

# CHAPITRE 4: PATRONS DU GANG OF FOUR

# Gang des 4

2

- Le premier livre sur les patrons de conceptions était :  
‘*Design Patterns, Elements of reusable objected-oriented softwares*’ par Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides’
- Connus comme le nom Gang des 4 (‘Gang of Four’)
- Ils ont répertorié 23 patrons de conception fondamentaux regroupés en **3 catégories**

# Classification du GoF

3

- **Patrons de création (Creational Patterns)**
  - ▣ Concernent l'abstraction du processus d'instanciation (création d'objets)
  - ▣ Aident à créer des objets, au lieu d'avoir à instancier les objets directement.
    - Indépendance entre la manière de créer et la manière d'utiliser
- **Patrons de structure (Structural Patterns)**
  - ▣ Concernent la composition de classes et d'objets.
  - ▣ Aident à composer des groupes d'objets en des structures plus larges, telles que des interfaces utilisateur complexes.
    - Indépendance entre les objets et les connexions
- **Patrons de comportement (Behavioural Patterns)**
  - ▣ Concernent l'interaction des classes et des objets : les algorithmes et la répartition des responsabilités entre les objets
  - ▣ Aident à définir la communication entre les objets du système et définir comment le flux est contrôlé.
    - Encapsulation de processus

# Classification du GoF

4

Patrons de création (5)	Patrons de structure (7)	Patrons de comportement (11)
<b>Factory Method</b> <b>Abstract Factory</b> Builder Prototype <b>Singleton</b>	<b>Adapter</b> Bridge <b>Composite</b> <b>Decorator</b> <b>Facade</b> Flyweight <b>Proxy</b>	Interpreter <b>Template Method</b> Chain of Responsibility <b>Command</b> <b>Iterator</b> Mediator Memento <b>Observer</b> <b>State</b> <b>Strategy</b> Visitor

L'usage montre que certains patterns sont plus utilisés que d'autres.  
Selon le livre DP-Tête la Première les 14 patterns les plus populaires sont ceux mis en gras sur le tableau.

# Éléments essentiels des patrons

5

- Nom du patron
- Problème (Applicabilité)
  - ▣ Quel est le problème et le contexte pour lesquels nous utiliserions ce patron?
  - ▣ Quelles sont les conditions pour l'utilisation de ce patron?
- Solution (Structure)
  - ▣ Une description des éléments constituant le patron de conception
- Conséquences
  - ▣ Le pour et le contre de l'utilisation du patron
  - ▣ Les impacts sur la réutilisabilité, la portabilité et l'extensibilité

# PATRONS DE CRÉATION



# Patrons de Création

7

- Abstraction du processus d'instanciation.
- Rendre le système indépendant du mode de création des objets qui le compose.
  - ▣ Encapsulation de la connaissance des classes concrètes à utiliser
  - ▣ Cacher la manière dont les instances sont créées et combinées
- Deux types
  - ▣ Patron de création de classe : utilisation de l'héritage pour faire varier la classe instanciée (ex. Factory)
  - ▣ Patron de création d'objets : délégation de la construction à un autre objet (ex. AbstractFactory, Builder)

# PATRONS DE CRÉATION

Factory Method (Fabrication/Usine)

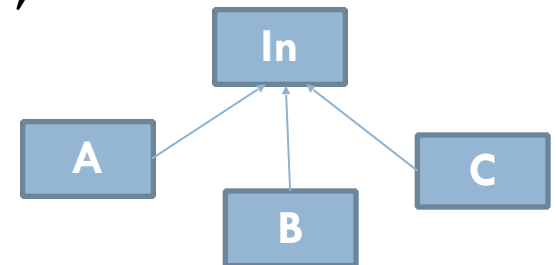


# Instanciación au moment de l'exécution

9

- Quand vous utilisez `new`, vous instanciez une classe concrète :
  - ▣ Il s'agit d'une implémentation, non d'une interface.
  - ▣ Lier le code à une classe concrète peut le rendre plus fragile et plus rigide.
- Quand vous avez tout un ensemble de classes concrètes apparentées, vous êtes souvent obligé d'écrire du code qui ressemble au fragment suivant:

```
In o;  
if (cond1) { o = new A();  
} else if (cond2) { o = new B();  
} else if (con3) { o = new C();  
}
```



# Instanciación au moment de l'exécution

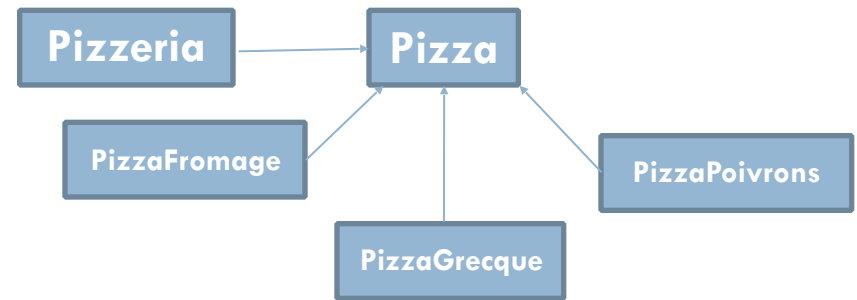
10

- Plusieurs instanciaciones de classes concrètes et la décision de la classe à instancier est prise au moment de l'exécution, en fonction d'un certain nombre de conditions.
- Lorsqu'il faudra apporter des modifications ou des extensions, il faut reprendre ce code et examiner ce qu'il faudra ajouter/supprimer.
- Ce type de code se retrouve souvent dans plusieurs parties de l'application
  - ▣ maintenance et mises à jour plus difficiles et plus sujettes à l'erreur
  - ▣ code n'ai pas « fermé à la modification »

# Identifier ce qui varie

11

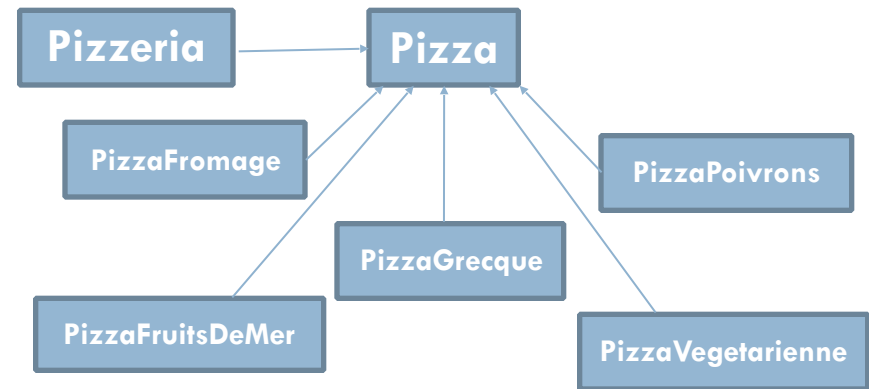
```
public class Pizzeria {  
    ...  
    Pizza commanderPizza(String type) {  
        Pizza pizza;  
        if (type.equals("fromage")) {  
            pizza = new PizzaFromage();  
        } else if (type.equals("grecque")) {  
            pizza = new PizzaGrecque();  
        } else if (type.equals("poivrons")) {  
            pizza = new PizzaPoivrons();  
        }  
  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
    ...  
}
```



# Identifier ce qui varie

11

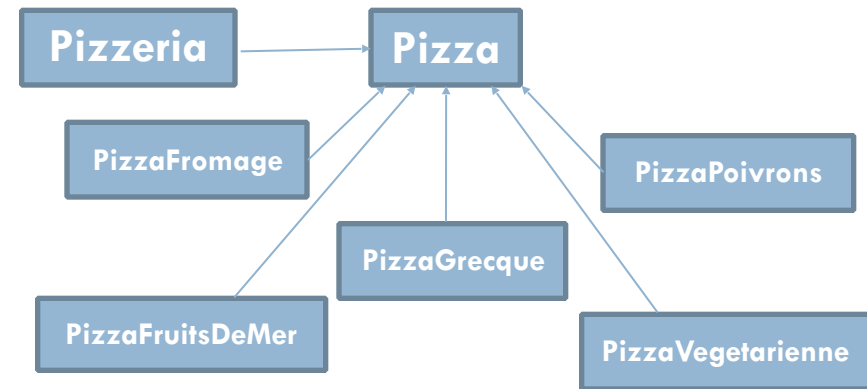
```
public class Pizzeria {  
    ...  
    Pizza commanderPizza(String type) {  
        Pizza pizza;  
        if (type.equals("fromage")) {  
            pizza = new PizzaFromage();  
        } else if (type.equals("grecque")) {  
            pizza = new PizzaGrecque();  
        } else if (type.equals("poivrons")) {  
            pizza = new PizzaPoivrons();  
        }  
  
        else if (type.equals("fruitsDeMer")) {  
            pizza = new PizzaFruitsDeMer();  
        } else if (type.equals("vegetarienne"))  
        {  
            pizza = new PizzaVegetarienne();  
        }  
  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
    ...  
}
```



# Identifier ce qui varie

11

```
public class Pizzeria {  
    ...  
    Pizza commanderPizza(String type) {  
        Pizza pizza;  
        if (type.equals("fromage")) {  
            pizza = new PizzaFromage();  
        } else if (type.equals("grecque")) {  
            pizza = new PizzaGrecque();  
        } else if (type.equals("poivrons")) {  
            pizza = new PizzaPoivrons();  
        }  
  
        else if (type.equals("fruitsDeMer")) {  
            pizza = new PizzaFruitsDeMer();  
        } else if (type.equals("vegetarienne"))  
        {  
            pizza = new PizzaVegetarienne();  
        }  
  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
    ...  
}
```



Voici ce qui varie. Comme le type de pizza change avec le temps, vous n'allez pas cesser de modifier ce code

# Extraire ce qui varie..

12

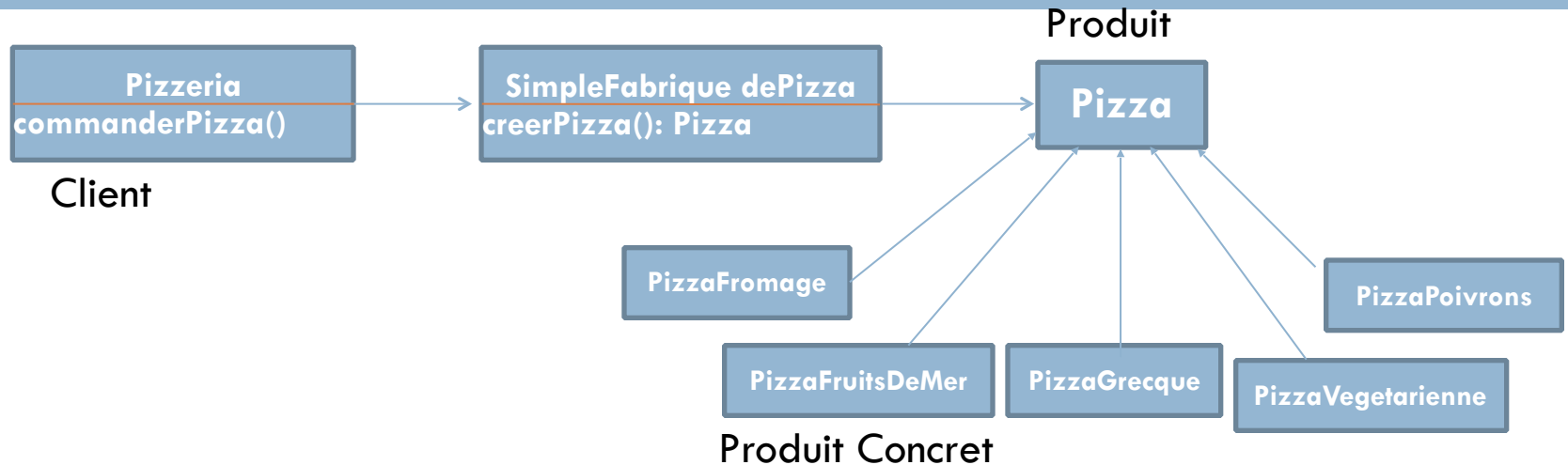
- Tout d'abord, nous extrayons le code qui crée les objets de la méthode `commanderPizza()` et le mettre dans la méthode `creerPizza()`

```
public class Pizzeria {  
    ...  
    Pizza creerPizza(String type) {...}  
    ..  
    Pizza commanderPizza(String type) {  
        Pizza pizza;  
        pizza=creerPizza(type)  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
    ...  
}
```

- Une technique courante est de rendre cette méthode statique (fabrique statique) car on n'a pas besoin d'instancier un objet pour utiliser la méthode de création, mais cela ne résout pas le problème
  - **Inconvénient** : vous ne pouvez pas sous-classer ni modifier le comportement de la méthode de création (méthode de classe)

# Encapsuler la création des objets

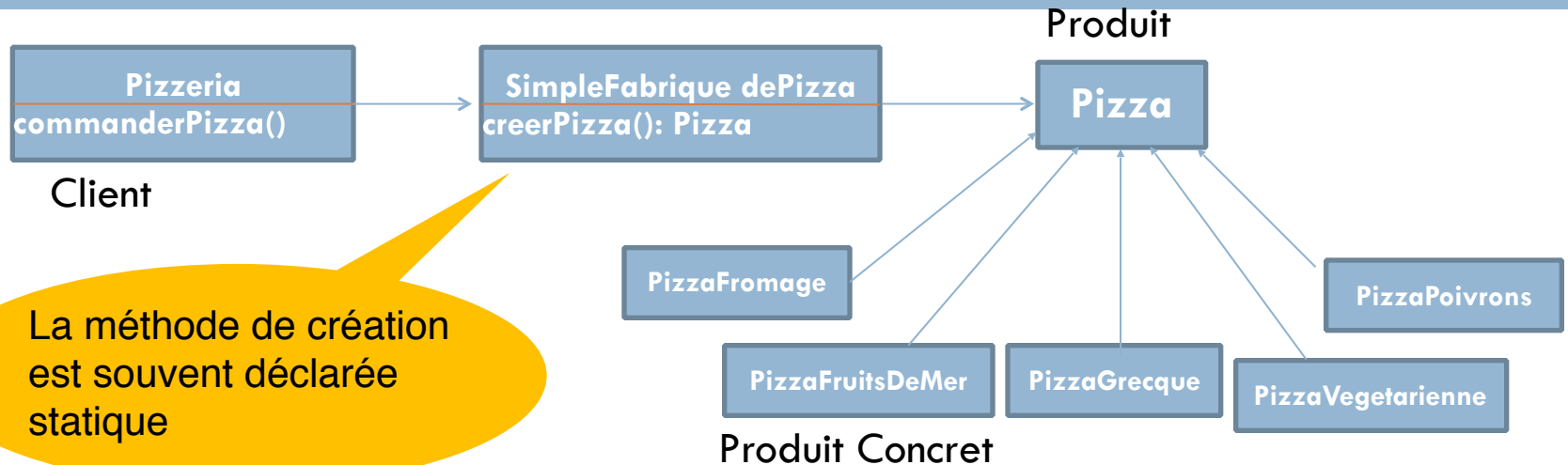
13



- Cette solution appelée **Fabrique Simple** est une bonne pratique de conception mais **pas un design pattern**.
- L'utilisateur du produit (Pizzeria) fait appel à la Fabrique Simple (SimpleFabrique dePizza) pour obtenir un Produit (la Pizza).
  - ▣ C'est la Fabrique Simple qui est chargée d'instancier et de retourner le Produit Concret attendu

# Encapsuler la création des objets

13



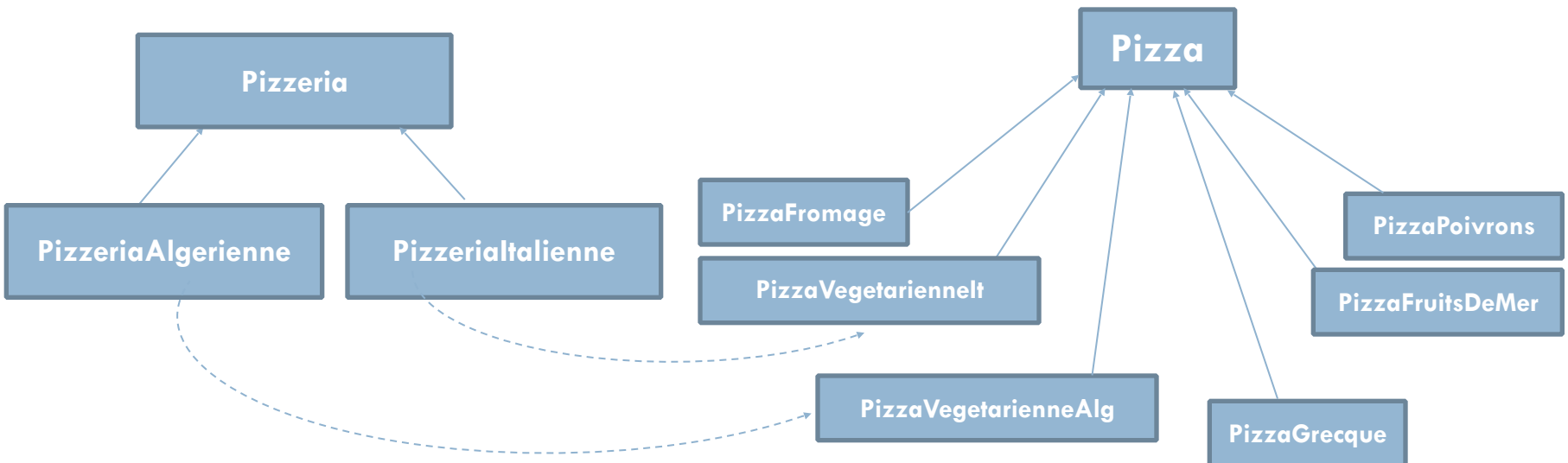
- Cette solution appelée **Fabrique Simple** est une bonne pratique de conception mais **pas un design pattern**.
- L'utilisateur du produit (Pizzeria) fait appel à la Fabrique Simple (SimpleFabrique dePizza) pour obtenir un Produit (la Pizza).
  - ▣ C'est la Fabrique Simple qui est chargée d'instancier et de retourner le Produit Concret attendu



# Laisser les sous-classes décider

14

- Si par la suite l'entreprise évolue et a besoin de plusieurs Pizzerias, chacune spécialisée dans la fabrication de certains Pizzas, Fabrique Simple ne va plus suffire.
  - ▣ Nous aurons un certain nombre de sous-classes concrètes de Pizzeria, chacune ayant ses propres variantes
- Nous allons mettre la méthode `creerPizza()` dans les sous-classes et la rendre responsable de la création du bon type de pizza.



