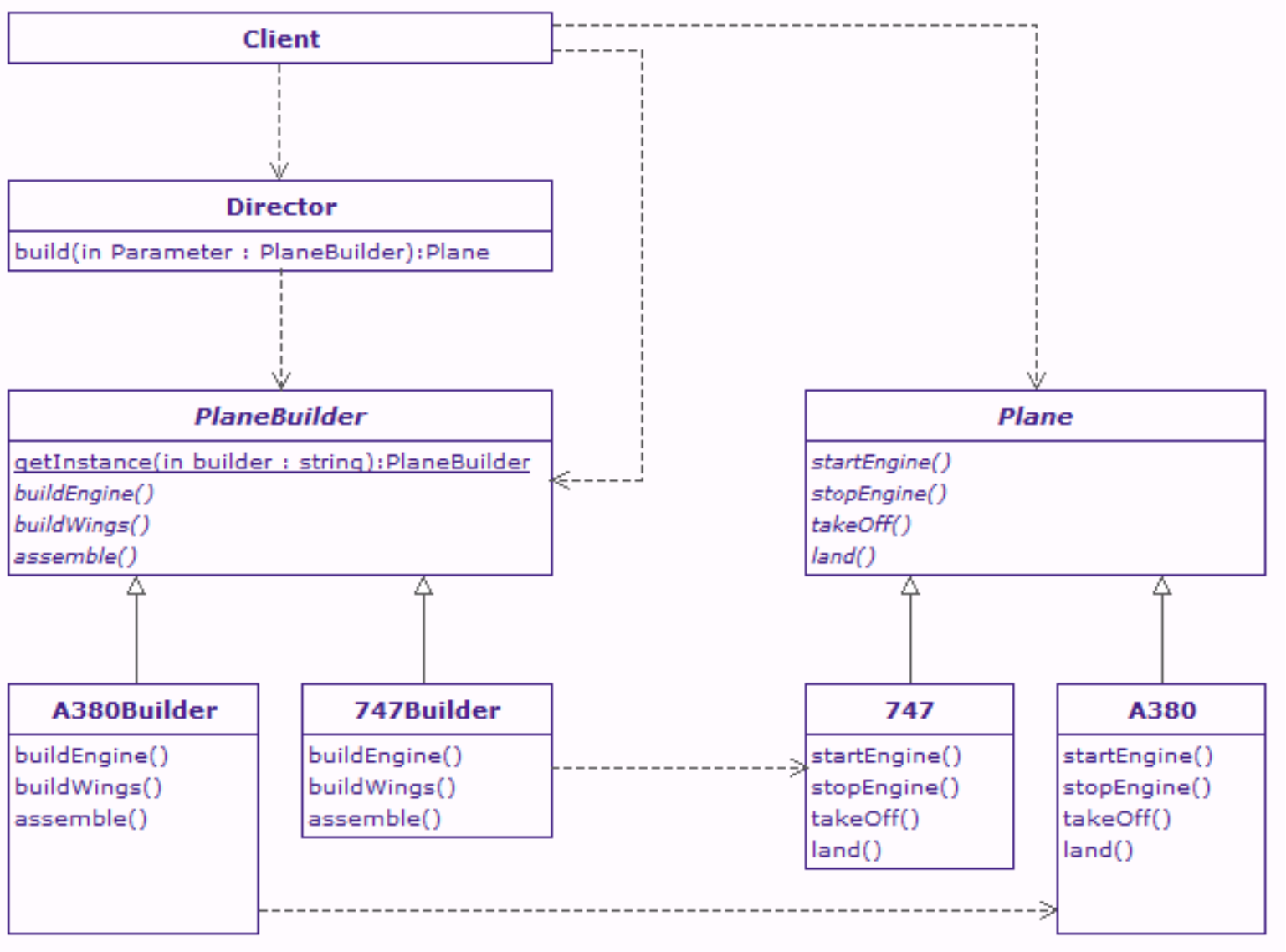
- **Intention**
  - Séparer la construction d'un objet complexe de sa représentation
  - De sorte qu'un même processus de construction puisse créer différentes représentations
- **Synonyme**
  - Constructeur
- **Motivation**
  - Motivation similaire au Pattern Abstract Factory
  - Niveau d'abstraction supplémentaire
    - On passe en paramètre un objet qui sait construire l'objet à partir d'une description
    - On construit alors cet objet de façon séquentielle (pas à pas)

Builder (2)

- **Champs d'application**
  - Quand l'algorithme pour créer un objet complexe doit être indépendant des parties qui le composent et de la façon de les assembler
  - Quand le processus de construction doit permettre différentes représentations pour l'objet qui est construit



Builder (3)

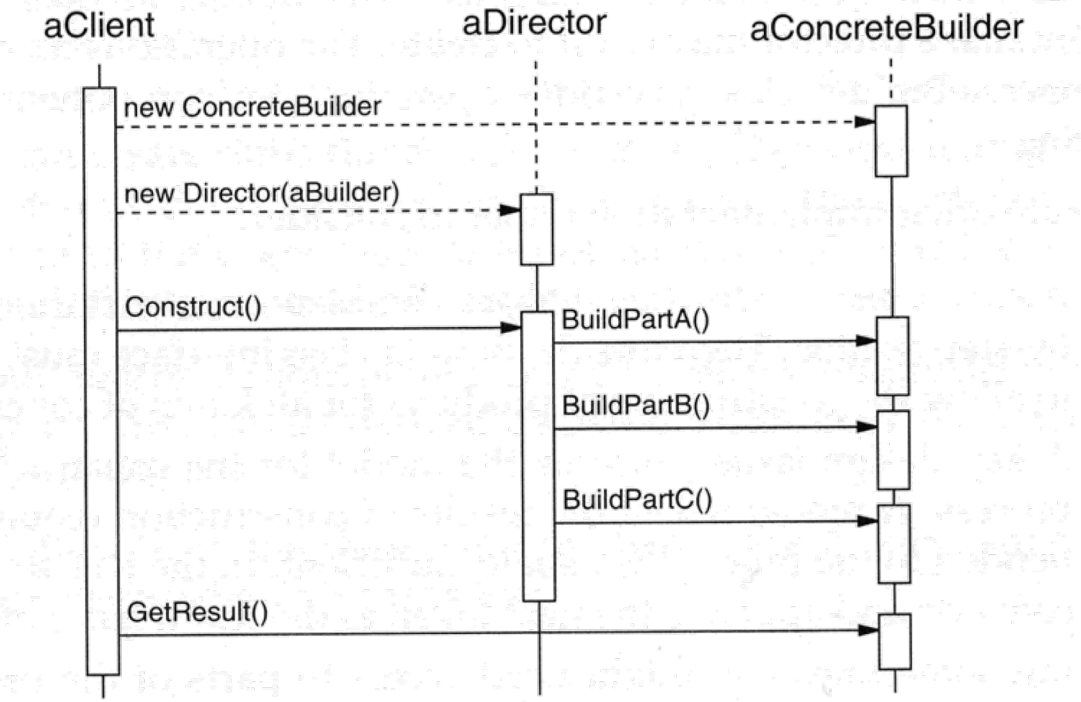

Client
Director
build(in Parameter : PlaneBuilder):Plane
PlaneBuilder
getInstance(in builder : string):PlaneBuilder
buildEngine()
buildWings()
assemble()
Plane
startEngine()
stopEngine()
takeOff()
land()
A380Builder
buildEngine()
buildWings()
assemble()
747Builder
buildEngine()
buildWings()
assemble()
747
startEngine()
stopEngine()
takeOff()
land()
A380
startEngine()
stopEngine()
takeOff()
land()

Builder (4)

- **Participants**
  - Builder
    - Spécifie une interface abstraite pour créer les différentes parties du Produit
  - ConcreteBuilder
    - Construit et assemble les parties du Produit en implémentant l'interface Builder
    - Fournit une interface qui donne accès au Produit
  - Director
    - Construit les objets en utilisant l'interface Builder
  - Product
    - Représente l'objet complexe après la construction

Builder (5)

- Collaboration



Builder (6)

```
public final class Hero {
  private final Profession profession;
  private final String name;
  private final HairType hairType;
  private final HairColor hairColor;
  private final Armor armor;
  private final Weapon weapon;

  private Hero(Builder builder) {
    this.profession = builder.profession;
    this.name = builder.name;
    this.hairColor = builder.hairColor;
    this.hairType = builder.hairType;
    this.weapon = builder.weapon;
    this.armor = builder.armor;
  }
}
```

```
Hero mage = new Hero.Builder(Profession.MAGE, "Riobard").withHairColor(HairColor.BLACK)
                                                        .withWeapon(Weapon.DAGGER).build();
```

```
public static class Builder {
  private final Profession profession;
  private final String name;
  private HairType hairType;
  private HairColor hairColor;
  private Armor armor;
  private Weapon weapon;

  public Builder(Profession profession, String name) {
    if (profession == null || name == null) {
      throw new IllegalArgumentException("profession and name can not be null");
    }
    this.profession = profession;
    this.name = name;
  }

  public Builder withHairType(HairType hairType) {
    this.hairType = hairType;
    return this;
  }

  public Builder withHairColor(HairColor hairColor) {
    this.hairColor = hairColor;
    return this;
  }

  public Builder withArmor(Armor armor) {
    this.armor = armor;
    return this;
  }

  public Builder withWeapon(Weapon weapon) {
    this.weapon = weapon;
    return this;
  }

  public Hero build() {
    return new Hero(this);
  }
}
```

Builder (8)

- **Conséquences**
  - Permet de changer la représentation interne d'un produit
  - Isole la construction de l'objet de sa représentation
  - Permet un contrôle plus fin du procédé de construction
- **Implémentation**
  - Utiliser des noms génériques dans l'interface Builder
  - Builder = classe abstraite
    - Méthodes vides et non abstraites
  - Product ≠ classe abstraite
    - Les produits sont normalement très différents dans leur représentation
- **Patterns associés**
  - Interface, Abstract Factory, Composite