# Déclarer une méthode de fabrique

15

```
public abstract class Pizzeria {  
    public Pizza commanderPizza(String type) {  
        Pizza pizza;  
        pizza = creerPizza(type);  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
    abstract Pizza creerPizza(String type);  
}
```

```
public class PizzeriaAlgerienne extends Pizzeria  
{  
    Pizza creerPizza(String item) {  
        if (choix.equals("fromage")) {  
            return new PizzaFromage ();  
        } else if (choix.equals("vegetarienne")) {  
            return new PizzaVegetarienneAlg();  
        } else return null;  
    }  
}
```

# Déclarer une méthode de fabrique

15

```
public abstract class Pizzeria {  
    public Pizza commanderPizza(String type) {  
        Pizza pizza;  
        pizza = creerPizza(type);  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
    abstract Pizza creerPizza(String type);  
}
```

Toute la responsabilité de l'instanciation des Pizzas a été transférée à une méthode qui se comporte comme une fabrique.

```
public class PizzeriaAlgerienne extends Pizzeria  
{  
    Pizza creerPizza(String item) {  
        if (choix.equals("fromage")) {  
            return new PizzaFromage ();  
        } else if (choix.equals("vegetarienne")) {  
            return new PizzaVegetarienneAlg();  
        }  
        return null;  
    }  
}
```

# Déclarer une méthode de fabrique

15

```
public abstract class Pizzeria {  
    public Pizza commanderPizza(String type) {  
        Pizza pizza;  
        pizza = creerPizza(type);  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
}
```

```
abstract Pizza creerPizza(String type) {  
    }  
}
```

```
public class PizzeriaAlgerienne extends Pizzeria  
{  
    Pizza creerPizza(String item) {  
        if (choix.equals("fromage")) {  
            return new PizzaFromage ();  
        } else if (choix.equals("vegetarienne")) {  
            return new PizzaVegetarienneAlg();  
        }  
        return null;  
    }  
}
```

Toute la responsabilité de l'instanciation des Pizzas a été transférée à une méthode qui se comporte comme une fabrique.

Une méthode de fabrication retourne un Produit qu'on utilise généralement dans les méthodes définies dans la superclasse

# Déclarer une méthode de fabrique

15

```
public abstract class Pizzeria {  
    public Pizza commanderPizza(String type) {  
        Pizza pizza;  
        pizza = creerPizza(type);  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
}
```

```
abstract Pizza creerPizza(String type) {  
    }  
}
```

```
public class PizzeriaAlgerienne extends Pizzeria  
{  
    Pizza creerPizza(String item) {  
        if (choix.equals("fromage")) {  
            return new PizzaFromage ();  
        } else if (choix.equals("vegetarienne")) {  
            return new PizzaVegetarienneAlg();  
        }  
        return null;  
    }  
}
```

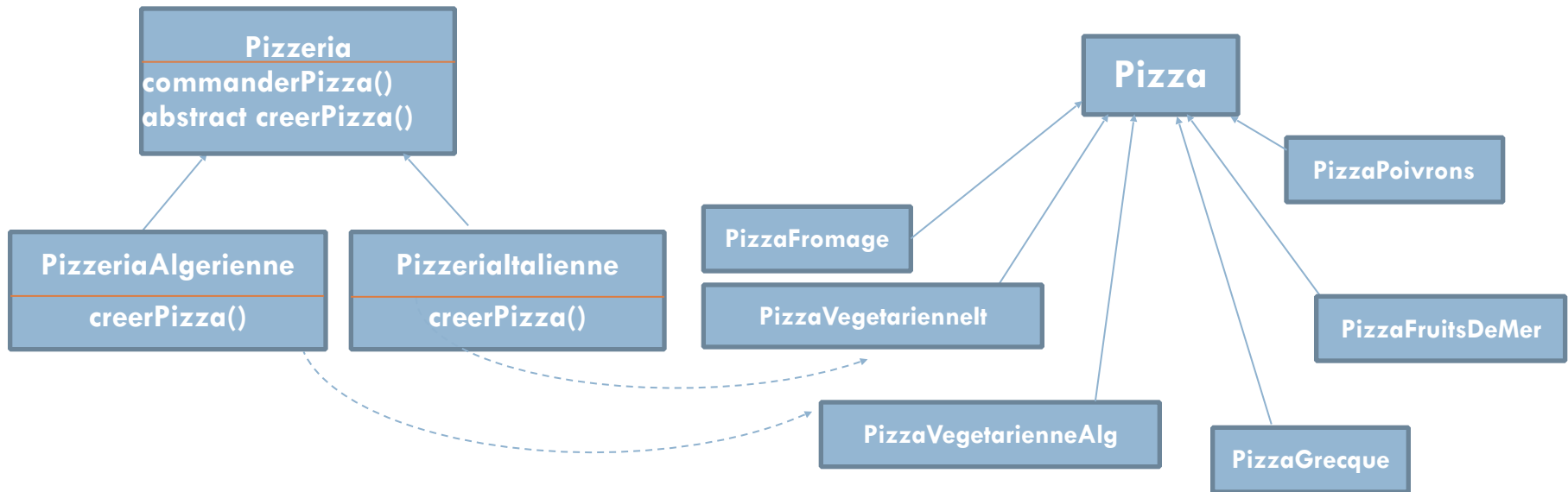
Toute la responsabilité de l'instanciation des Pizzas a été transférée à une méthode qui se comporte comme une fabrique.

Une méthode de fabrication retourne un Produit qu'on utilise généralement dans les méthodes définies dans la superclasse

Une méthode de fabrique **peut être paramétrée (ou non)** pour choisir entre différentes variantes d'un produit (ou un seul type)

# Le pattern Factory Method

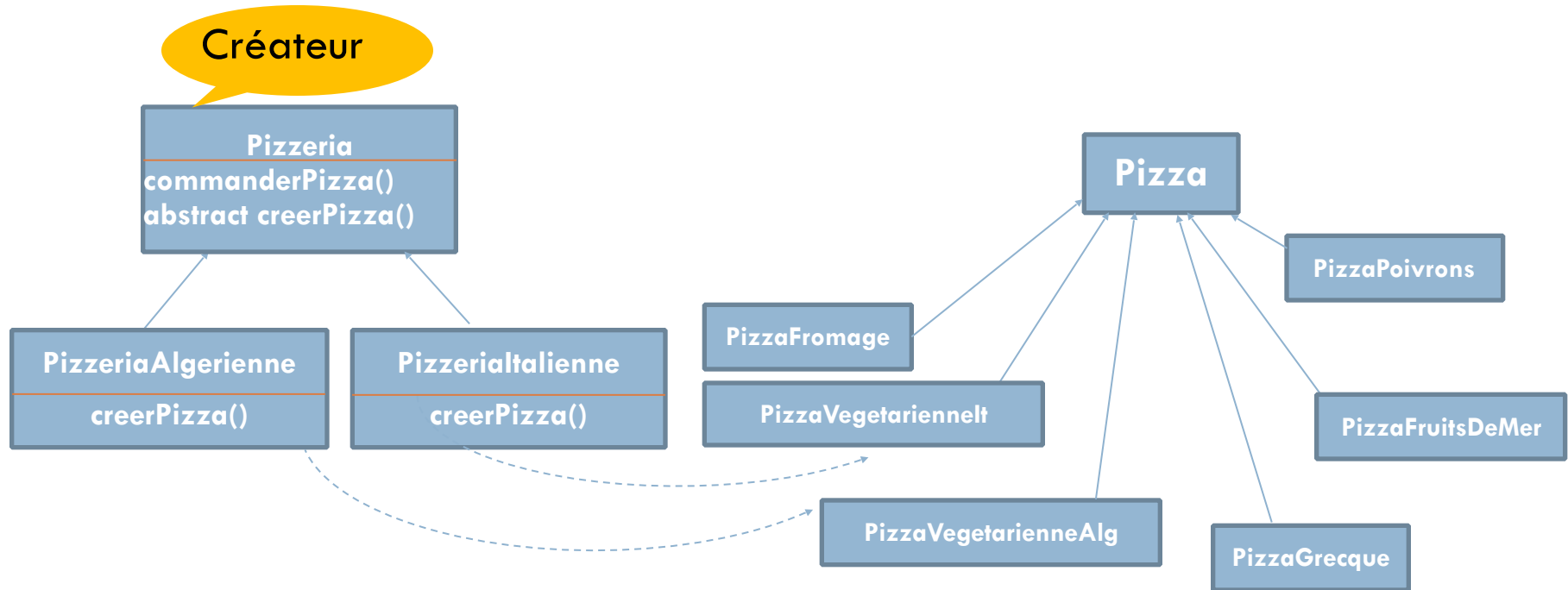
16



**Le pattern Fabrication (Factory Method) définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Elle permet à une classe de déléguer l'instanciation à des sous-classes.**

# Le pattern Factory Method

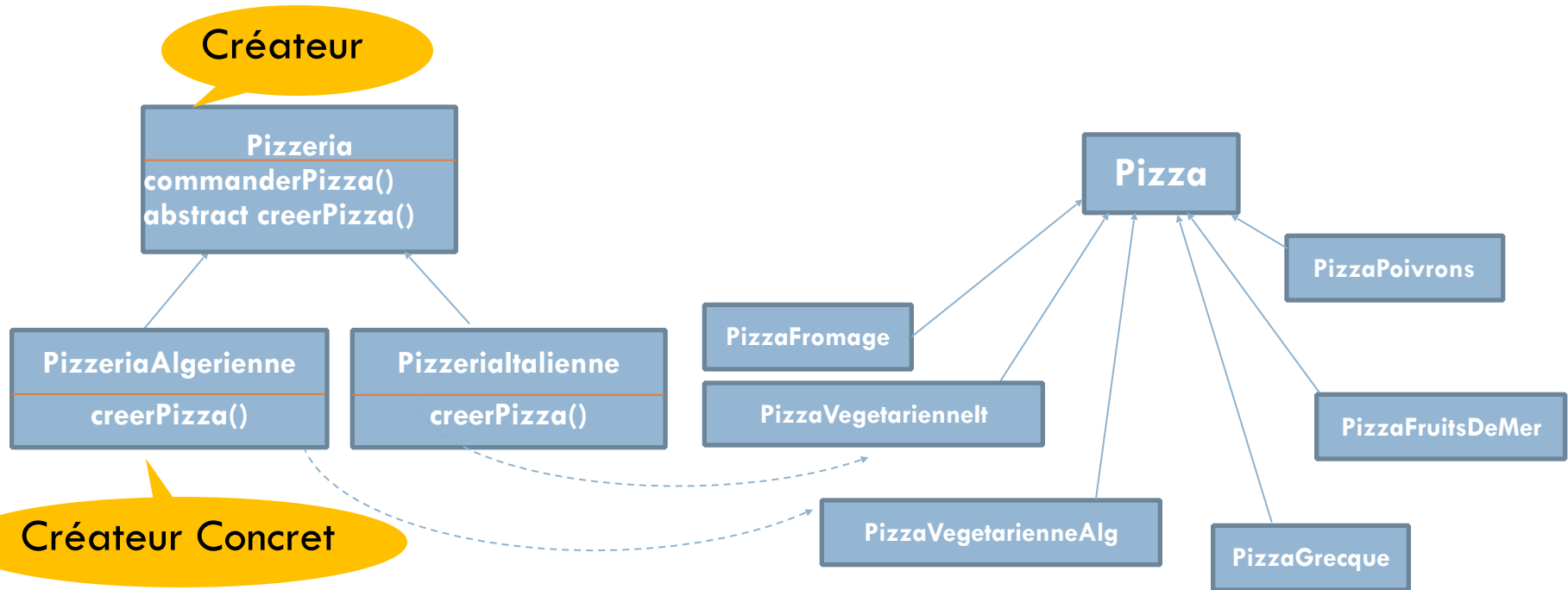
16



**Le pattern Fabrication (Factory Method) définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Elle permet à une classe de déléguer l'instanciation à des sous-classes.**

# Le pattern Factory Method

16

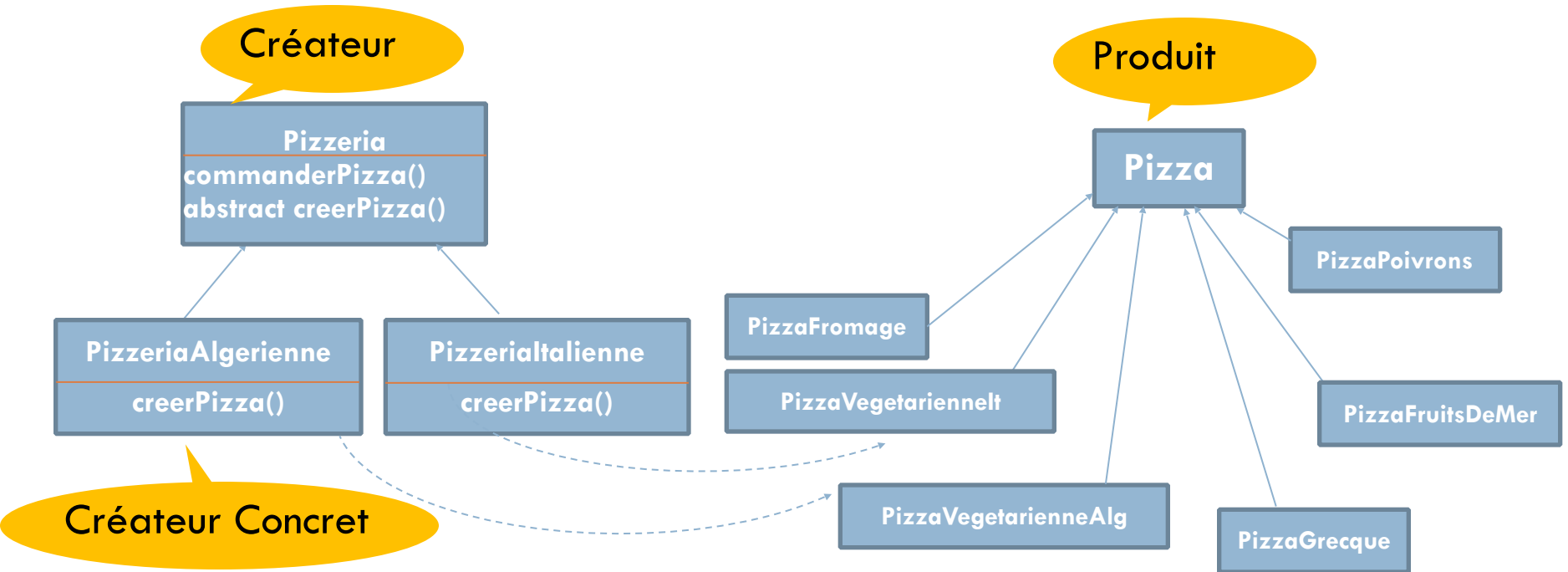


**Le pattern Fabrication (Factory Method) définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Elle permet à une classe de déléguer l'instanciation à des sous-classes.**



# Le pattern Factory Method

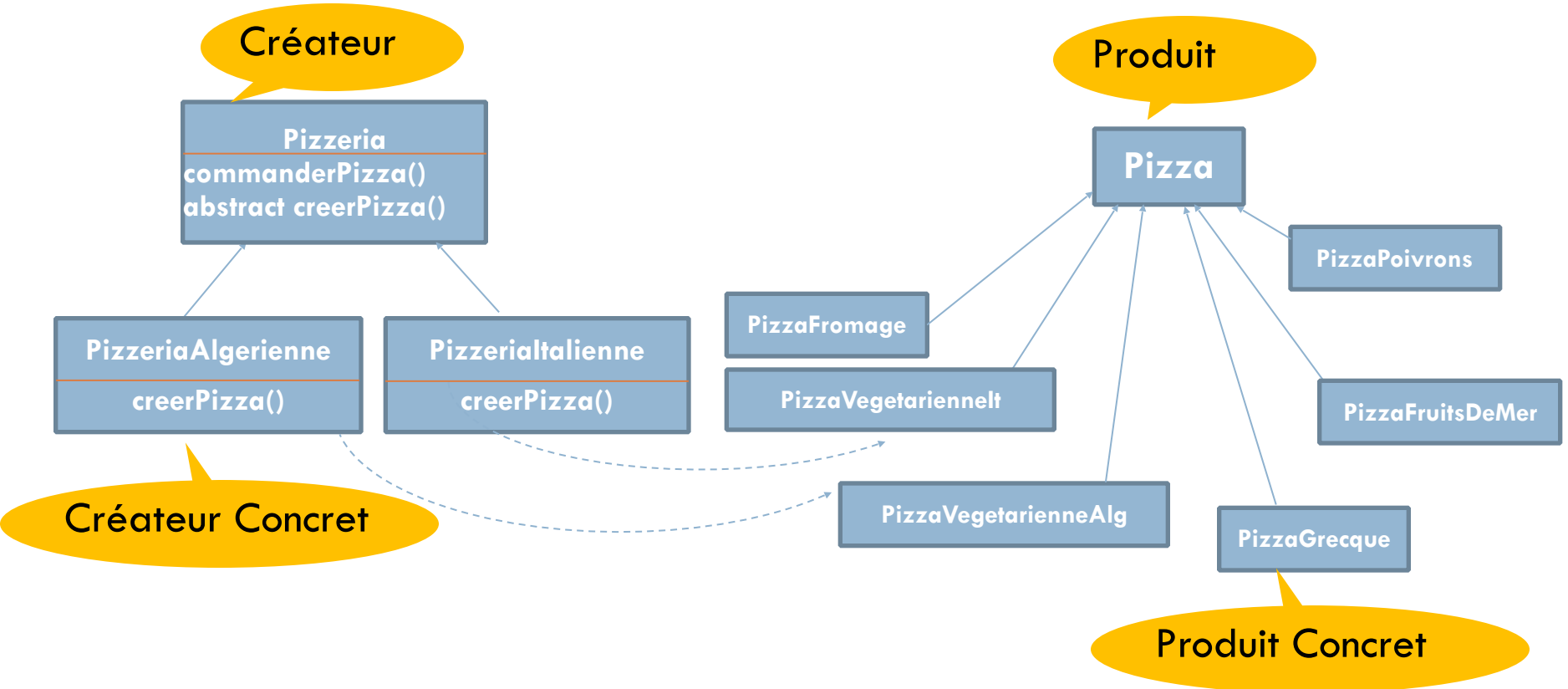
16



**Le pattern Fabrication (Factory Method) définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Elle permet à une classe de déléguer l'instanciation à des sous-classes.**

# Le pattern Factory Method

16



**Le pattern Fabrication (Factory Method) définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Elle permet à une classe de déléguer l'instanciation à des sous-classes.**

# Description du pattern Factory Method

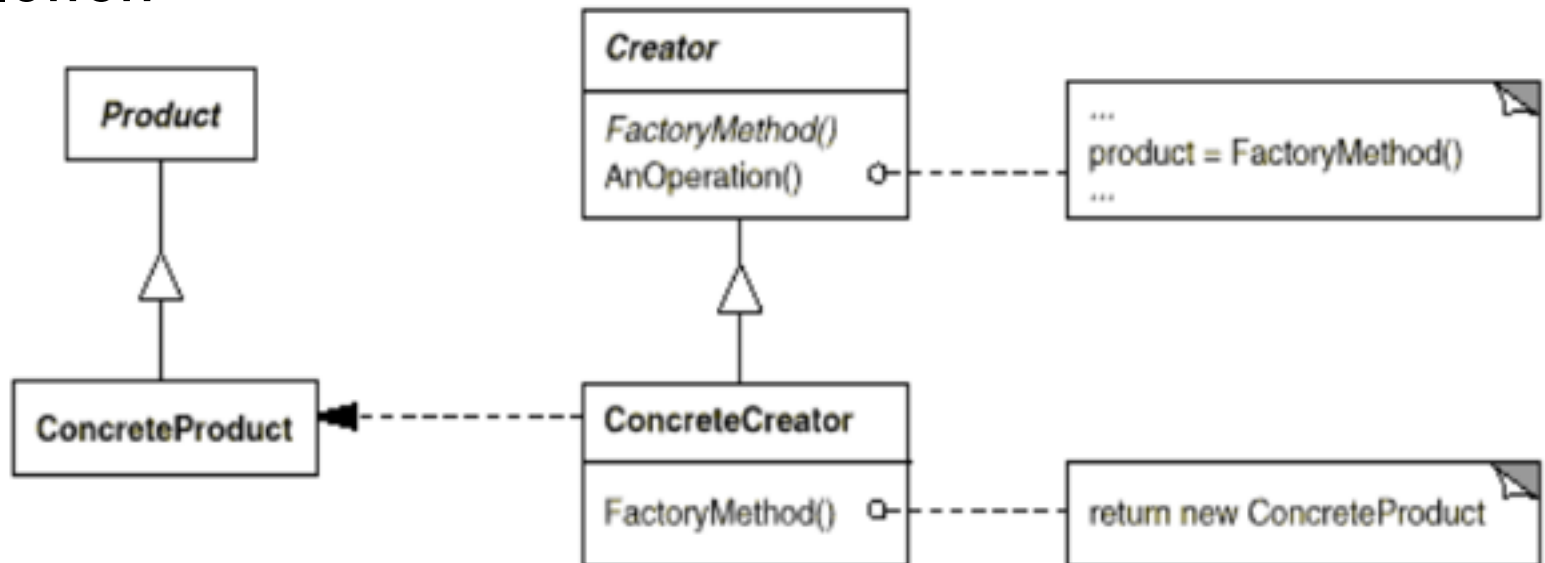
17

- Intention
  - ▣ Choisit la bonne sous-classe en fonction de certains paramètres
  - ▣ Permet la délégation d'instanciation : cache la classe concrète utilisée (permet l'évolution)
  
- *Utilisations connues*
  - ▣ Applications graphiques, un peu partout...
  
- *Synonyme*
  - ▣ *Constructeur virtuel, Fabrique, Fabrication, Usine*
  
- *Patterns associés (en relation)*
  - ▣ *Abstract Factory, Template Method, Prototype*

# Description du pattern Factory Method

18

- Problème (Applicabilité, Quand l'utiliser?)
  - ▣ Une classe est incapable d'anticiper le type d'objets qu'elle doit créer
  - ▣ Une classe désire laisser le choix du type d'objets créés à ses sous-classes
- Solution



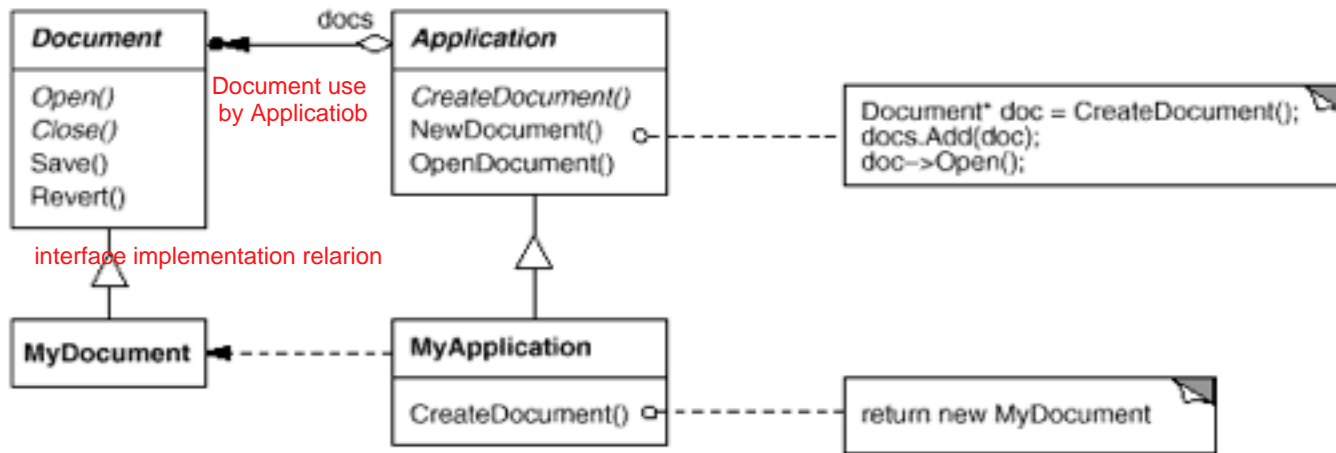
# Description du pattern Factory Method

19

- Participants
  - ▣ **Product** : définit l'interface des objets créés par la fabrication
  - ▣ **ConcreteProduct** : implémente l'interface Product
  - ▣ **Creator** : 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** : surcharge la fabrication pour renvoyer une instance d'un *ConcreteProduct*

# Example

20



```

public interface Document { ... }
public class PlainTextDocument implements Document { ... }
    
```

```

public abstract class Application {
    public Document newDocument() { ... }
    public Document openDocument() { ... }
    protected abstract Document createDocument();
}

public class MyApplication extends Application {
    protected Document createDocument() {
        return new PlainTextDocument(); ConcreteProductIntantiation
    }
}
    
```

# Conséquences

21

- **Avantage :** Elimine la nécessité de lier les classes spécifiques à l'application dans votre code (le client).
  - Le code ne traite que l'interface du produit; il peut donc travailler avec toutes classe ConcreteProduct définie par l'utilisateur.
- **Inconvénient :** Les clients pourraient avoir à sous-classer la classe Créateur juste pour créer un objet ConcreteProduct particulier.

# Implantation

22

- Deux variétés principales
  - Le Créateur ne fournit pas une implémentation pour la méthode de fabrique
  - Le Créateur fournit une implémentation par défaut de la méthode de fabrique.
  
- Factory Method paramétrée
  - Créer plusieurs types de produits
  - La méthode de fabrication prend un paramètre qui identifie le type d'objet à créer.
  - L'objet partage l'interface du produit



# PATRONS DE CRÉATION


Abstract Factory

# Gérer différentes familles

24

- Conception d'un système d'**affichage** et d'**impression** de formes géométrique
  - ⇒ Prendre en compte la capacité de l'ordinateur
  - ⇒ Contrôler les pilotes utilisés

Action du pilote	Pour une machine à faible capacité, utiliser	Pour une machine à grande capacité, utiliser
<b>Affichage</b>	<b><i>Pilote d’Affichage à Basse Résolution (PABR)</i></b>	<b><i>Pilote d’Affichage à Haute Résolution (PAHR)</i></b>
<b>Impression</b>	<b><i>Pilote d’Impression à Basse Résolution (PIBR)</i></b>	<b><i>Pilote d’Impression à Haute Résolution (PIHR)</i></b>



Famille à basse résolution

Famille à Haute résolution

# Première solution: utiliser un switch

25

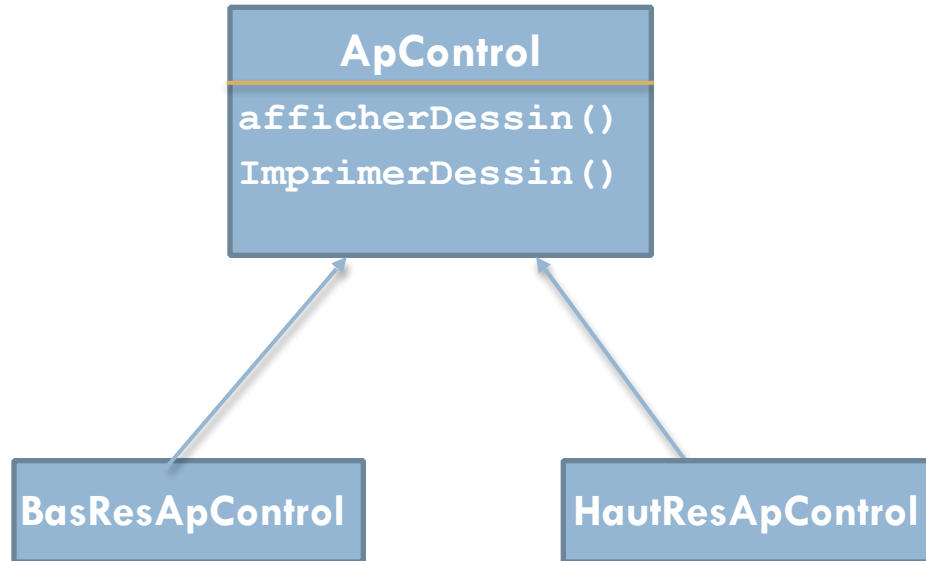
```
class ApControl {
...
void afficherDessin () {
...
    switch (RESOLUTION) {
        case BASSE:
            // utiliser PABR
        case HAUTE:
            // utiliser PAHR
    } }

void ImprimerDessin () {
...
    switch (RESOLUTION) {
        case BASSE:
            // utiliser PIBR
        case HAUTE:
            // utiliser PIHR
    } }
}
```

- Les règles de sélection du pilote sont confondues avec son utilisation
  - ▣ Fort couplage : Si la règle de résolution change (ajout d'une valeur Intermédiaire de résolution), le code doit changer à deux endroits qui sont par ailleurs pas liés.
  - ▣ Faible cohésion: `afficherDessin` et `ImprimerDessin` ont deux missions indépendantes: ils doivent à la fois créer une forme et se préoccuper du pilote à utiliser.