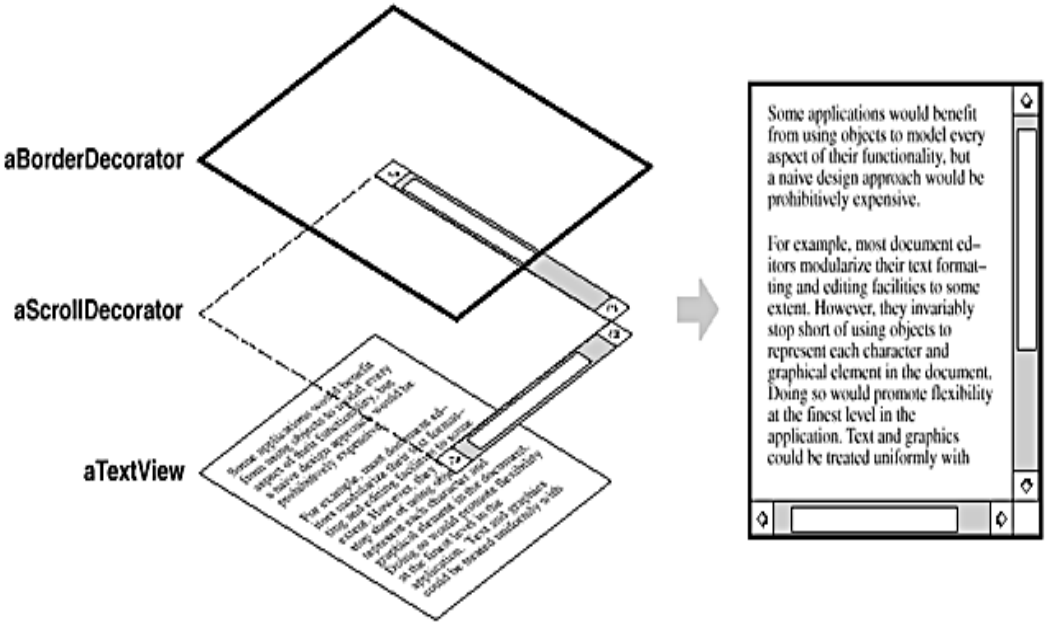
Quelques patrons de structure

Decorator (structure)

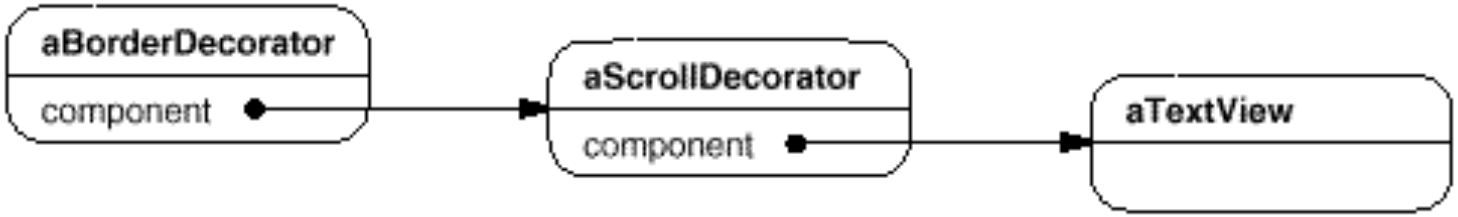
- **Intention**
  - Attacher dynamiquement des capacités additionnelles à un objet
  - Fournir ainsi une alternative flexible à l'héritage pour étendre les fonctionnalités
- **Synonymes :** Wrapper (attention !)
- **Motivation**
  - Ajout de capacités pour objet individuellement et dynamiquement
  - Englober l'objet existant dans un autre objet qui ajoute les capacités (plus que d'hériter)

Fréquence :

Decorator (2)

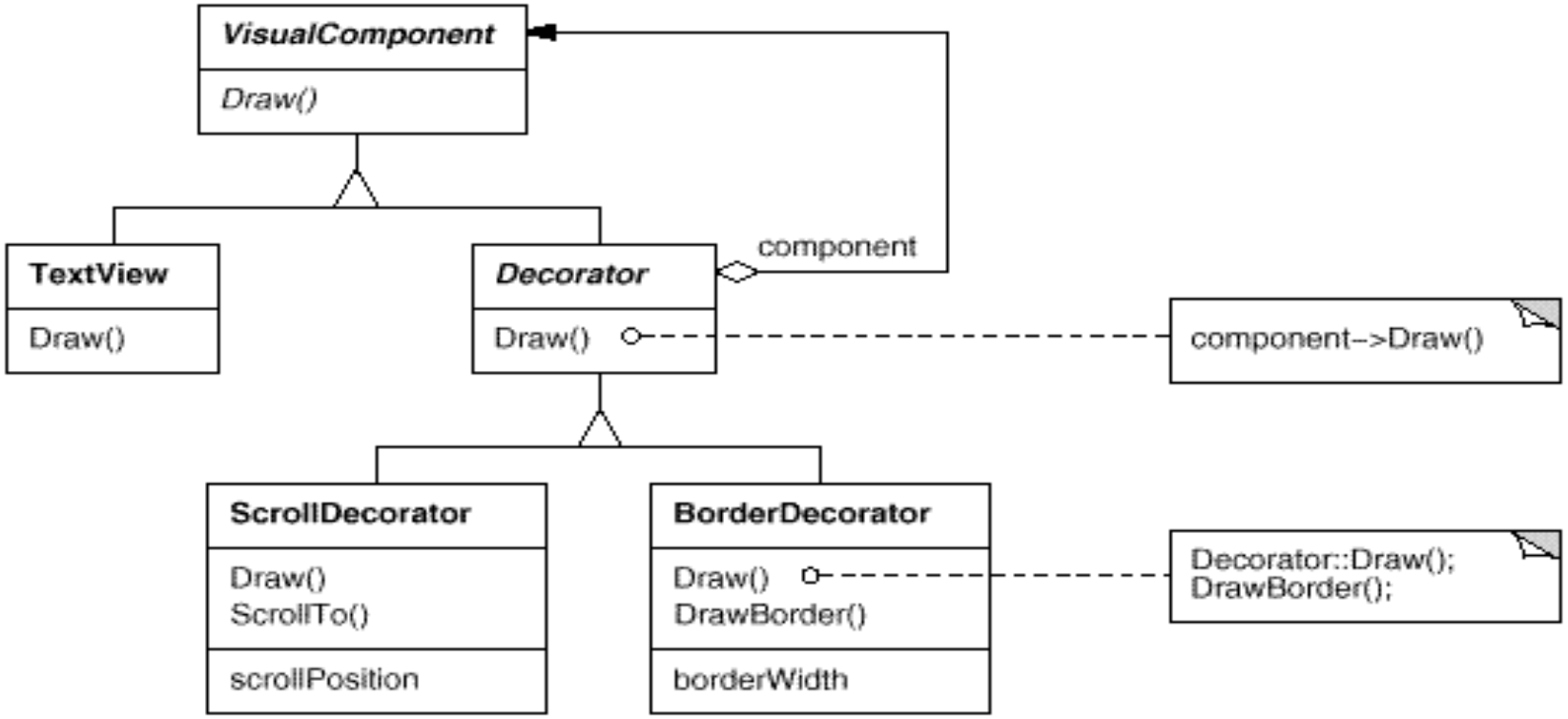


- **Champs d'application**
  - Pour ajouter des capacités de manière transparente
  - Pour des capacités qui peuvent être retirées



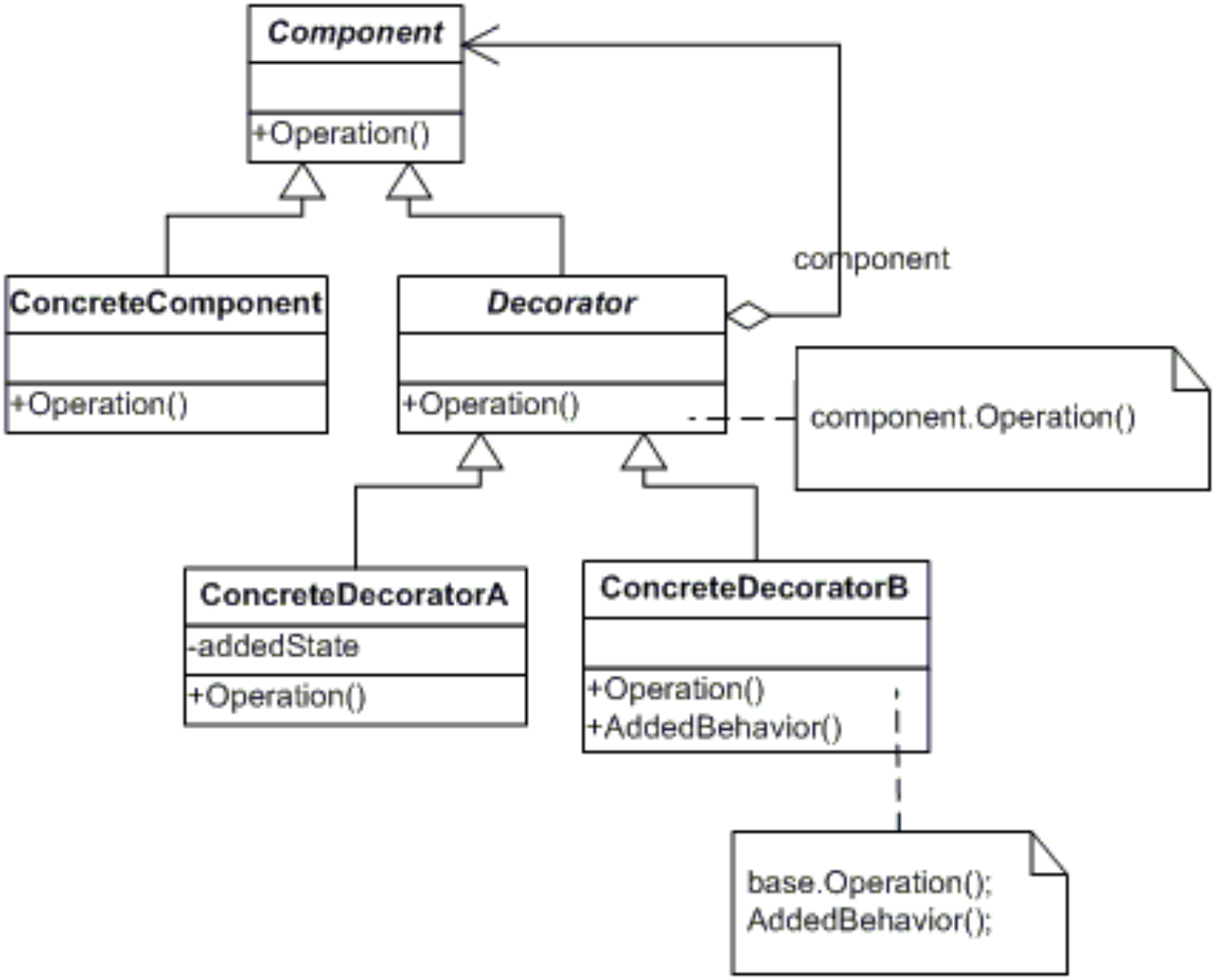
Decorator (3)

- Quand l'extension par héritage produirait un nombre trop important d'extensions indépendantes
- Quand l'héritage est interdit



Decorator (4)

- **Structure**



Decorator (5)

- **Participants**
  - **Component** (VisualComponent) définit l'interface des objets qui peuvent avoir des capacités dynamiques
  - **ConcreteComponent** (TextView) définit l'objet auquel on peut attacher de nouvelles capacités
  - **Decorator** maintient une référence à l'objet Component et définit une interface conforme à ce même Component
  - **ConcreteDecorator** (BorderDecorator, ScrollDecorator) ajoute les capacités à component
- **Collaborations**
  - Decorator transmets les demandes à son objet Component. Il peut aussi effectuer du travail en plus avant et après la transmission

Decorator (6)

- **Conséquences**
  1. Plus de flexibilité que l'héritage
  2. Évite la surcharge inutile de classe en service en haut de la hiérarchie *(PAY AS YOU GO)*
  3. Le décorateur et son composant ne sont pas identiques (identité)
  4. Plein de petits objets
- **Implémentation**
  - Java : utilisation d'interface pour la conformité
  - Pas forcément besoin d'un décorateur abstrait

Decorator (7)

  - Maintenir une classe de base légère
  - Decorator est fait pour le changement d'aspect, Strategy est fait pour le changement radical d'approche

- **Utilisations connues**
  - Organisation de java.io
- **Patterns associés**
  - Adapter, Composite, Strategy

Decorator (Code Example)

```java
public interface Troll {

  void attack();

  int getAttackPower();

  void fleeBattle();

}
```

```java
public class SimpleTroll implements Troll {

  private static final Logger LOGGER = LoggerFactory.getLogger(SimpleTroll.class);

  @Override
  public void attack() { LOGGER.info("The troll tries to grab you!"); }

  @Override
  public int getAttackPower() { return 10; }

  @Override
  public void fleeBattle() { LOGGER.info("The troll shrieks in horror and runs away!"); }
}
```

```java
public static void main(String[] args) {

  // simple troll
  LOGGER.info("A simple looking troll approaches.");
  Troll troll = new SimpleTroll();
  troll.attack();
  troll.fleeBattle();
  LOGGER.info("Simple troll power {}.\n", troll.getAttackPower());

  // change the behavior of the simple troll by adding a decorator
  LOGGER.info("A troll with huge club surprises you.");
  Troll clubbedTroll = new ClubbedTroll(troll);
  clubbedTroll.attack();
  clubbedTroll.fleeBattle();
  LOGGER.info("Clubbed troll power {}.\n", clubbedTroll.getAttackPower());
}
```

Decorator (Code Example 2)

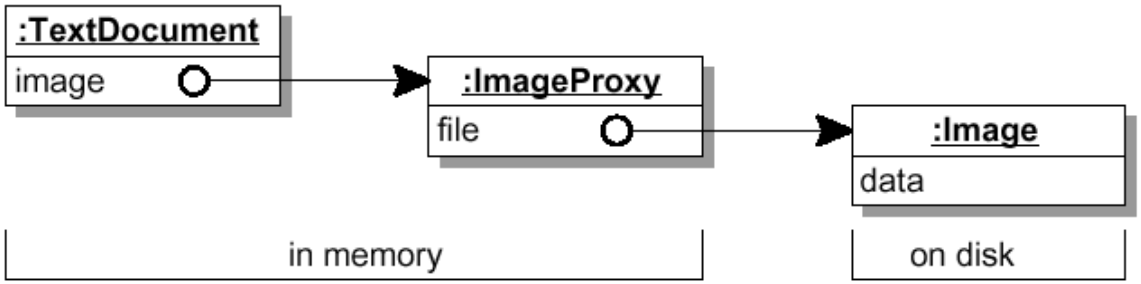
```java
public class ClubbedTroll implements Troll {

  private static final Logger LOGGER = LoggerFactory.getLogger(ClubbedTroll.class);

  private Troll decorated;

  public ClubbedTroll(Troll decorated) { this.decorated = decorated; }

  @Override
  public void attack() {
    decorated.attack();
    LOGGER.info("The troll swings at you with a club!");
  }

  @Override
  public int getAttackPower() { return decorated.getAttackPower() + 10; }

  @Override
  public void fleeBattle() { decorated.fleeBattle(); }
}
```

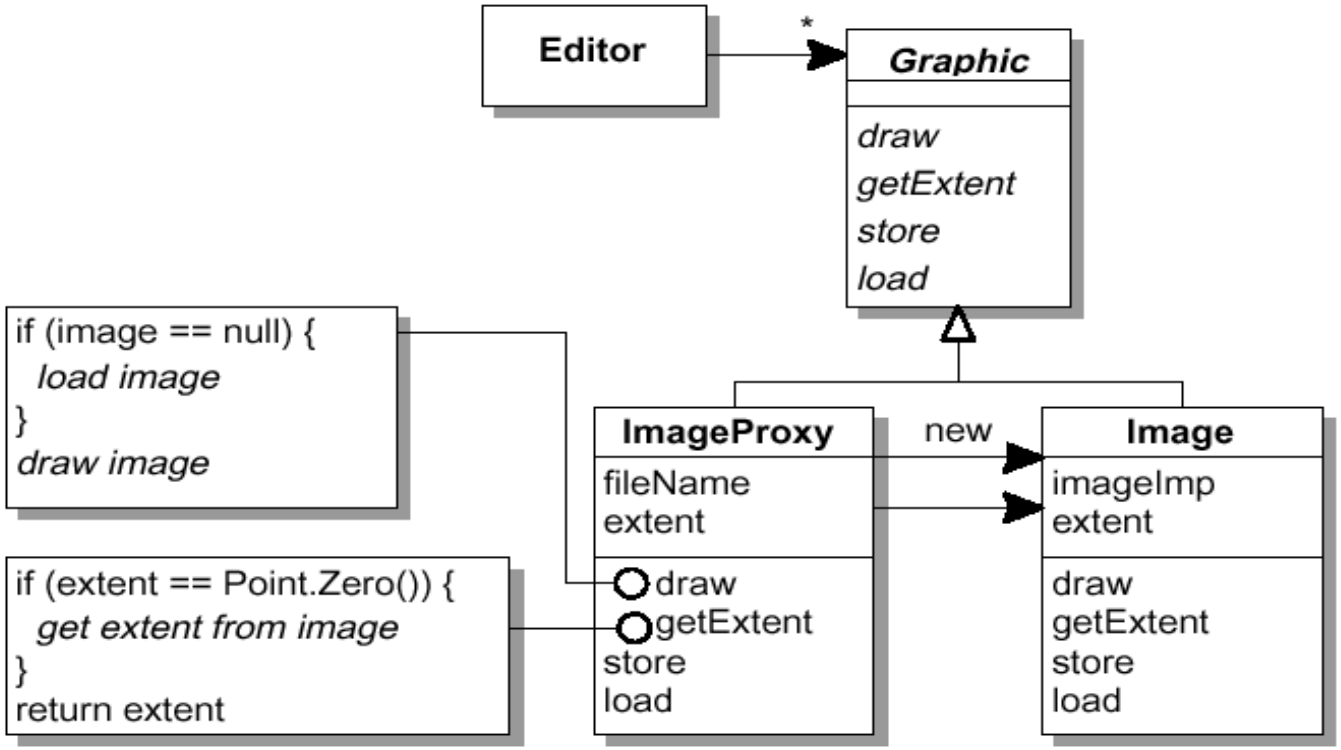
Proxy (structure)

- **Intention**
  - Fournir un substitut afin d'accéder à un autre objet souvent inaccessible.
  - Séparer l 'interface de l'implémentation
- **Synonyme**
  - Procuration, mandat, *surrogate*
- **Motivation**
  - Si un objet, comme une image volumineuse, met beaucoup de temps à se charger
  - Si un objet est situé sur une machine distante
  - Si un objet a des droits d'accès spécifiques

Fréquence :

Proxy (2)


:TextDocument
image
:ImageProxy
file
:Image
data
in memory
on disk

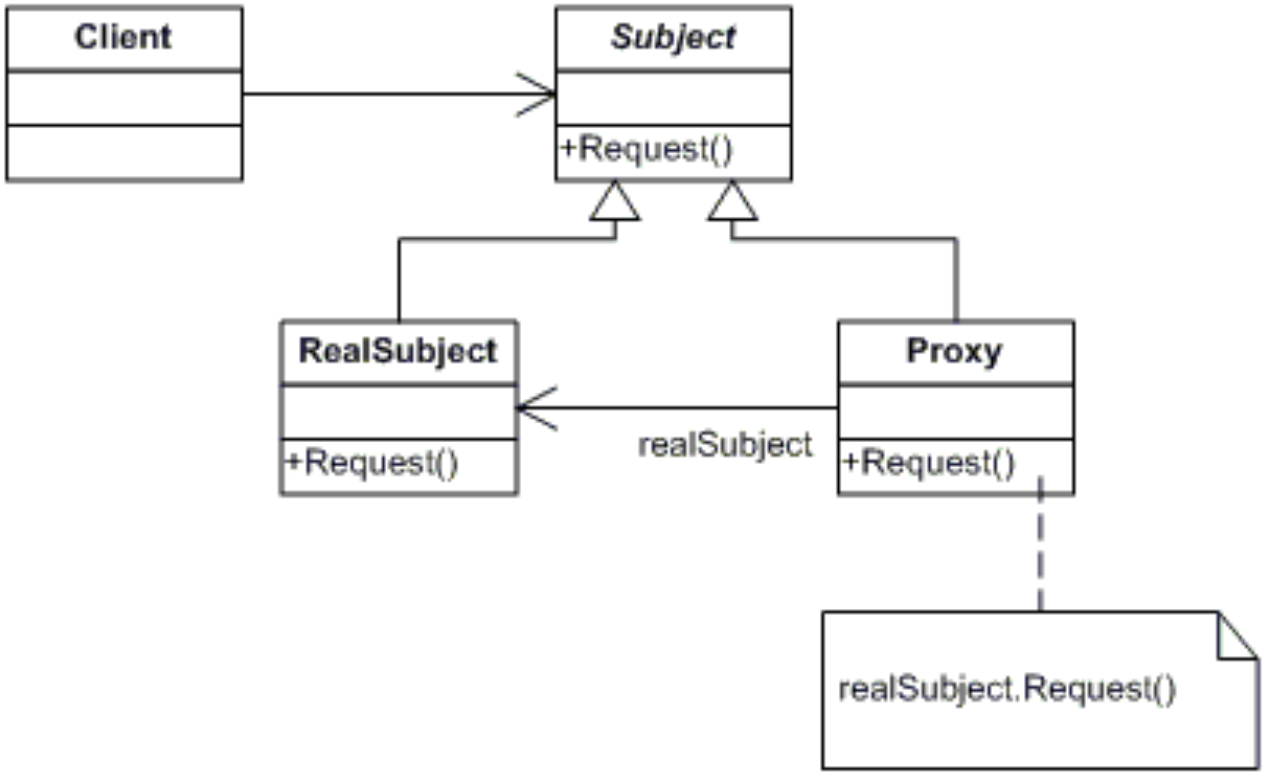

Editor
*
Graphic
draw
getExtent
store
load
if (image == null) {
load image
}
draw image
if (extent == Point.Zero()) {
get extent from image
}
return extent
ImageProxy
fileName
extent
draw
getExtent
store
load
new
Image
imageImp
extent
draw
getExtent
store
load