# Seconde Alternative: L'héritage

26



- Pour chaque nouvelle famille, on doit créer une nouvelle sous-classe de **ApControl**
  - Ne respecte pas le principe « préférer la composition à l'héritage »

# Seconde Alternative: L'héritage

26



Construit et instancie des Pilotes BR et les assignes aux méthodes afficher et Imprimer

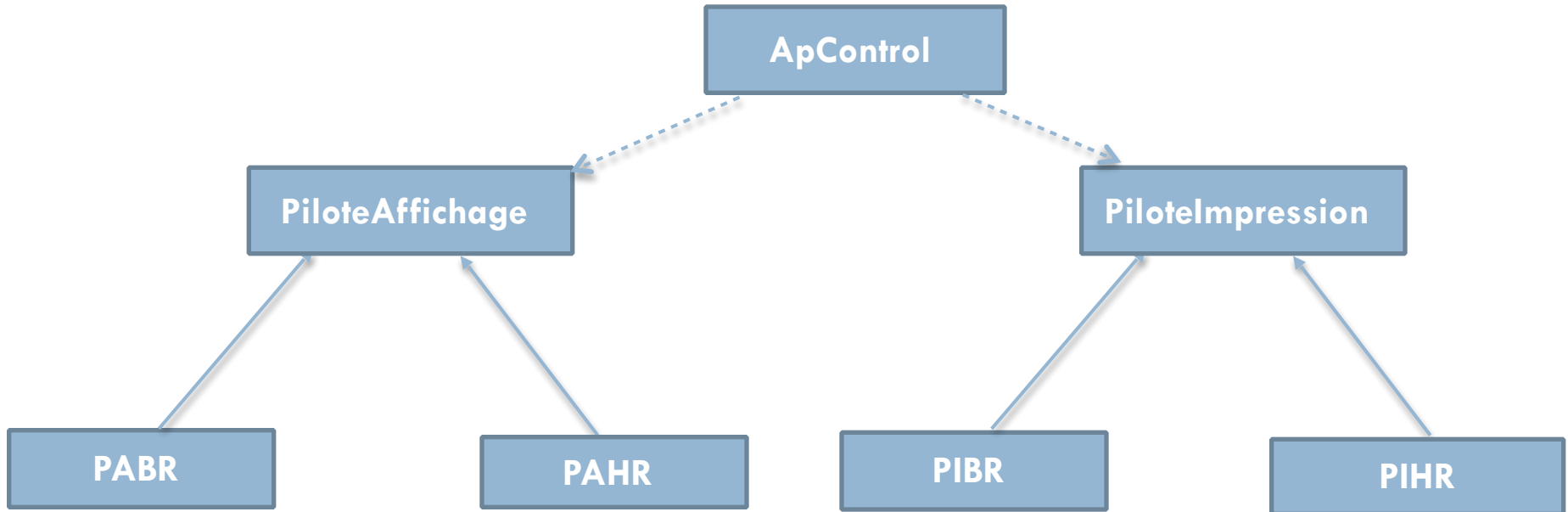
**BasResApControl**

**HautResApControl**

- Pour chaque nouvelle famille, on doit créer une nouvelle sous-classe de ApControl
  - Ne respecte pas le principe « préférer la composition à l'héritage »

# Faire des Abstractions

27



```
class ApControl {
...
PiloteAffichage monPiloteAff;
PiloteImpression monPiloteImp;

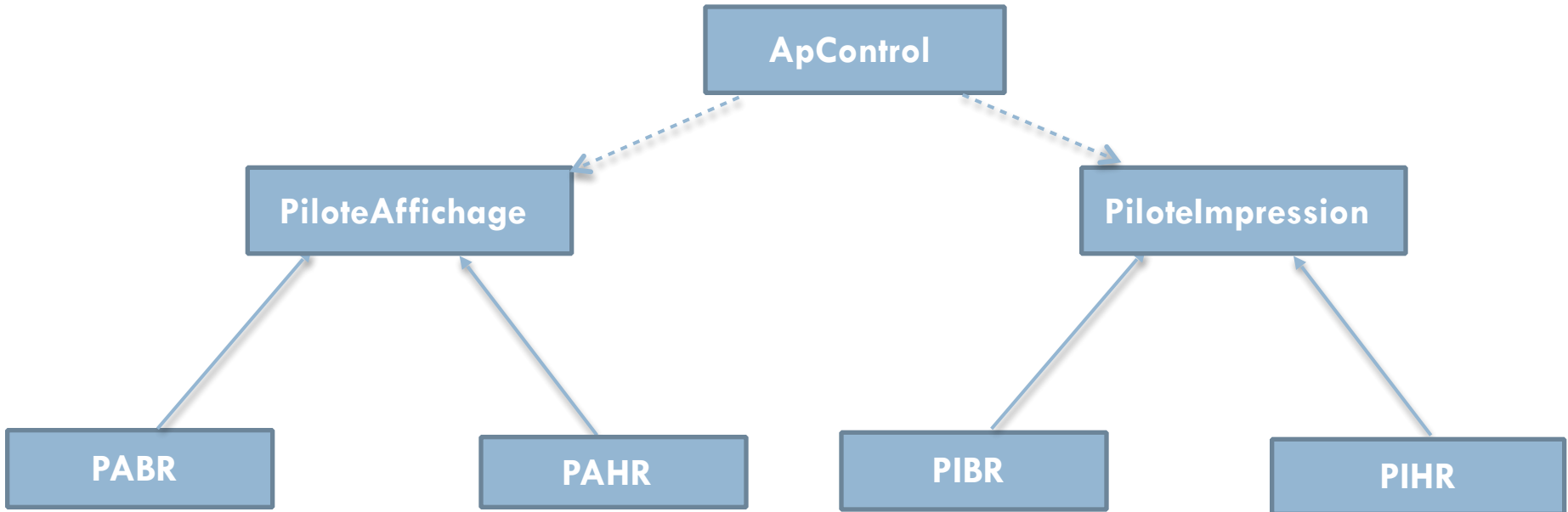
void afficherDessin () {
...
monPiloteAff.afficher();
}
```

```

}
void ImprimerDessin () {
...
monPiloteImp.Imprimer();
}
}
```

# Faire des Abstractions

27



```
class ApControl {
...
PiloteAffichage monPiloteAff;
PiloteImpression monPiloteImp;

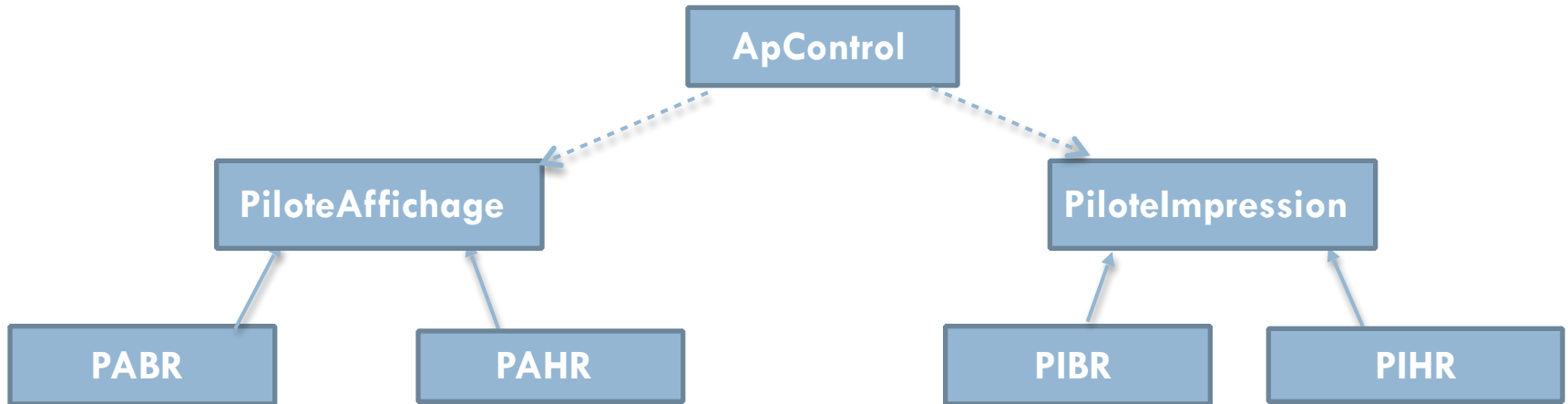
void afficherDessin () {
...
monPiloteAff.afficher();
}
```

```
}
void ImprimerDessin () {
...
monPiloteImp.Imprimer();
}
```

Comment créer ces objets  
selon les deux familles?

# Encapsuler la création d'objet

28

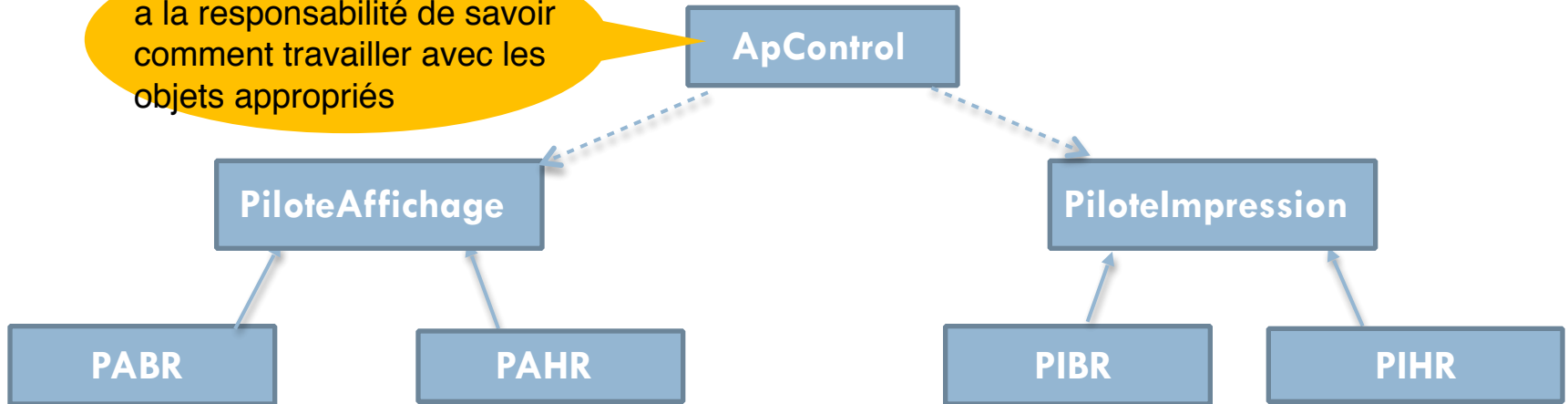




# Encapsuler la création d'objet

28

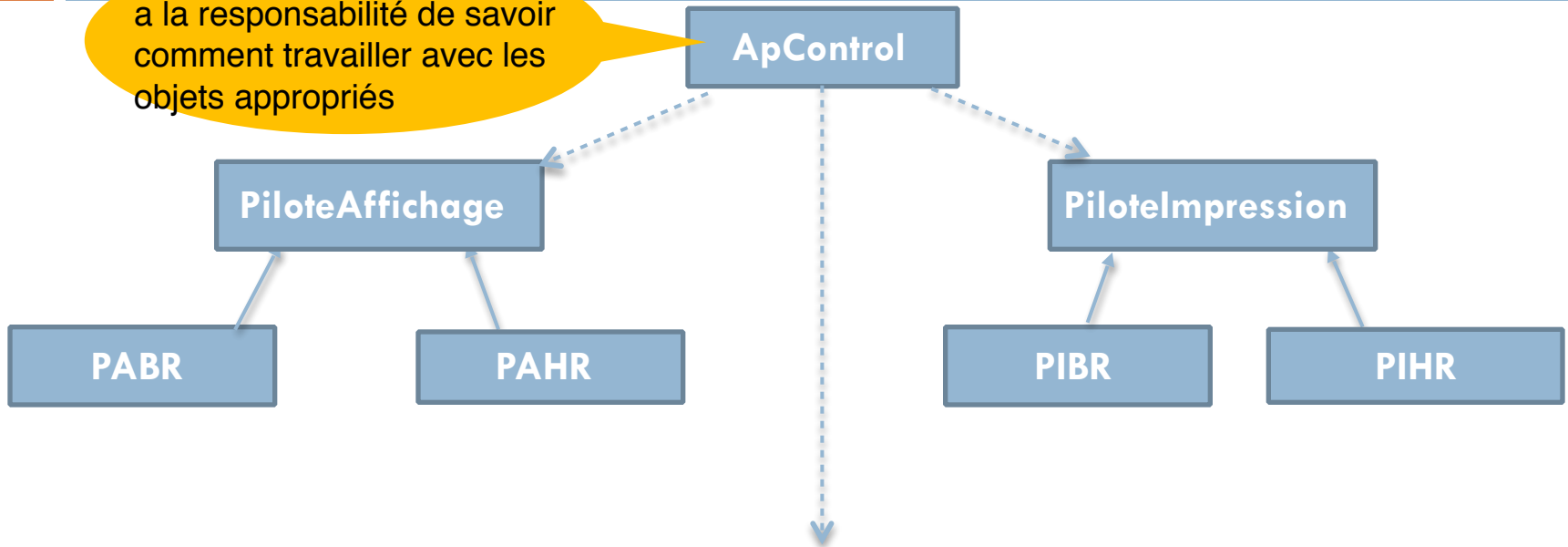
a la responsabilité de savoir  
comment travailler avec les  
objets appropriés



# Encapsuler la création d'objet

28

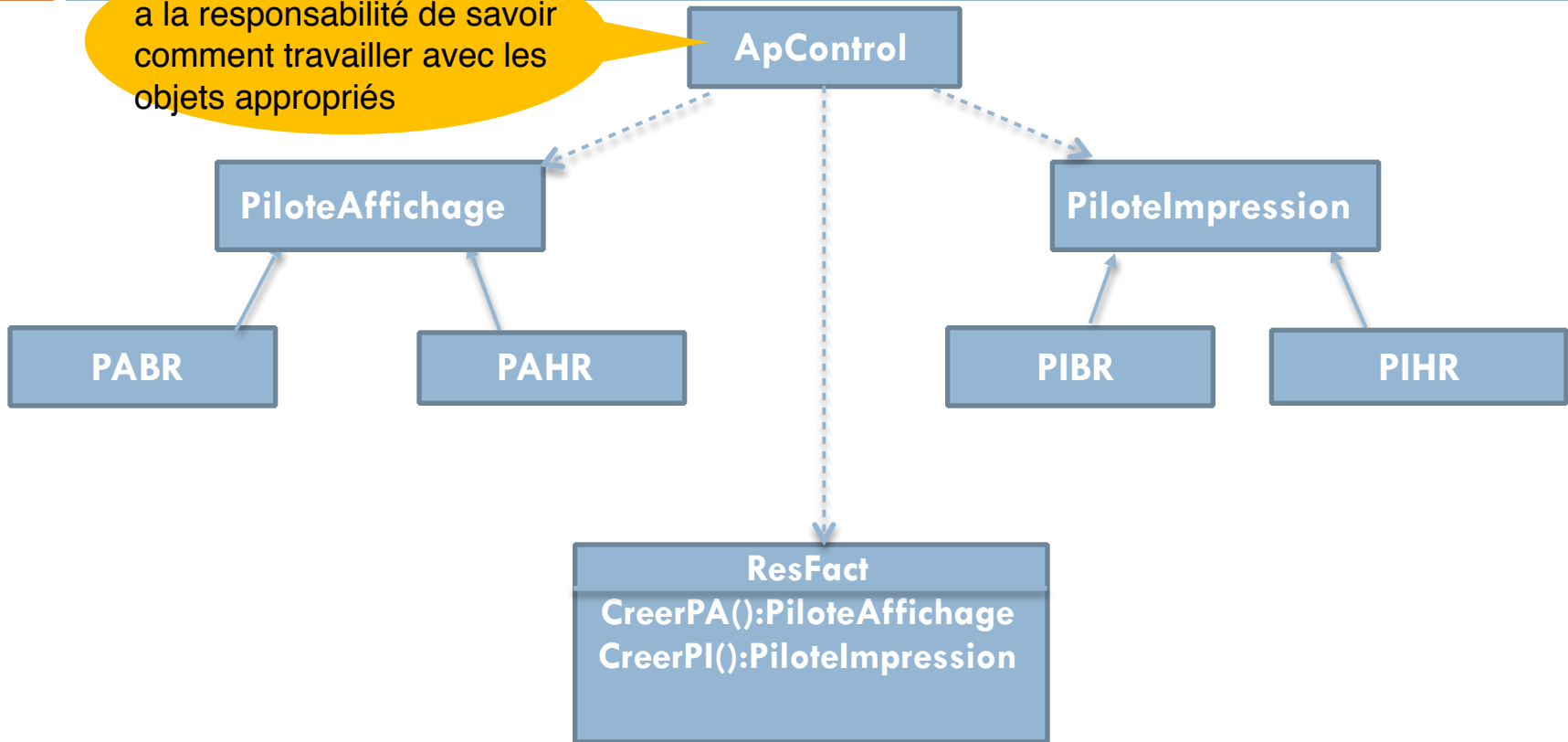
a la responsabilité de savoir  
comment travailler avec les  
objets appropriés



# Encapsuler la création d'objet

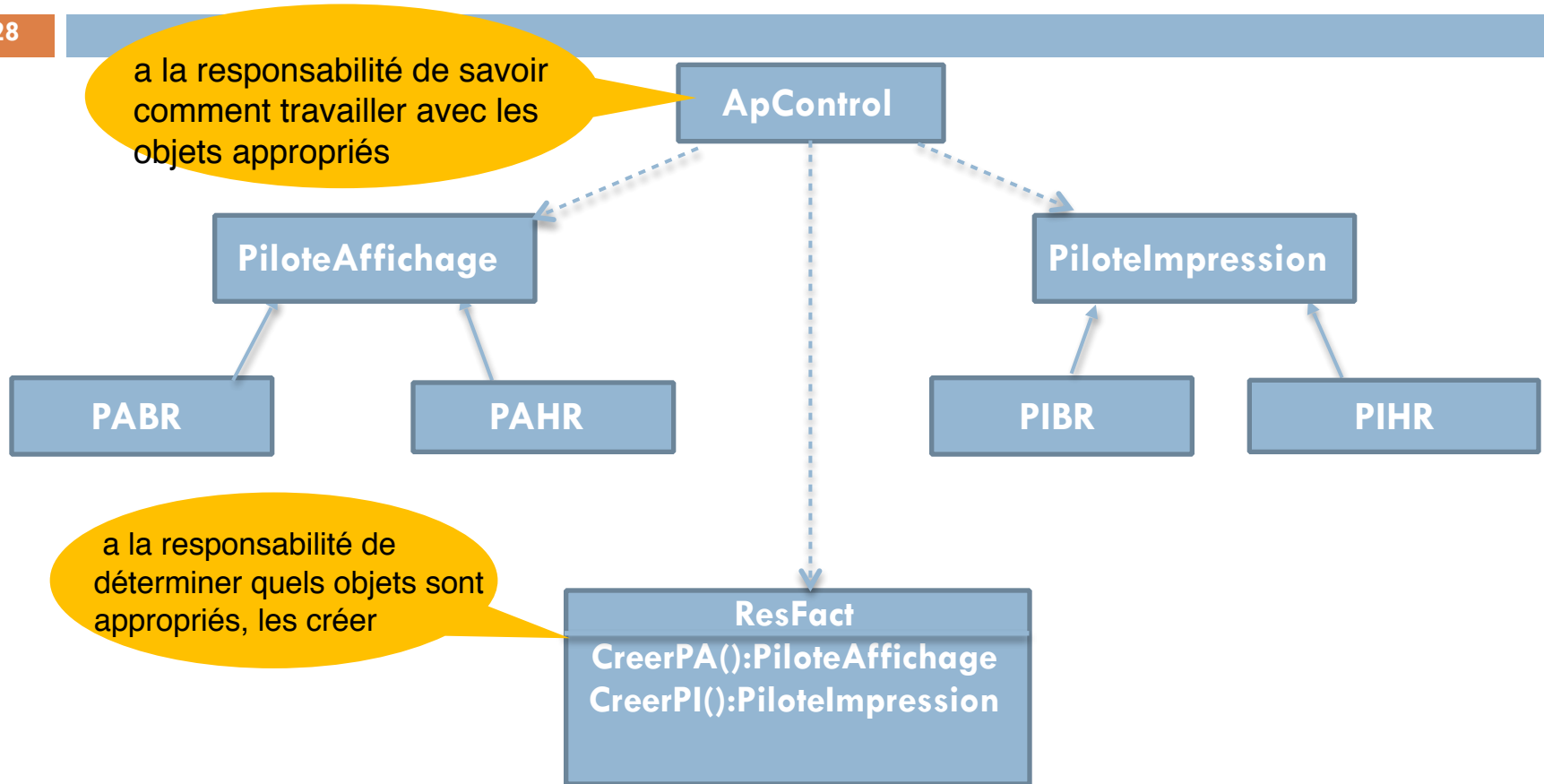
28

a la responsabilité de savoir  
comment travailler avec les  
objets appropriés



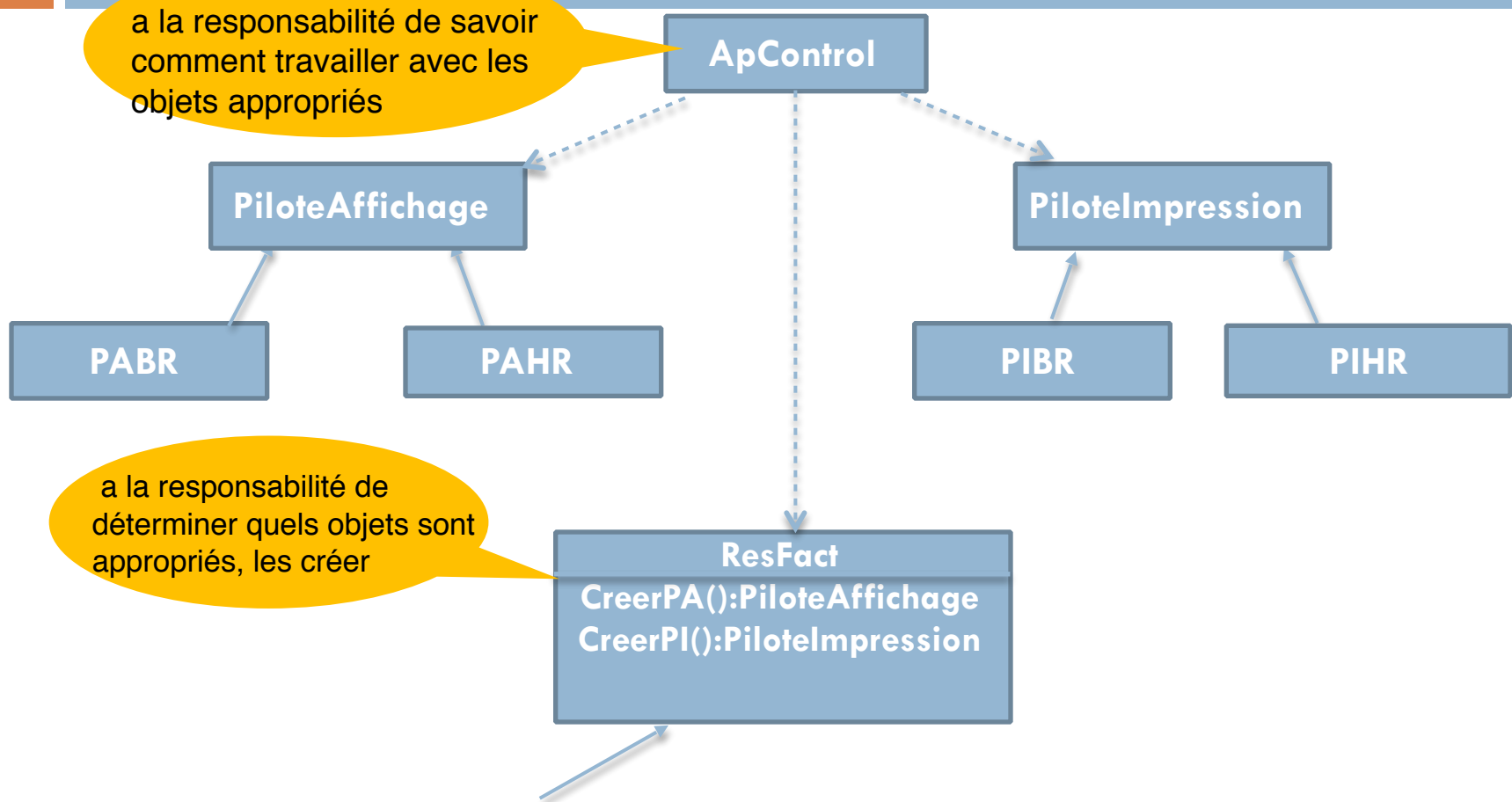
# Encapsuler la création d'objet

28



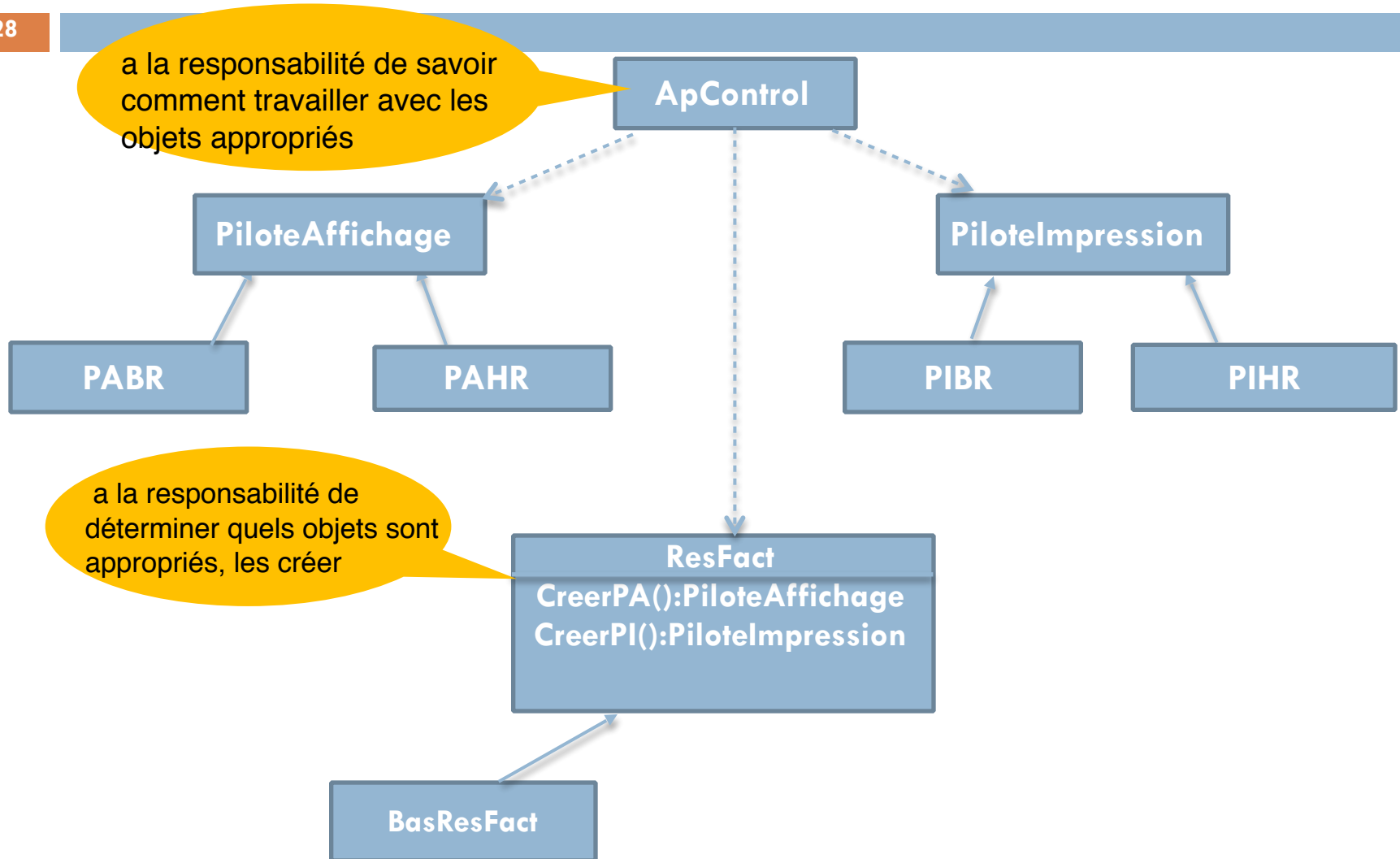
# Encapsuler la création d'objet

28



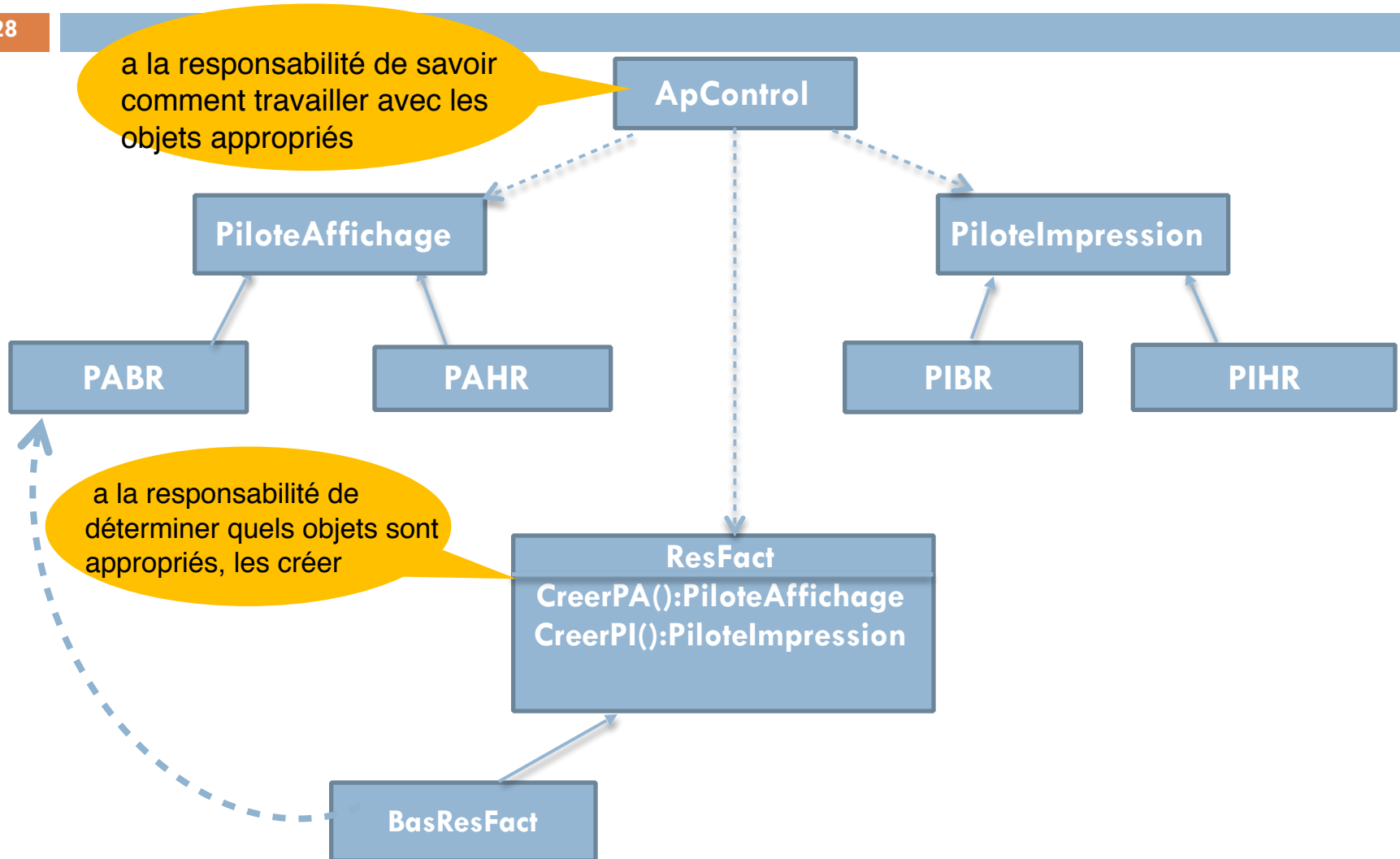
# Encapsuler la création d'objet

28



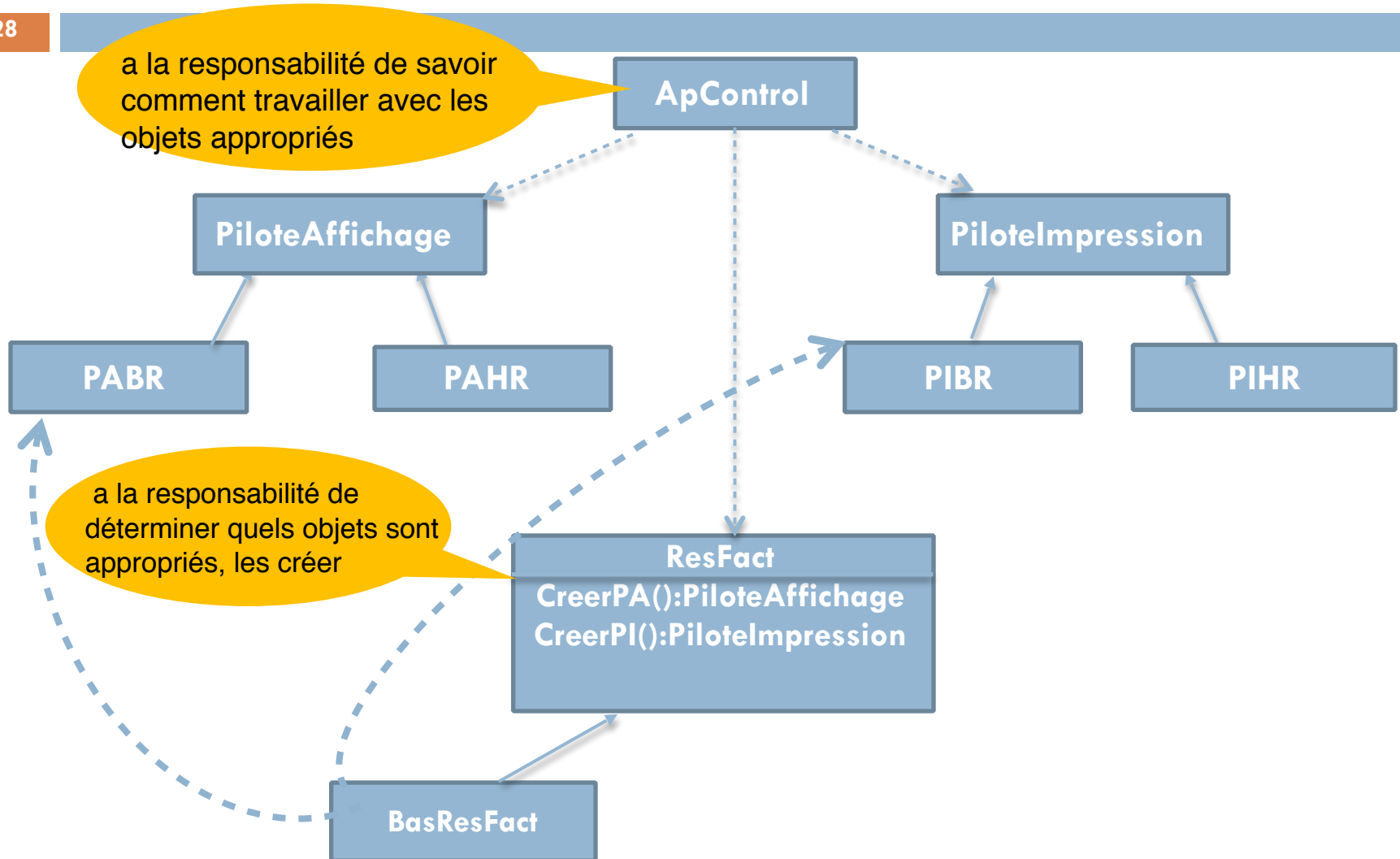
# Encapsuler la création d'objet

28



# Encapsuler la création d'objet

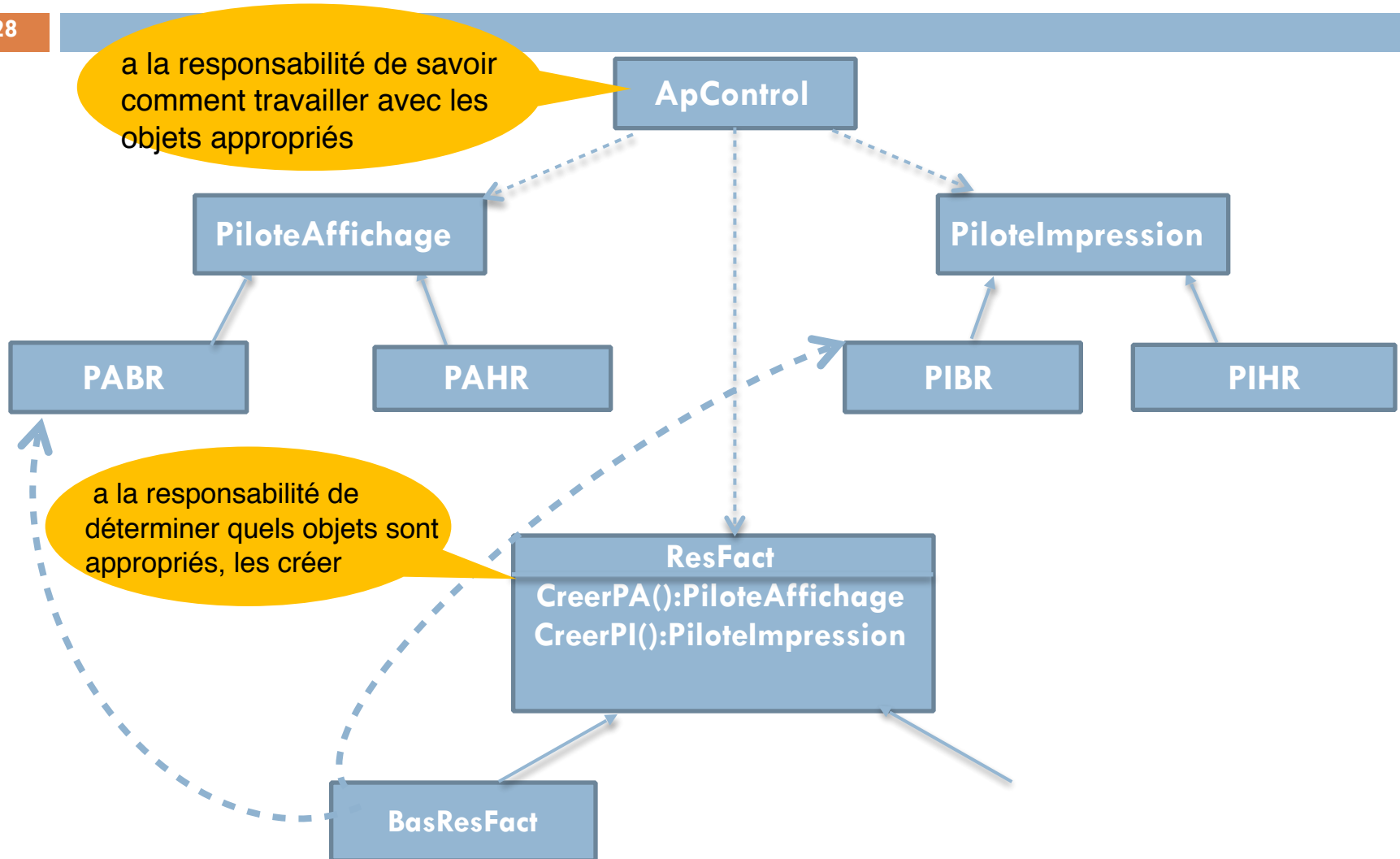
28





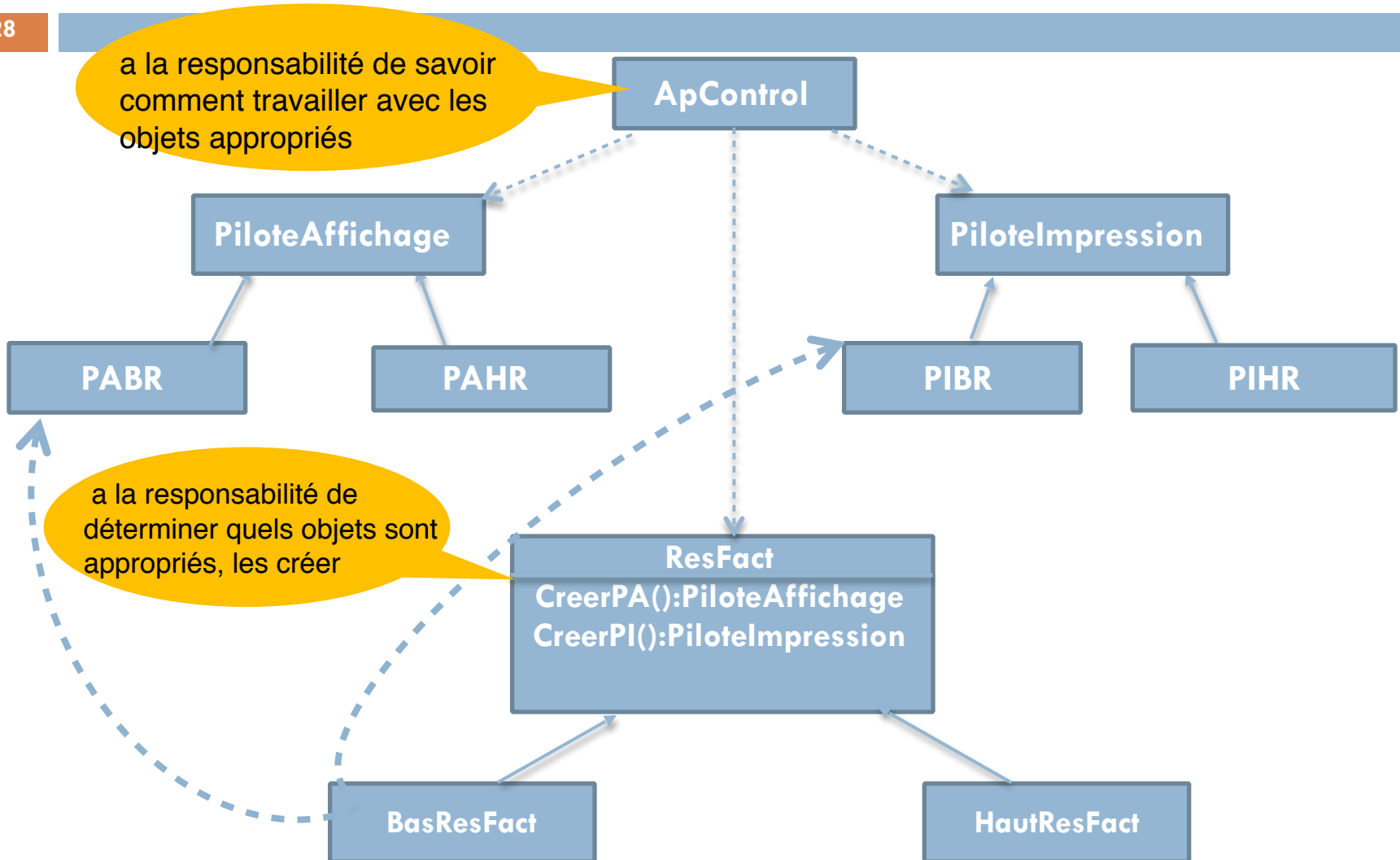
# Encapsuler la création d'objet

28



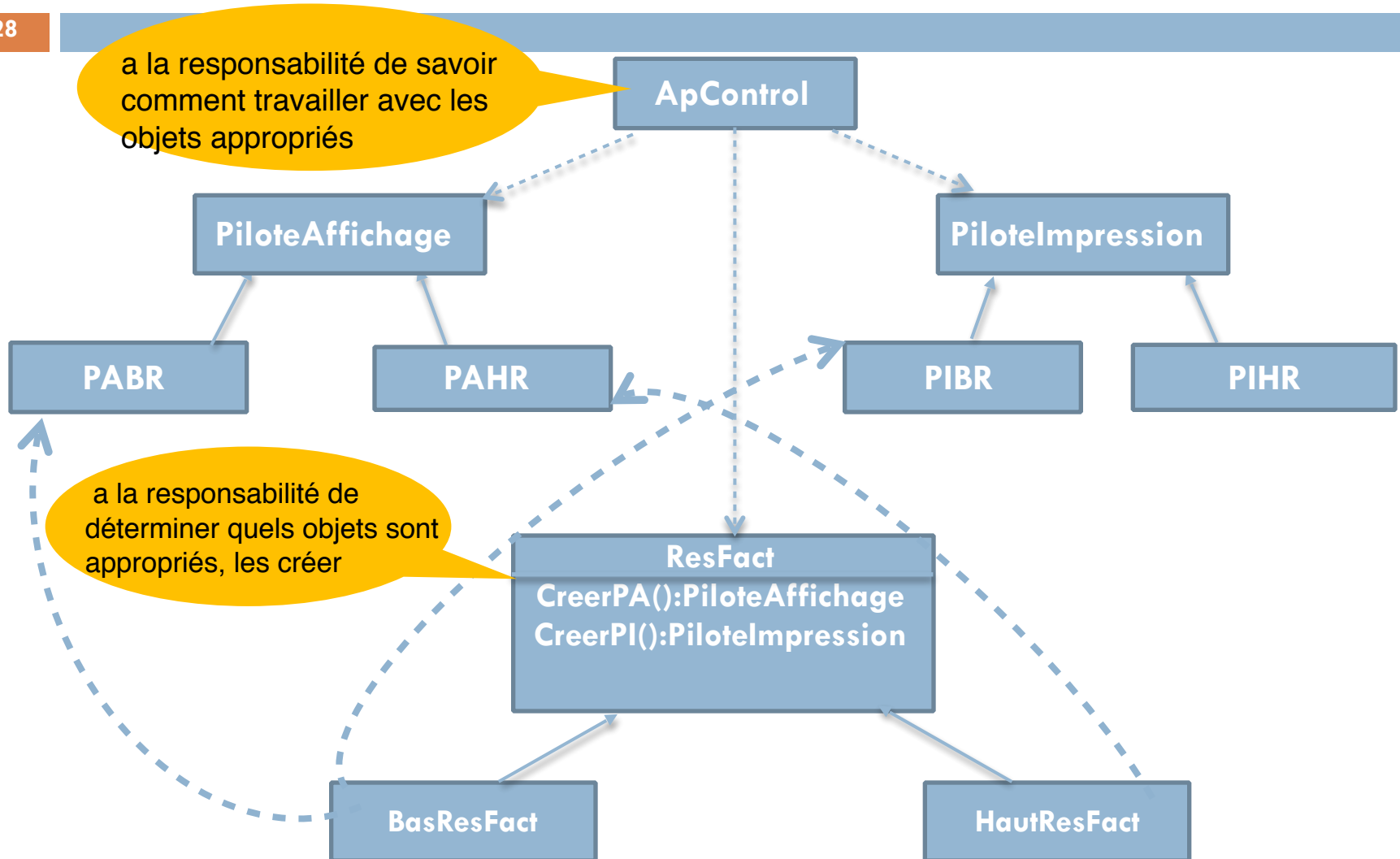
# Encapsuler la création d'objet

28



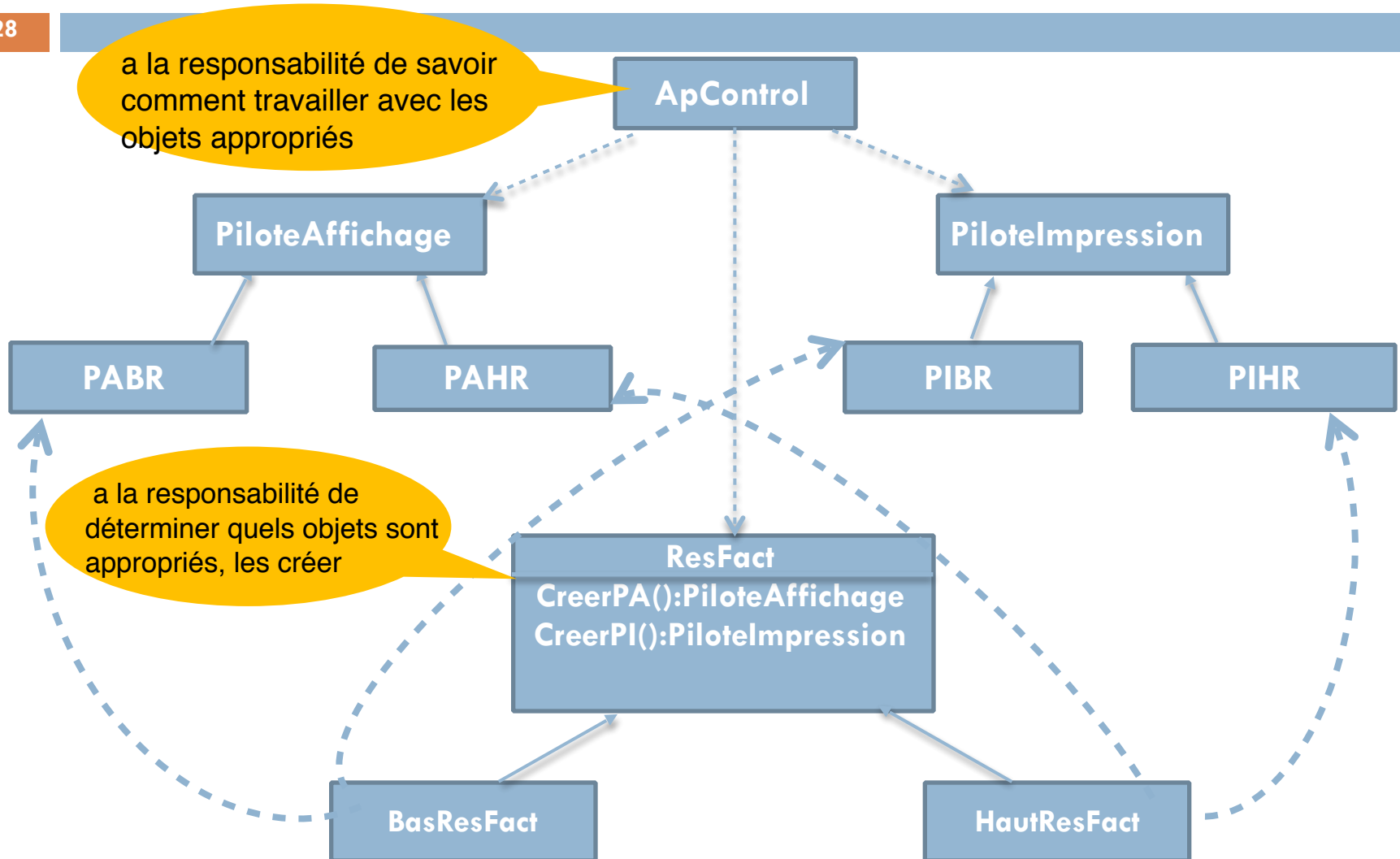
# Encapsuler la création d'objet

28



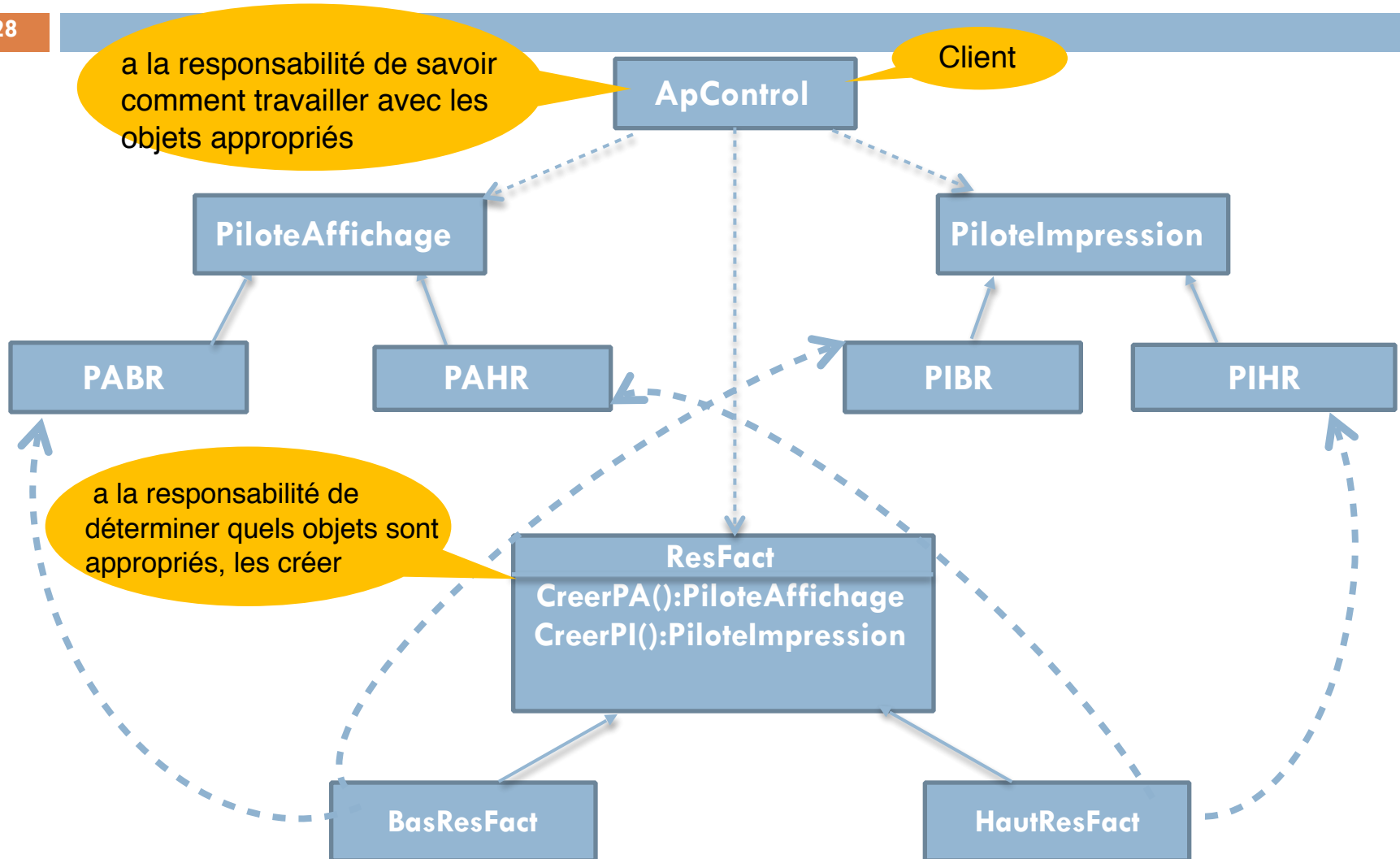
# Encapsuler la création d'objet

28



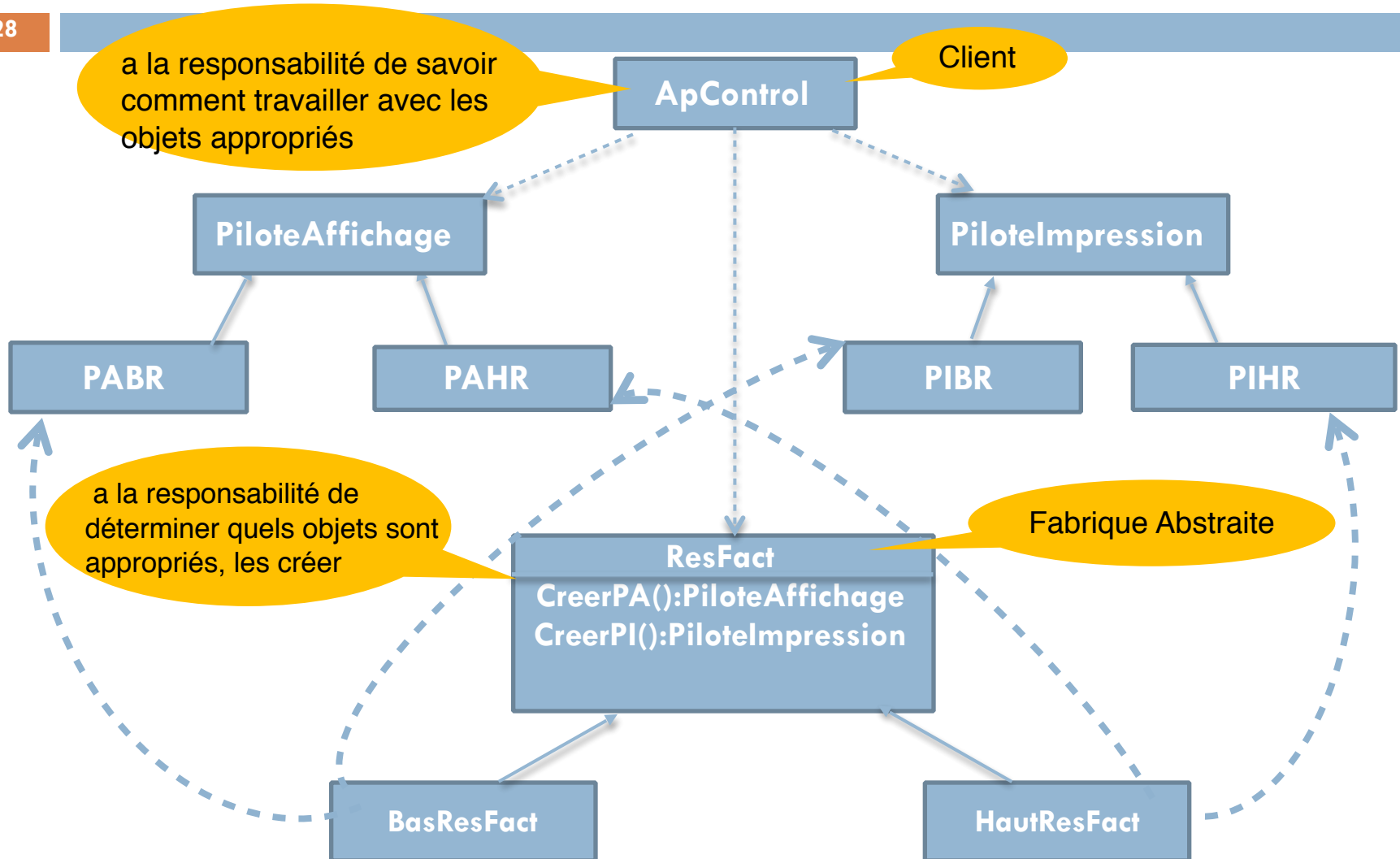
# Encapsuler la création d'objet

28



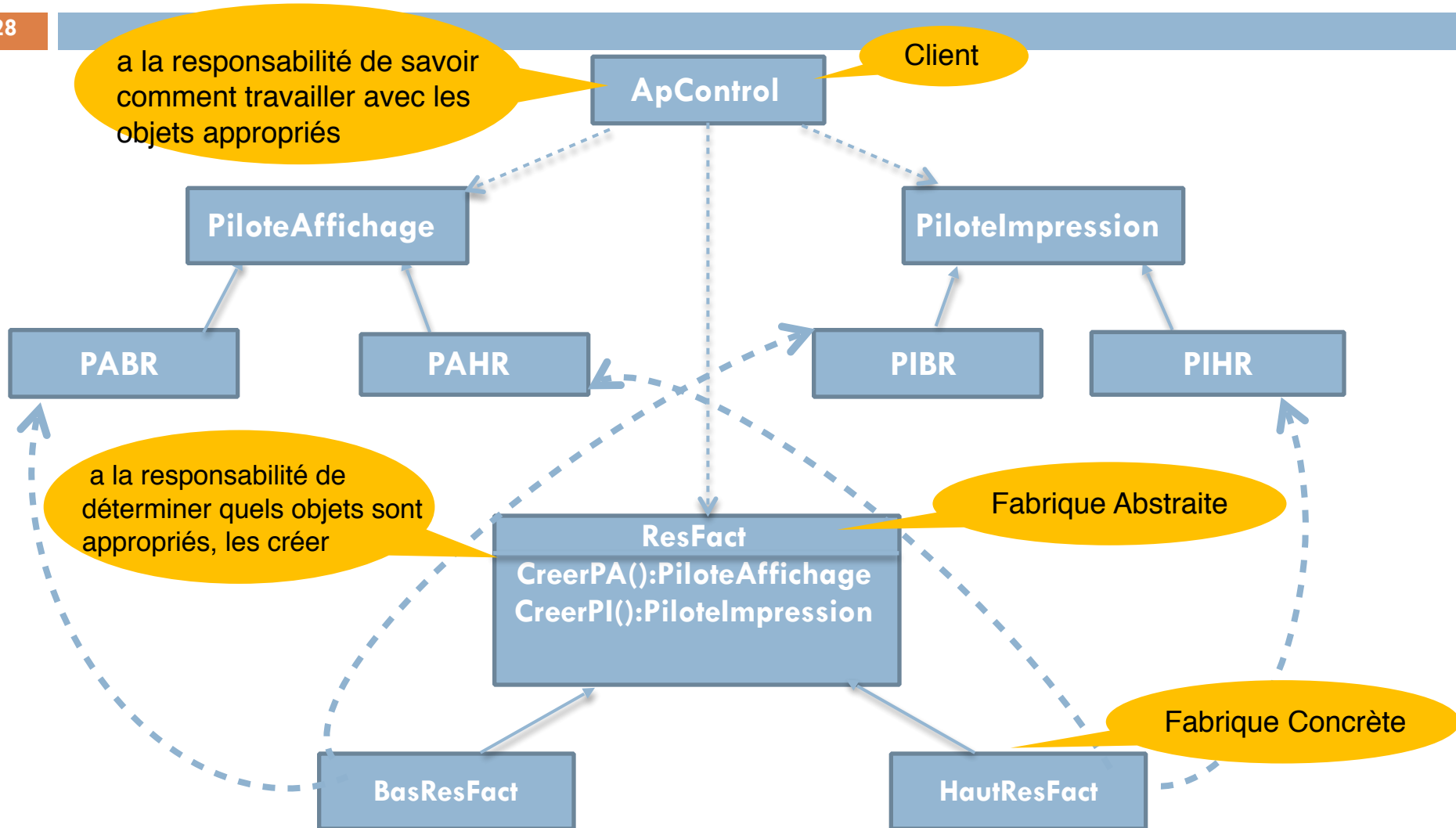
# Encapsuler la création d'objet

28



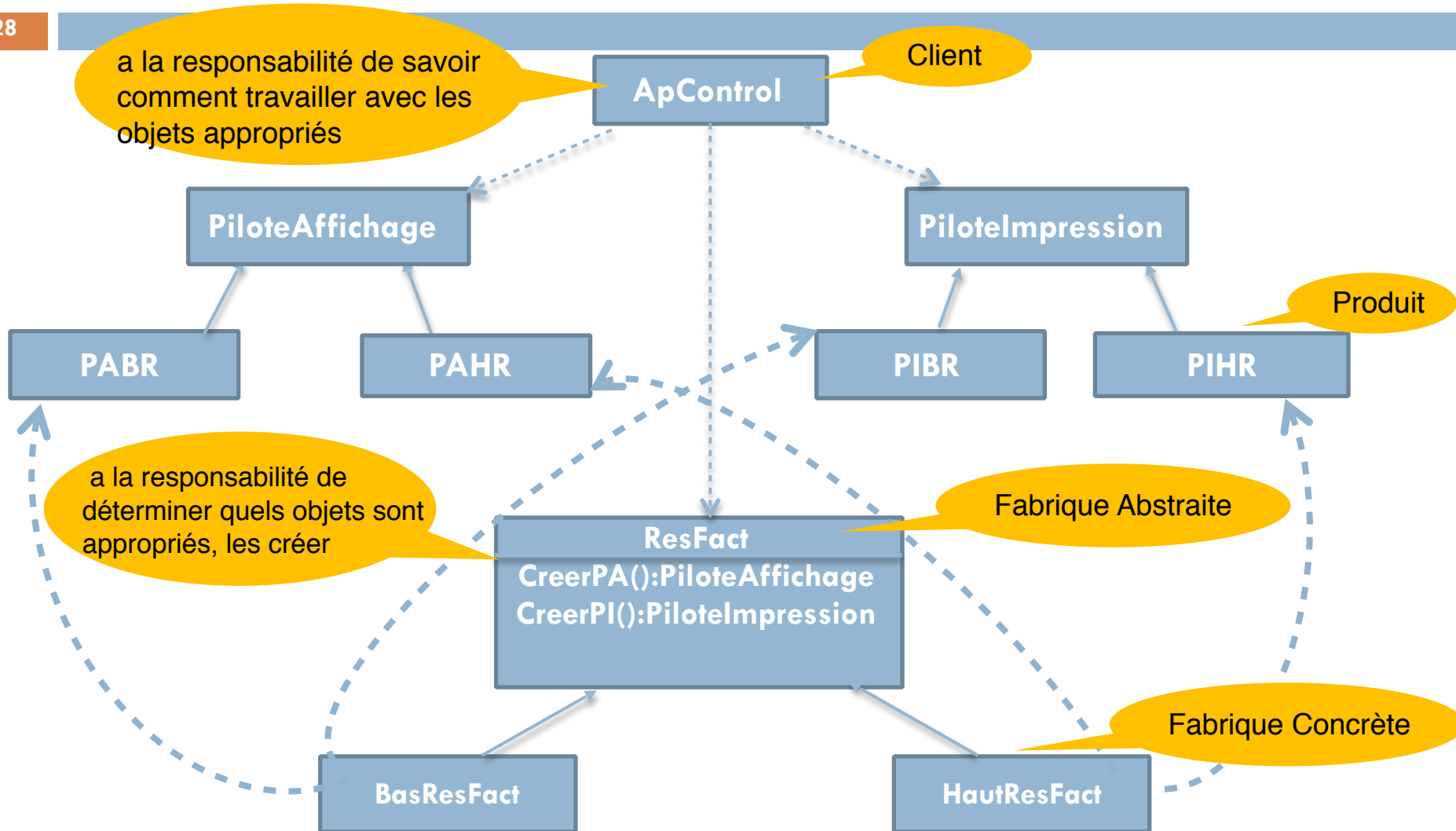
# Encapsuler la création d'objet

28



# Encapsuler la création d'objet

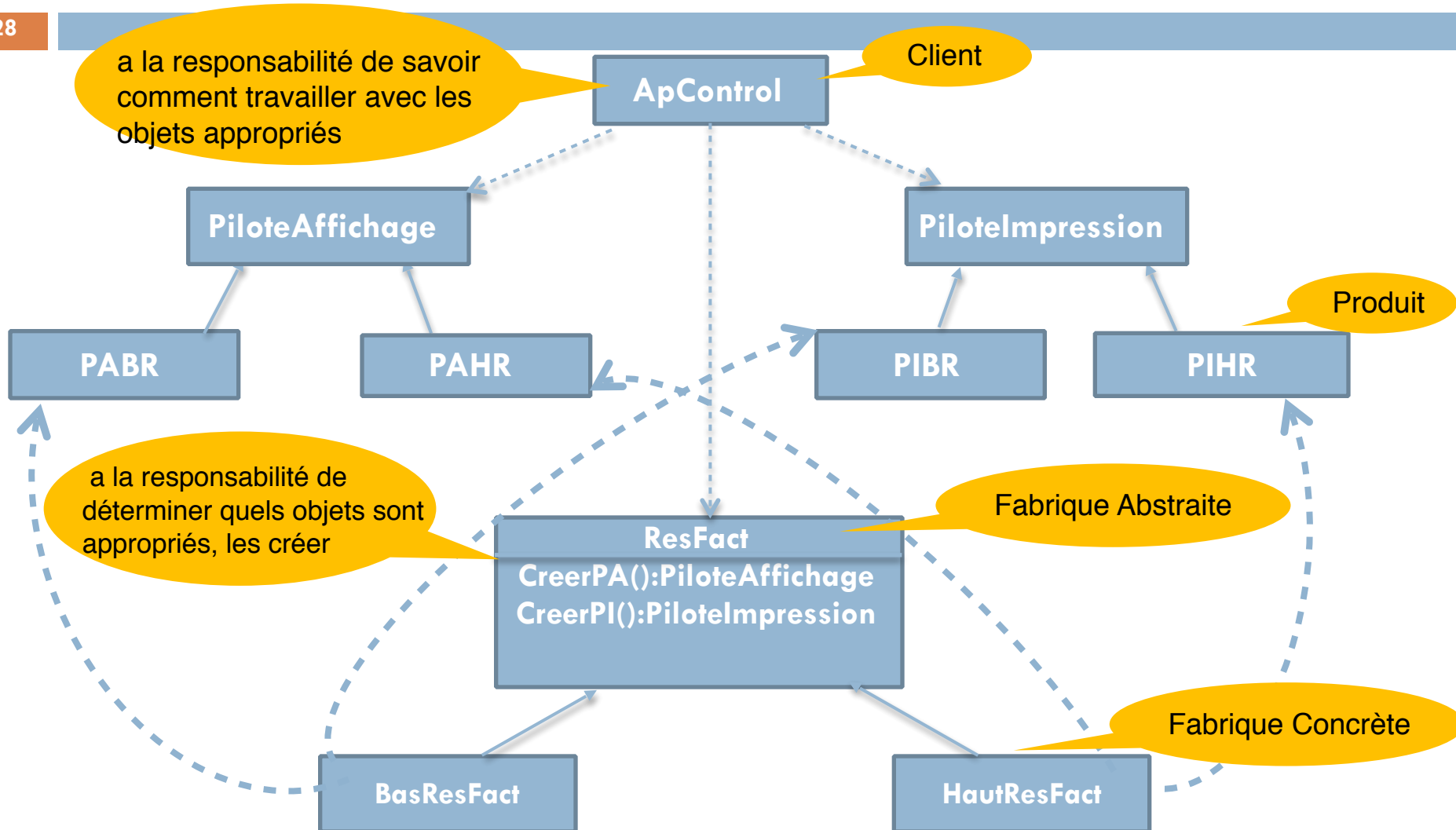
28





# Encapsuler la création d'objet

28



Le pattern Fabrique Abstraite fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes

# Encapsuler la création d'objet

29

```
abstract class ResFact {
public PiloteAffichage CreerPA();
public PiloteImpression CreerPI();
}

class BasResFact extends ResFactory {
public PiloteAffichage CreerPA() {
return new PABR(); }

public PiloteImpression CreerPI() {
return new PIBR();
}

}

class HautResFact extends
ResFactory {
public PiloteAffichage CreerPA() {