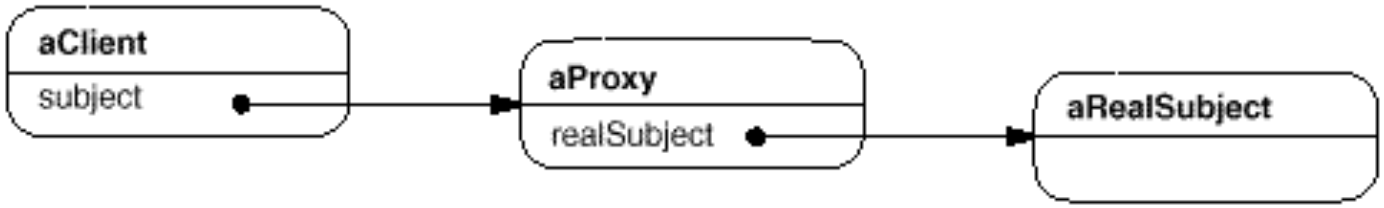
Proxy (3)

- **Champs d'application**
  - Dès qu'il y a un besoin de référencement sophistiqué ou polyvalent, autre qu'un simple pointeur
  - **Remote Proxy** est un représentant d'un objet situé dans un autre espace d'adressage
  - **Virtual Proxy** crée des objets « coûteux » à la demande
  - **Access Proxy** contrôle l'accès à un objet
  - **SmartReference** effectue un travail supplémentaire lors de l'accès à l'objet
    - Comptage de références (*smart pointers)*
    - Chargement d'objets persistants
    - Vérification de non verrouillage

Proxy (4)

- **Structure**



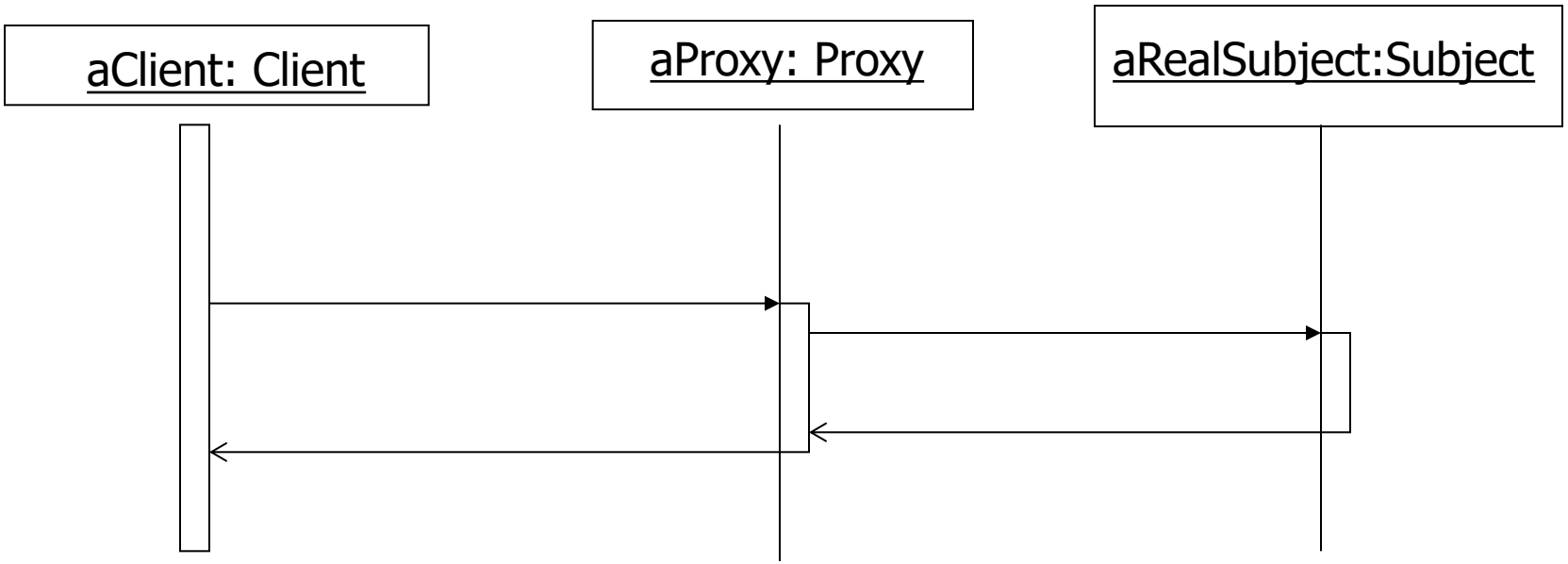


Proxy (5)

- **Participants**
  - **Proxy** maintient une référence qui permet au Proxy d'accéder au *RealSubject*. Il fournit une interface identique à celle de *Subject*, pour pouvoir se substituer au *RealSubject*. Il contrôle les accès au RealSubject
  - ***Subject*** définit une interface commune pour *RealSubject* et Proxy. Proxy peut ainsi être utilisé partout où le RealSubject devrait être utilisé
  - ***RealSubject*** définit l'objet réel que le Proxy représente

Proxy (6)

- **Collaborations**
  - Le Proxy *retransmet les requêtes au RealSubject lorsque c'est nécessaire, selon la forme du Proxy*



Proxy (7)

- **Conséquences**
  - L'inaccessibilité du sujet est transparente
  - Comme le proxy a la même interface que son sujet, il peuvent être librement interchangeable
  - le proxy n'est pas un objet réel mais simplement la réplique exacte de son sujet
- **Implémentation**
  - Possibilité de nombreuses optimisations…

Proxy (8)

- **Utilisations connues**
  - Utilisation courante (systématique…) pour faire apparaître un objet comme étant local dans les applications distribuées (CORBA, Java RMI)
- **Patterns associés**
  - Adapter, Decorator
  - Access Proxy, Remote Proxy, Virtual Proxy, SmartReference

Proxy (Code Example)

```
public static void main(String[] args) {

  WizardTowerProxy proxy = new WizardTowerProxy(new IvoryTower());
  proxy.enter(new Wizard( name: "Red wizard"));
  proxy.enter(new Wizard( name: "White wizard"));
  proxy.enter(new Wizard( name: "Black wizard"));
  proxy.enter(new Wizard( name: "Green wizard"));
  proxy.enter(new Wizard( name: "Brown wizard"));

}
```

```
public interface WizardTower {

  void enter(Wizard wizard);
}

public class IvoryTower implements WizardTower {

  private static final Logger LOGGER = LoggerFactory.getLogger(IvoryTower.class);

  public void enter(Wizard wizard) { LOGGER.info("{} enters the tower.", wizard); }

}
```

Proxy (Code Example 2)

```java
public class WizardTowerProxy implements WizardTower {

  private static final Logger LOGGER = LoggerFactory.getLogger(WizardTowerProxy.class);

  private static final int NUM_WIZARDS_ALLOWED = 3;

  private int numWizards;

  private final WizardTower tower;

  public WizardTowerProxy(WizardTower tower) { this.tower = tower; }

  @Override
  public void enter(Wizard wizard) {
    if (numWizards < NUM_WIZARDS_ALLOWED) {
      tower.enter(wizard);
      numWizards++;
    } else {
      LOGGER.info("{} is not allowed to enter!", wizard);
    }
  }
}
```

Quelques patrons de comportement

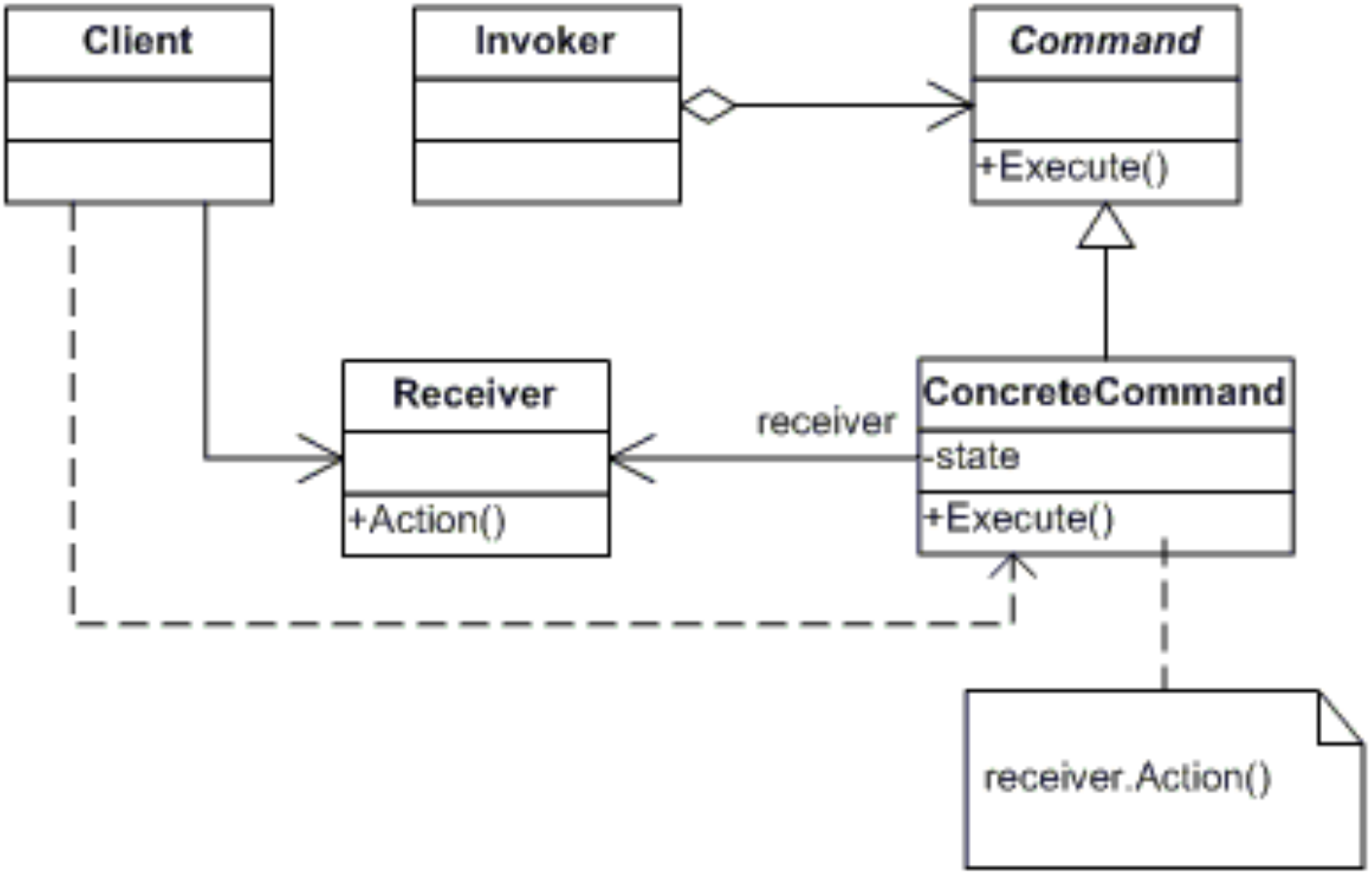
Command (comportement)