return new PAHR();
}

public PiloteImpression CreerPI() {
return new PIHR();
}

}
```

# Abstract Factory

30

- **Intention**
  - ▣ Fournir une interface pour créer des familles d'objets dépendants ou associés sans connaître leur classe réelle
  - ▣ Fabriquer des fabriques.
- *Utilisations connues*
  - ▣ Fabriquer des widgets qui ont tous le même look&feel
- *Synonymes :*
  - ▣ *Kit, Fabrique abstraite, Usine abstraite*
- *Patrons en relation*
  - ▣ Factory Method, Prototype, Singleton.

# Abstract Factory

31

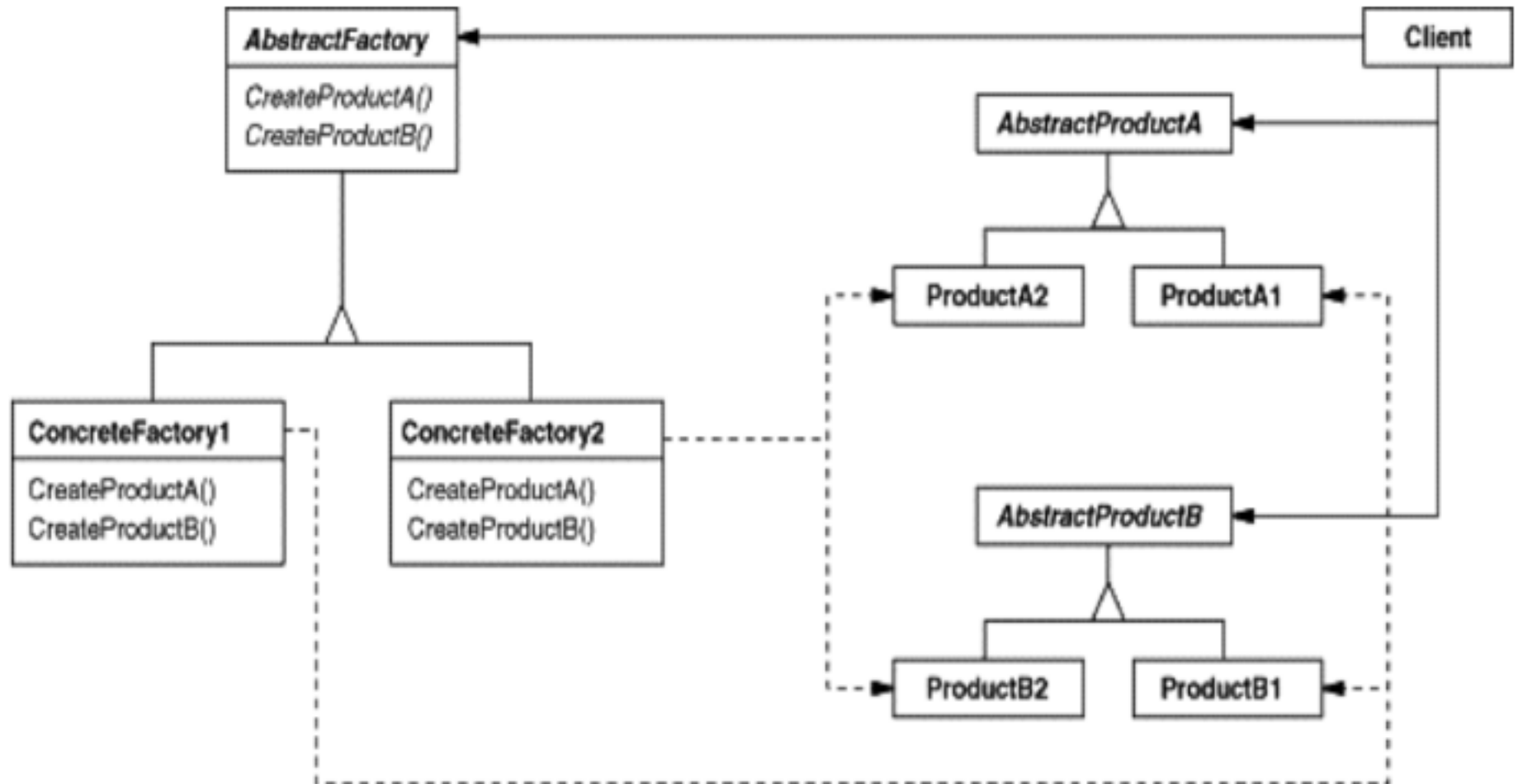
## □ Problème

- ▣ Un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
- ▣ Un système repose sur un produit d'une famille de produits
- ▣ Une famille de produits doit être utilisée ensemble, pour renforcer cette contrainte
- ▣ On veut définir une interface unique à une famille de produits concrets

# Abstract Factory

32

## □ Solution



# Abstract Factory

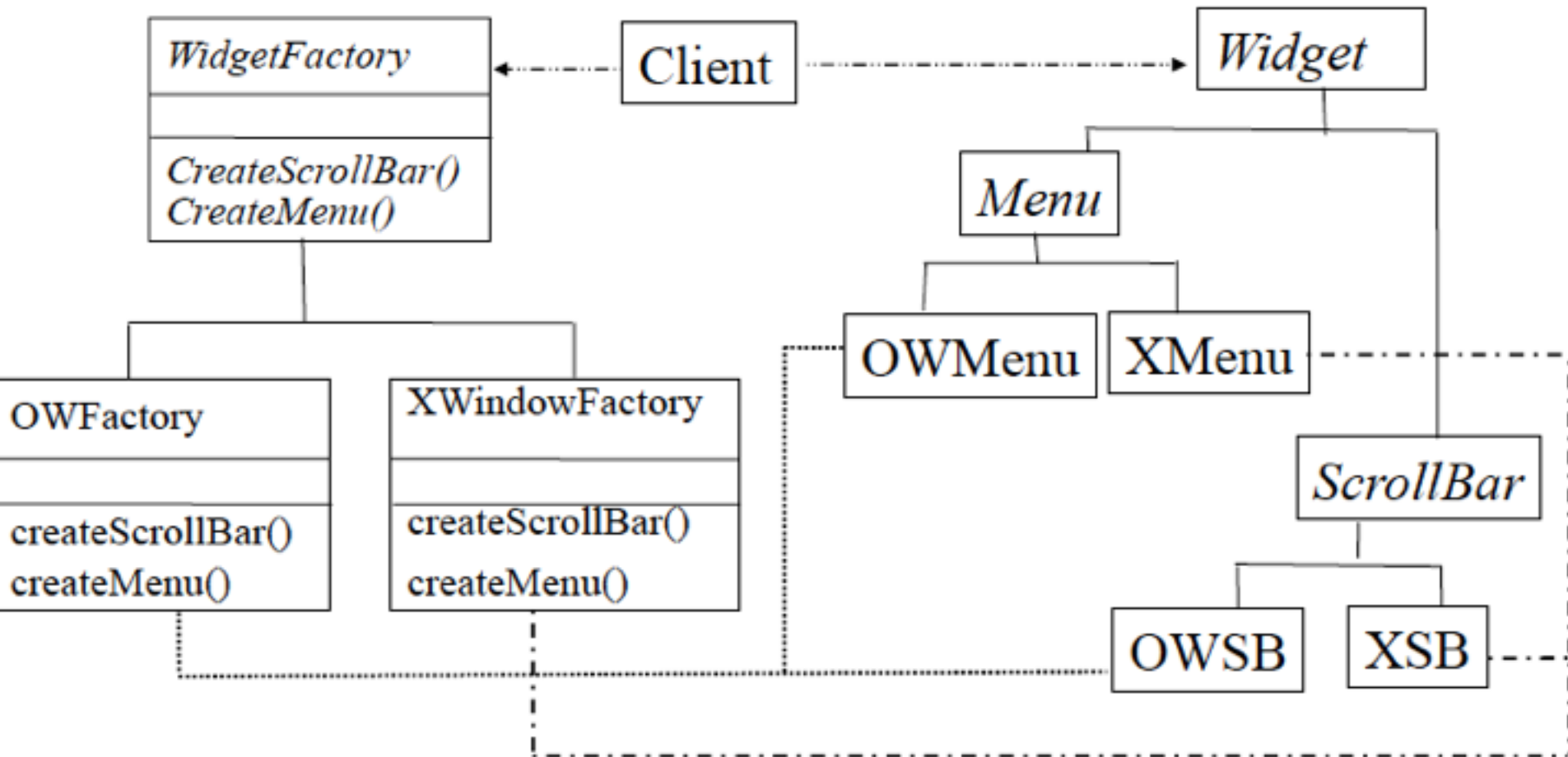
33

- 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*

# Exemple

34

- Changer l'apparence des éléments d'interface graphique



# Conséquences

35

- Isolation des classes concrètes (seules les classes abstraites sont connues)
- Échange facile des familles de produit
- Encouragement de la cohérence entre les produits
- Mais...prise en compte difficile de nouvelles formes de produit



# Implantation

36

- 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

# Factory Method vs Abstract Factory

37

<i>Factory Method</i>	<i>Abstract Factory</i>
s'appuie sur l'héritage : la création des objets est déléguée aux sous-classes qui implémentent la méthode de fabrication pour créer des objets.	s'appuie sur la composition : la création des objets est implémentée dans les méthodes exposées dans l'interface fabrique.
L'intention de Fabrication est de permettre à une classe de <b>déléguer l'instanciation à ses sous-classes.</b>	<b>L'intention de Fabrique Abstraite est de créer des familles d'objets apparentés sans avoir à dépendre de leurs classes concrètes.</b>
Fournit une interface abstraite pour créer UN produit.	Fournit une interface abstraite pour créer une famille de produits
Chaque sous-classe choisit quelle classe concrète instancier	Chaque sous-classe concrète crée une famille de produits

- Dans une Fabrique Abstraite, les méthodes pour créer les produits sont souvent implémentées à l'aide d'une Fabrication...

# Exercice

38

- On desire implementer un moteur de jeu générique dans lequel deux elements peuvent varier : les adversaires (on prévoit des obstacles Puzzle et NastyVillain) et les joueurs (on prévoit initialement deux types de joueurs Kitty et KungFuGuy). Cependant les divers jeux que l'on veut créer visent des publics différents : Kitty vs. Puzzle d'un côté, et KungFuGuy vs. NastyVillain de l'autre : pas question de faire jouer une instance de Kitty contre une de NastyVillain.
- Questions
  - Utilisez le patron de la fabrique abstraite pour permettre la création et le lancement de différents jeux a partir du moteur générique.
  - Comment ajouter un jeu Fairies vs. Gnomes dans votre moteur de jeu ?

# PATRONS DE CRÉATION

Singleton

# Motivation

40

- Une fabrique de chocolat moderne a un bouilleur assisté par ordinateur. La tâche du bouilleur consiste à contenir un mélange de chocolat et de lait, à le porter à ébullition puis à le transmettre à la phase suivante où il est transformé en plaquettes de chocolat.