- **Intentions**
  - Encapsuler une requête comme un objet
  - Permettre de *défaire* des traitements (undo)
- **Synonymes**
  - Action, Transaction
- **Motivations**
  - Découpler les objets qui invoquent une action de ceux qui l'exécutent
  - Réaliser un traitement sans avoir besoin de savoir de quoi il s'agit et de qui va l'effectuer

Fréquence :

Command (2)

- **Structure**

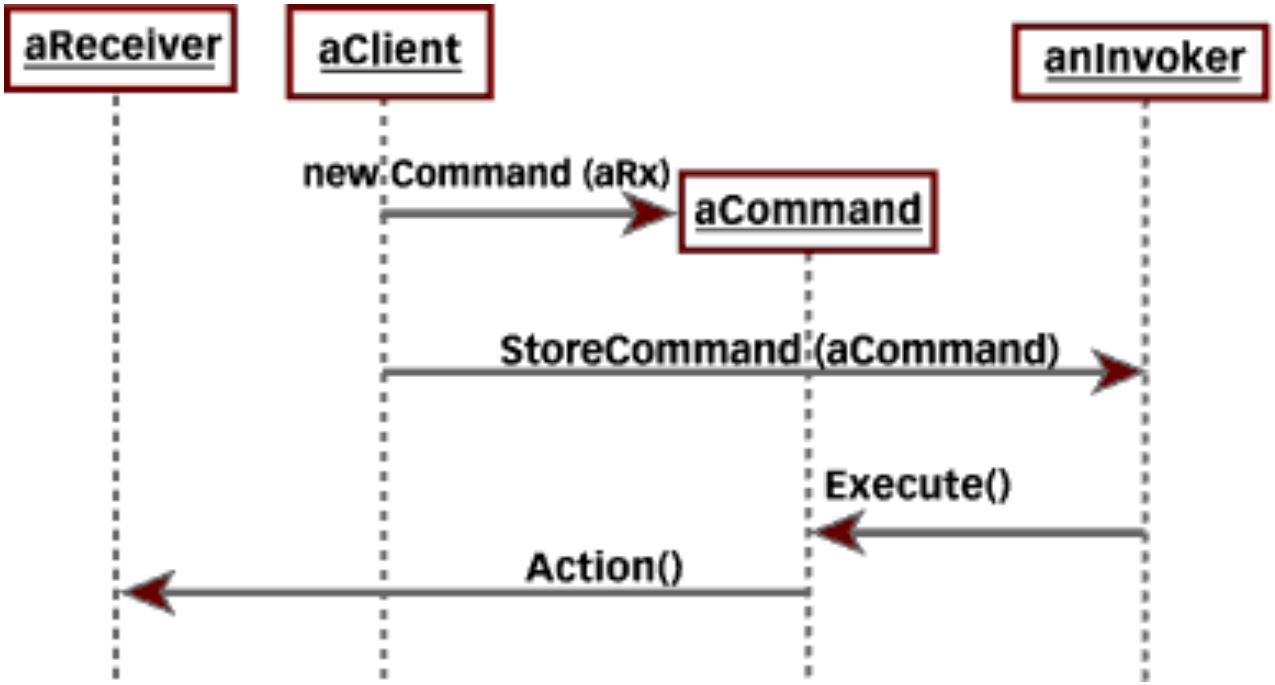


Command (3)

- **Participants**
  - **Command** déclare une interface pour exécuter une opération
  - **ConcreteCommand** définit une astreinte entre l'objet Receiver et l'action concrétise execute() en invoquant la méthode correspondante de l'objet Receiver
  - **Client** crée l'objet ConcreteCommand et positionne son Receiver
  - **Invoker** demande à l'objet Command d'entreprendre la requête
  - **Receiver** sait comment effectuer la ou les opérations associées à la demande

Command (4)

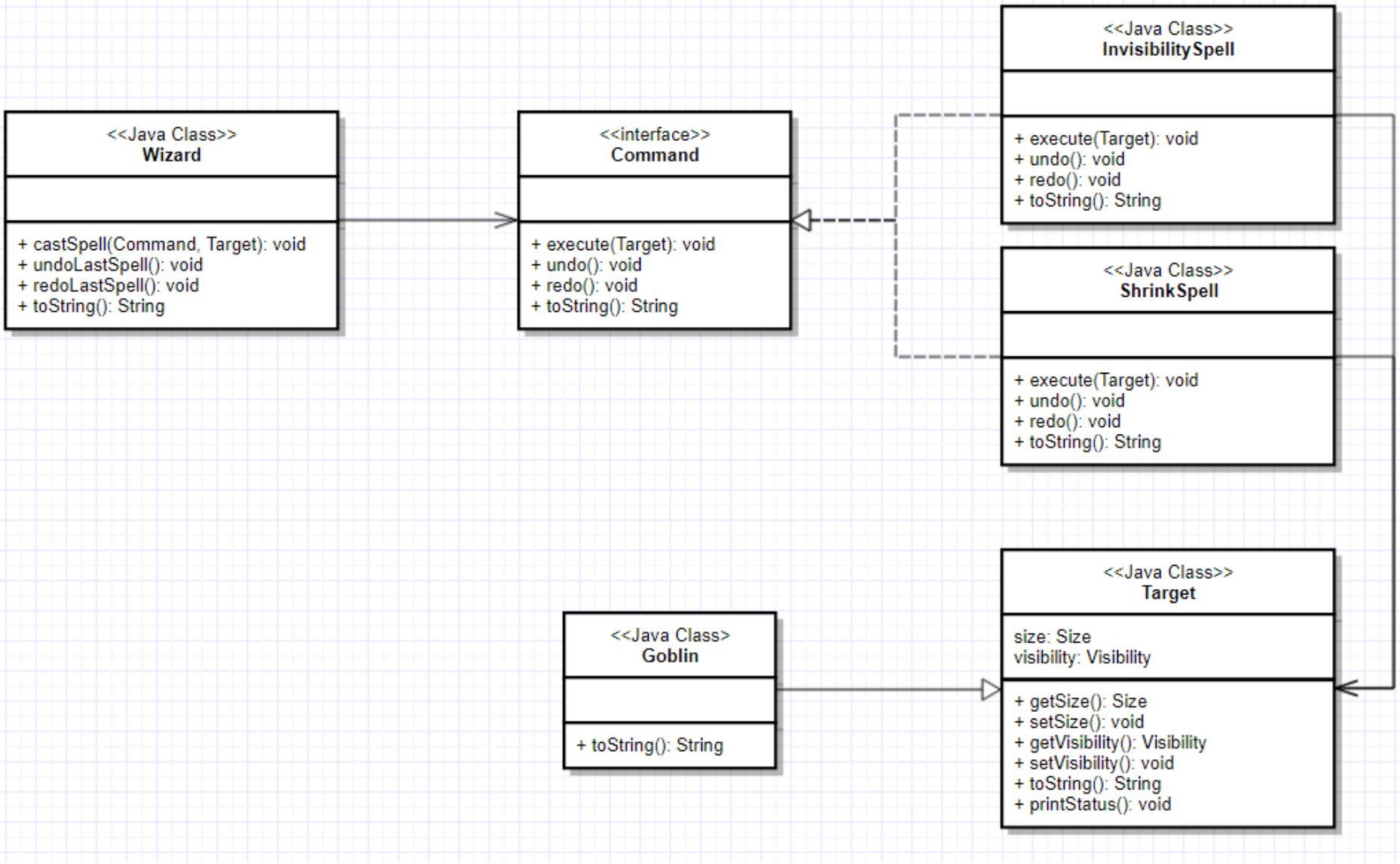
- **Champs d'application**
  - Défaire des requêtes (undo)
  - Paramétrer les objets par des actions
  - Manipuler les requêtes, les exécuter à différents moments
- **Collaboration**



Command (5)

- **Conséquences**
  - Découplage entre invocation et réalisation
  - Commandes manipulées comme n'importe quel autre objet
  - Assemblage composite => MacroCommande
  - Facilité d'ajouts de nouvelles commandes
- **Implémentation**
  - Pour supporter les undo et redo, il faut définir les opérations correspondantes et la classe ConcreteCommand doit stocker, pour cela, certaines informations
  - Un historique des commandes doit être conservé pour réaliser les undo à plusieurs niveaux
- **Patterns associés**
  - Composite, Memento, Prototype

Command (Code Example)



Command (Code Example 2)

```java
public class Wizard {

  private static final Logger LOGGER = LoggerFactory.getLogger(Wizard.class);

  private Deque<Command> undoStack = new LinkedList<>();
  private Deque<Command> redoStack = new LinkedList<>();

  public void castSpell(Command command, Target target) {
    LOGGER.info("{} casts {} at {}", this, command, target);
    command.execute(target);
    undoStack.offerLast(command);
  }

  public void undoLastSpell() {
    if (!undoStack.isEmpty()) {
      Command previousSpell = undoStack.pollLast();
      redoStack.offerLast(previousSpell);
      LOGGER.info("{} undoes {}", this, previousSpell);
      previousSpell.undo();
    }
  }
}
```

```java
public abstract class Command {

  public abstract void execute(Target target);

  public abstract void undo();

  public abstract void redo();

  @Override
  public abstract String toString();

}
```

Command (Code Example 3)

```
public abstract class Target {

  private static final Logger LOGGER = LoggerFactory.getLogger(Target.class);

  private Size size;

  private Visibility visibility;

  public Size getSize() { return size; }

  public void setSize(Size size) { this.size = size; }

  public Visibility getVisibility() { return visibility; }

  public void setVisibility(Visibility visibility) { this.visibility = visibility; }


public class Goblin extends Target {

  public Goblin() {
    setSize(Size.NORMAL);
    setVisibility(Visibility.VISIBLE);
  }
```

Command (Code Example 4)

```java
public class InvisibilitySpell extends Command {

  private Target target;

  @Override
  public void execute(Target target) {
    target.setVisibility(Visibility.INVISIBLE);
    this.target = target;
  }

  @Override
  public void undo() {
    if (target != null) {
      target.setVisibility(Visibility.VISIBLE);
    }
  }

  @Override
  public void redo() {
    if (target != null) {
      target.setVisibility(Visibility.INVISIBLE);
    }
  }
```

```java
public class ShrinkSpell extends Command {

  private Size oldSize;
  private Target target;

  @Override
  public void execute(Target target) {
    oldSize = target.getSize();
    target.setSize(Size.SMALL);
    this.target = target;
  }

  @Override
  public void undo() {
    if (oldSize != null && target != null) {
      Size temp = target.getSize();
      target.setSize(oldSize);
      oldSize = temp;
    }
  }

  @Override
  public void redo() { undo(); }
```

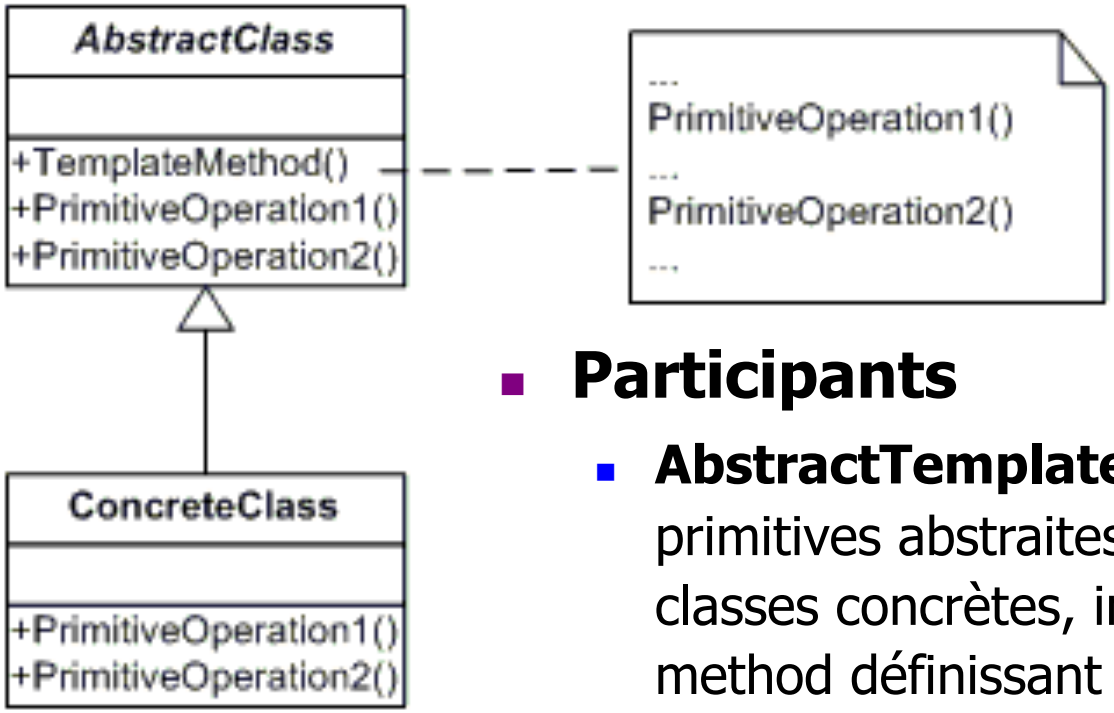
Template Method (comportement)

- **Intention**
  - Définir le squelette d'un algorithme dans une opération, et laisser les sous-classes définir certaines étapes
  - Permet de redéfinir des parties de l'algorithme sans avoir à modifier celui-ci
- **Motivation**
  - Faire varier le comportement de chaque opération de façon indépendante dans les sous-classes
- **Champs d'application**
  - Implanter une partie invariante d'un algorithme
  - Partager des comportements communs d'une hiérarchie de classes

Fréquence :

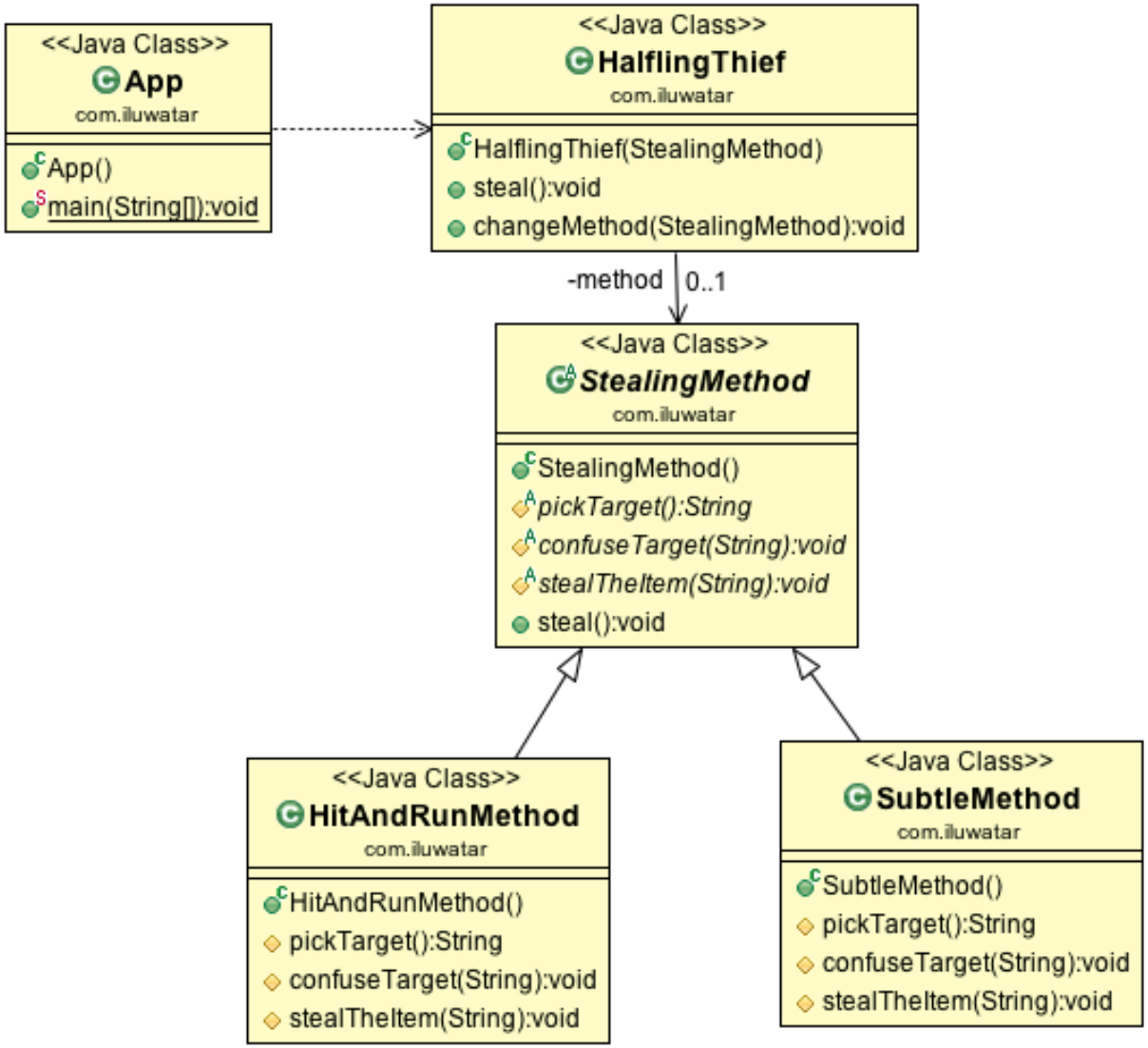
Template Method (2)

- **Structure**



- **Participants**
  - **AbstractTemplate** définit les opérations primitives abstraites implémentées par les sous-classes concrètes, implémente un template method définissant la trame d'un algorithme
  - **ConcreteTemplate** implémente les opérations primitives pour exécuter les étapes spécifiques de l'algorithme

Template Method (2)



https://github.com/iluwatar/java-design-patterns/tree/master/template-method

Template Method (4)

- **Conséquences**
  - Réutilisation du code
  - Structure de contrôle inversé
  - Permet d'imposer des règles de surcharge
  - Mais il faut sous-classer pour spécialiser le comportement
- **Implémentation**
  - Réduction du nombre des opérations primitives
  - Convention de nomenclature (préfix do- ou faire-)
- **Patterns associés**
  - Factory Method, Strategy

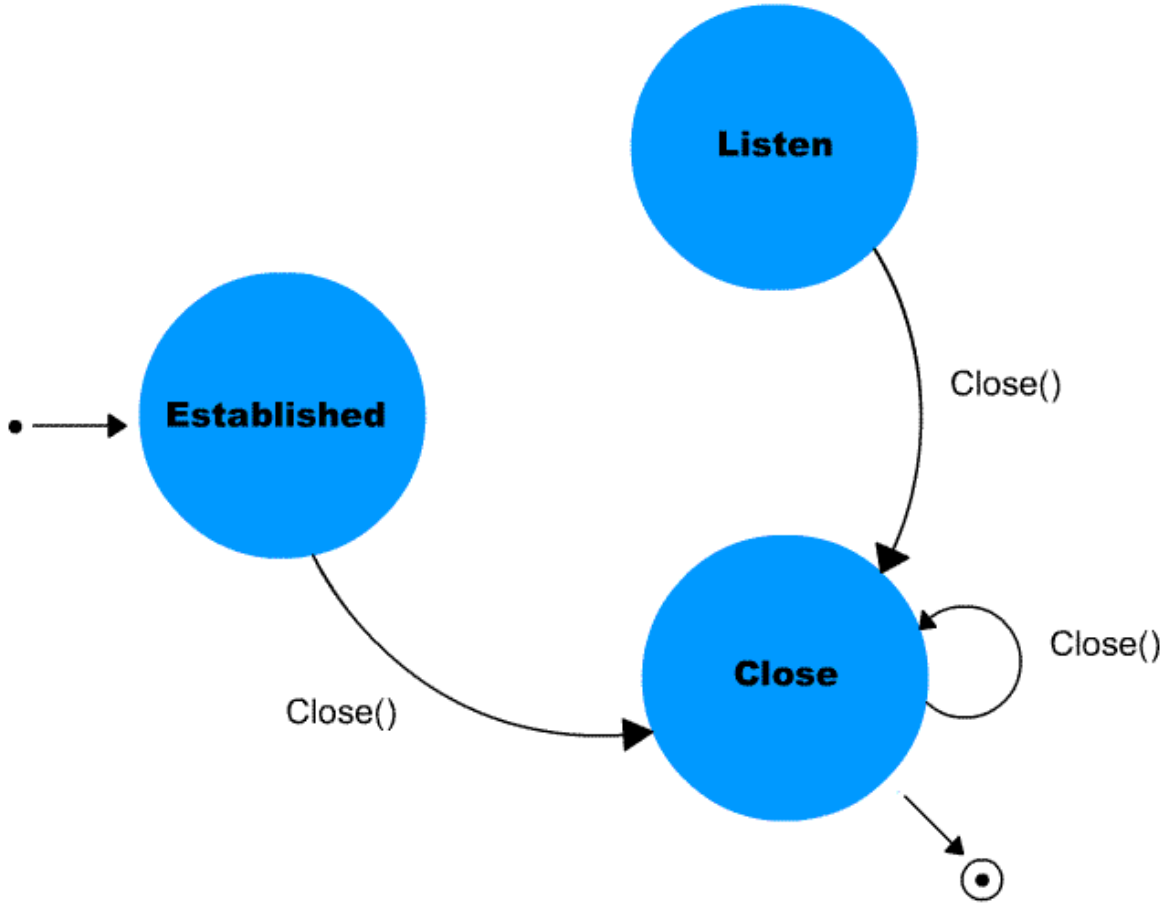
State (comportement)

- **Intention**
  - Modifier le comportement d'un objet quand son état interne change
  - Obtenir des traitements en fonction de l'état courant
  - Tout est mis en place pour donner l'impression que l'objet lui-même a été modifié
- **Motivation**
  - Eviter les instructions conditionnelles de grande taille (if then else)
  - Simplifier l'ajout et la suppression d'un état et le comportement qui lui est associé
- **Champs d'application**
  - Implanter une partie invariante d'un algorithme
  - Partager des comportements communs d'une hiérarchie de classes

Fréquence :

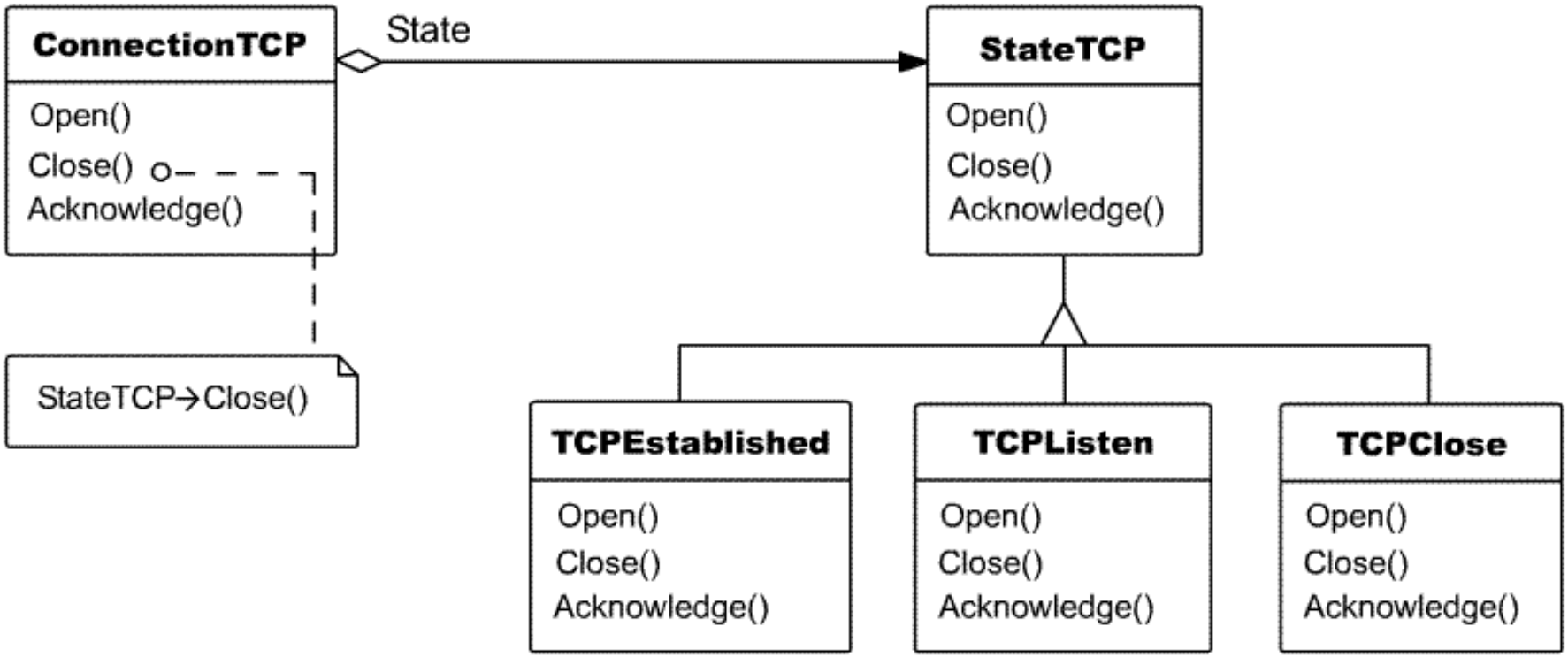
State (2)

- **Exemple : connexion TCP**



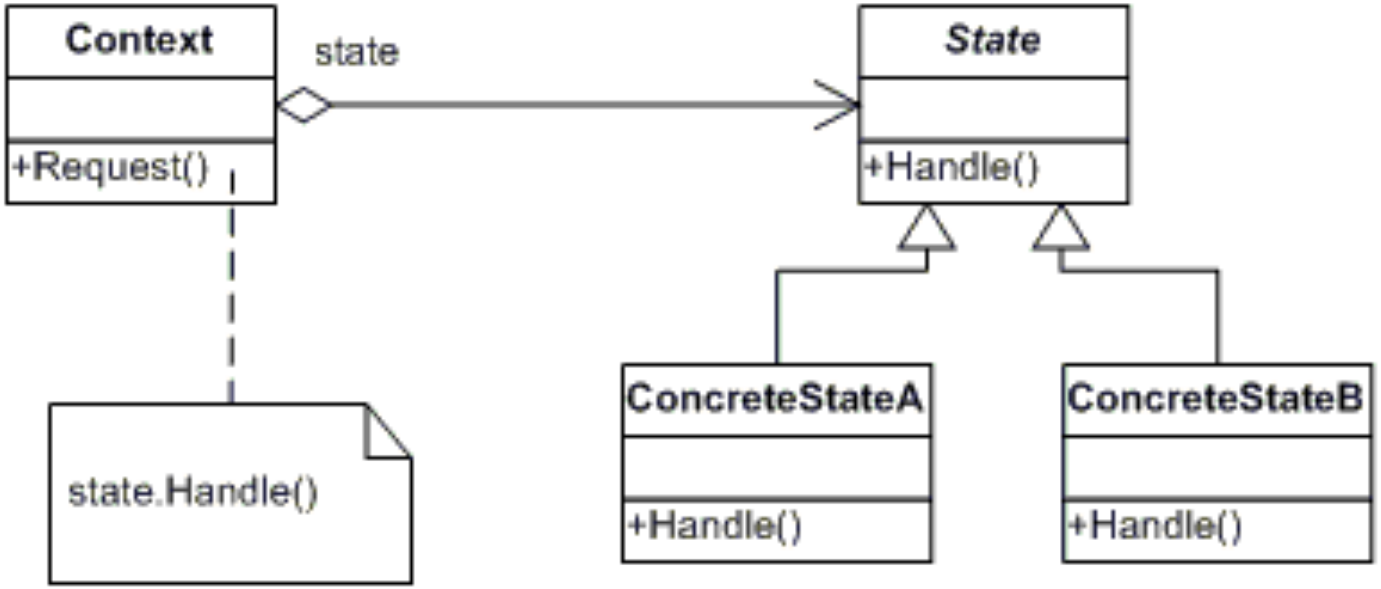
State (3)

- **Exemple : modélisation**



State (4)

- **Structure**



  - Créer des classes d'états qui implémentent une interface commune
  - Déléguer des opérations dépendantes des états de l'objet *Contexte* à son objet état actuel
  - Veiller à ce que l'objet *Contexte* pointe sur un objet état qui reflète son état actuel

State (5)

- **Participants**
  - **Context** (ConnectionTCP) est une classe qui permet d'utiliser un objet à états et qui gère une instance d'un objet ConcreteState
  - **State** (StateTCP) définit une interface qui encapsule le comportement associé avec un état particulier de Context
  - **ConcretState** (EstablishedTCP, ListenTCP, CloseTCP) implémente un comportement associé avec l'état de Context)
- **Collaborations**
  - Il revient soit à Context, soit aux ConcreteState de décider de l'état qui succède à un autre état
- **Conséquences**
  - Séparation des comportements relatifs à chaque état
  - transitions plus explicites

State (Code Example)

```java
public interface State {

  void onEnterState();

  void observe();

}
```

```java
public class PeacefulState implements State {

  private static final Logger LOGGER = LoggerFactory.getLogger(PeacefulState.class);

  private Mammoth mammoth;

  public PeacefulState(Mammoth mammoth) { this.mammoth = mammoth; }

  @Override
  public void observe() { LOGGER.info("{} is calm and peaceful.", mammoth); }

  @Override
  public void onEnterState() { LOGGER.info("{} calms down.", mammoth); }

}
public class AngryState implements State {

  private static final Logger LOGGER = LoggerFactory.getLogger(AngryState.class);

  private Mammoth mammoth;

  public AngryState(Mammoth mammoth) { this.mammoth = mammoth; }

  @Override
  public void observe() { LOGGER.info("{} is furious!", mammoth); }

  @Override
  public void onEnterState() { LOGGER.info("{} gets angry!", mammoth); }
```

State (Code Example 2)

```java
public class Mammoth {

  private State state;

  public Mammoth() { state = new PeacefulState( mammoth: this); }

  /**
   * Makes time pass for the mammoth
   */
  public void timePasses() {
    if (state.getClass().equals(PeacefulState.class)) {
      changeStateTo(new AngryState( mammoth: this));
    } else {
      changeStateTo(new PeacefulState( mammoth: this));
    }
  }

  private void changeStateTo(State newState) {
    this.state = newState;
    this.state.onEnterState();
  }

  @Override
  public String toString() { return "The mammoth"; }

  public void observe() { this.state.observe(); }
```

```java
public static void main(String[] args) {

  Mammoth mammoth = new Mammoth();
  mammoth.observe();
  mammoth.timePasses();
  mammoth.observe();
  mammoth.timePasses();
  mammoth.observe();
```

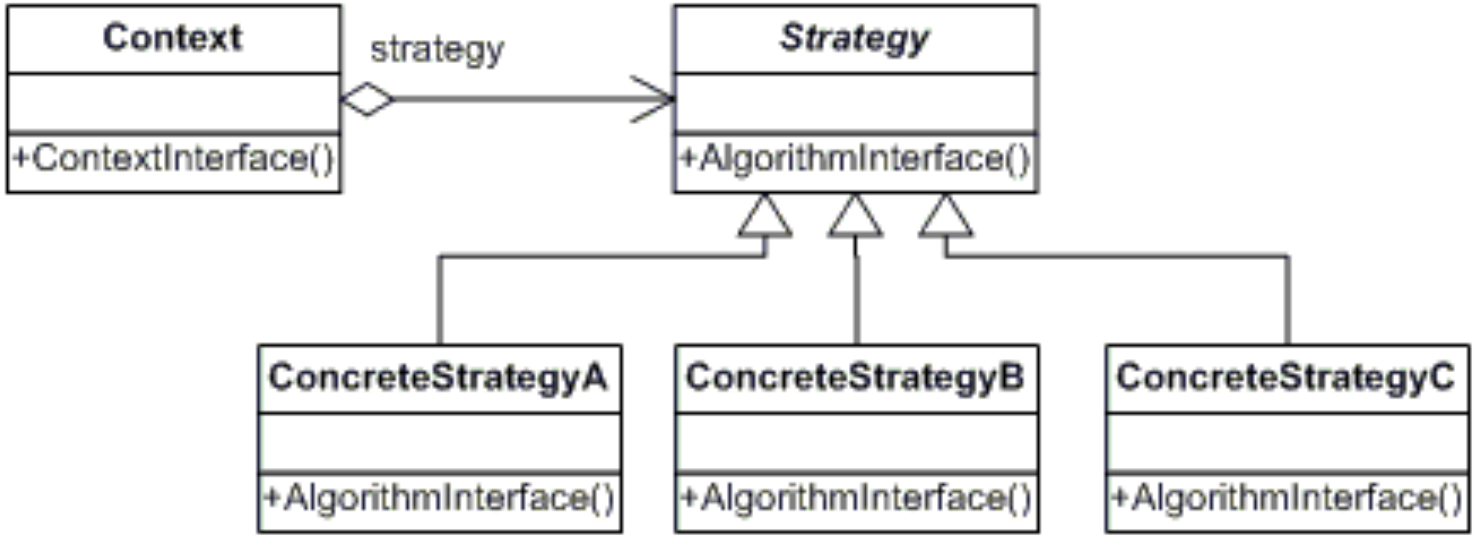
Strategy (comportement)

- **Intention**
  - Définir une hiérarchie de classes pour une famille d'algorithmes, encapsuler chacun d'eux, et les rendre interchangeables.
  - Les algorithmes varient indépendamment des clients qui les utilisent
- **Synonyme**
  - Policy
- **Motivation**
  - Cas des classes ayant le même comportement et utilisant des variantes d'un même algorithme :
    - Encapsuler ces algorithmes dans une classe (la stratégie)
    - Implémenter les "stratégies" dans les classes héritées
  - Evite les répétitions de code

Fréquence :

Strategy (2)

- **Champs d'application**
  - Lorsque de nombreuses classes associées ne diffèrent que par leur comportement
  - Lorsqu'on a besoin de plusieurs variantes d'un algorithme
  - Lorsqu'un algorithme utilise des données que les clients ne doivent pas connaître
  - Lorsqu'une classe définit plusieurs comportements
- **Structure**



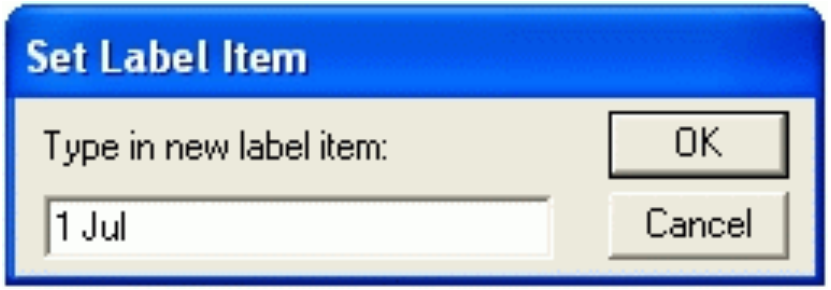
Strategy (3)

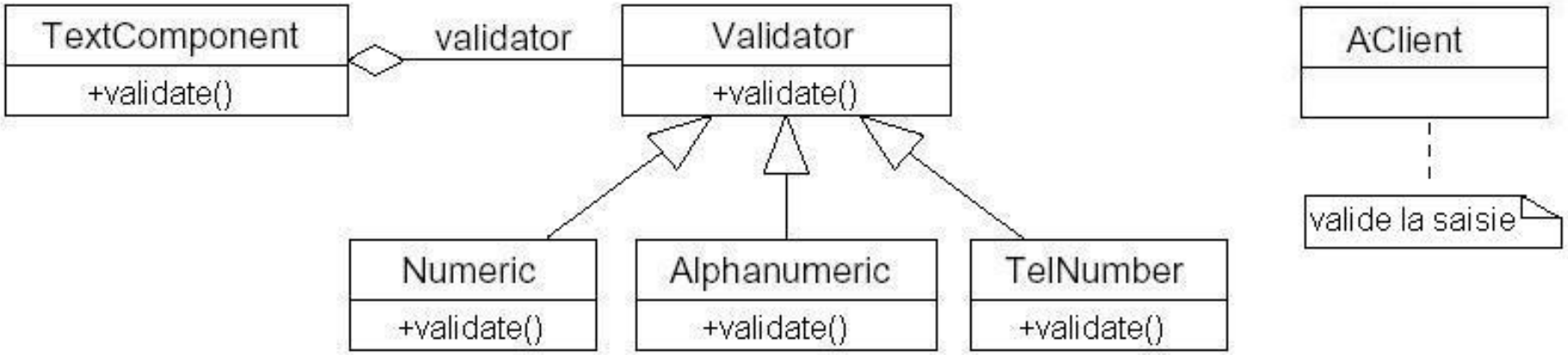
- **Participants**
  - **Context**
    - maintient une référence à l'objet Strategy
    - peut définir un interface qui permet à l 'objet Strategy d'accéder à ses données
  - **Strategy** déclare une interface commune à tous les algorithmes
  - **ConcreteStrategy** implémente l'algorithme en utilisant l'interface Strategy
- **Collaborations**
  - Le Context envoie les requêtes de ses clients à l'une de ses stratégies.
  - Les clients créent la classe concrète qui implémente l'algorithme.
  - Puis ils passent la classe au Context.
  - Enfin ils interagissent avec le Context exclusivement.

Strategy (4)

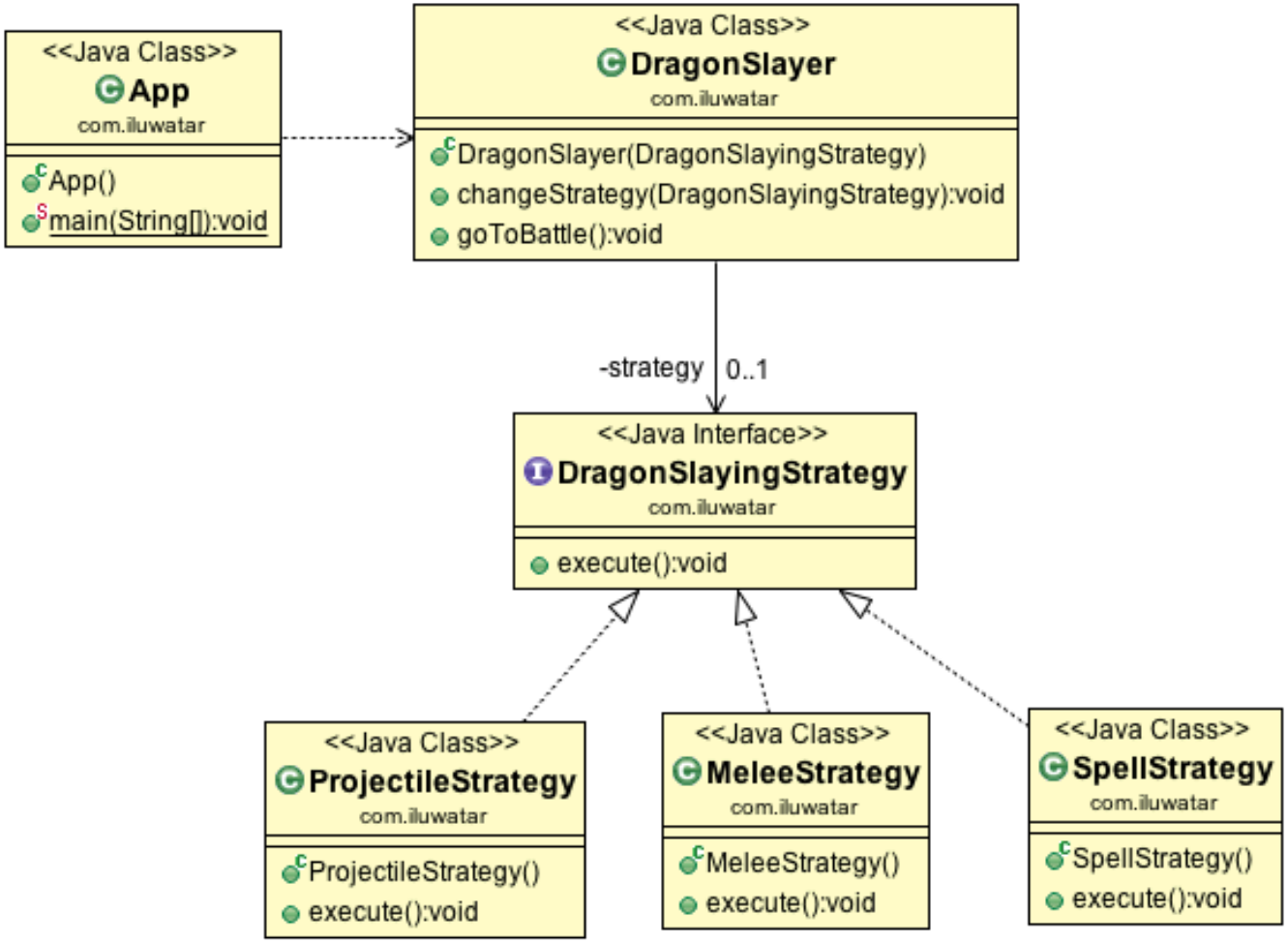
- ***Exemple :*** Validation des données dans un dialogue
  - Plusieurs stratégies de validation selon le type de données : numériques, alphanumériques,…

```
switch(dataType){
case NUMERIC : { //… }
case ALPHANUMERIC : { //… }
case TEL_NUMBER : { //…}
}
```





Strategy (5)



https://github.com/iluwatar/java-design-patterns/tree/master/strategy

Strategy (Code Example - 6)

```java
public class DragonSlayer {

  private DragonSlayingStrategy strategy;

  public DragonSlayer(DragonSlayingStrategy strategy) { this.strategy = strategy; }

  public void changeStrategy(DragonSlayingStrategy strategy) { this.strategy = strategy; }

  public void goToBattle() { strategy.execute(); }
}

@FunctionalInterface
public interface DragonSlayingStrategy {

  void execute();

}
```

```java
public class SpellStrategy implements DragonSlayingStrategy {

  private static final Logger LOGGER = LoggerFactory.getLogger(SpellStrategy.class);

  @Override
  public void execute() {
    LOGGER.info("You cast the spell of disintegration and the dragon vaporizes in a pile of dust!");
  }

}
```

Strategy (7)

- **Conséquences**

+	-
+ Simplification du code client + Élimination des boucles conditionnelles + Extension des algorithmes + Les clients n'ont pas besoin d'avoir accès au code des classes concrètes + Les familles d'algorithmes peuvent être rangées hiérarchiquement ou rassemblées dans une super-classe commune	- Le Context ne doit pas changer - Les clients doivent connaître les différentes stratégies - Le nombre d'objets augmente beaucoup - Imaginons une classe Strategy avec beaucoup de méthodes. Chaque sous-classe connaît ses méthodes mais peut être qu'elle ne les utilisera pas

Références

Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1994

Descriptions:

- **http://www.dofactory.com/**
- **http://sourcemaking.com/design_patterns**
- **http://www.oodesign.com**
- **http://www.fluffycat.com/Java-Design-Patterns**

Codes Java des patterns :

- **https://www.tutorialspoint.com/design_pattern/**
- **https://github.com/iluwatar/java-design-patterns**