- Pour éviter les catastrophes la classe contrôleur du bouilleur industriel doit éviter de vider le bouilleur (deux mille litres de mélange) qui n'a pas bouilli, de remplir un bouilleur déjà plein ou de faire bouillir un bouilleur vide.

# Besoin d'une instance unique

41

```
public class BouilleurChocolat {
    private boolean vide;
    private boolean bouilli;
    public BouilleurChocolat() {
        vide = true;
        bouilli = false;
    }

    public void remplir() {
        if (estVide()) {
            vide = false;
            bouilli = false;
            // remplir le bouilleur du mélange lait/chocolat
        }
    }

    public void vider() {
        if (!estVide() && estBouilli()) {
            // vider le mélange
            vide = true;
        }
    }
}
```

```
public void bouillir() {
    if (!estVide() && !estBouilli()) {
        // porter le contenu à ébullition
        bouilli = true;
    }
}

public boolean estVide() {
    return vide;
}

public boolean estBouilli() {
    return bouilli;
}
}
```

- Que pourrait-il se passer si on créait plusieurs instances de BouilleurChocolat dans une application ?
  - => Il faut s'assurer qu'une seule et unique instance est créée.
  - Une variable globale? Il est toujours possible d'avoir plusieurs instances!
  - Rendre la classe elle même responsable de la création d'une seule instance

# Création d'une instance unique

42

```
public class BouilleurChocolat {  
    private boolean vide;  
    private boolean bouilli;  
    private static BouilleurChocolat uniqueInstance;  
  
    public static BouilleurChocolat getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new BouilleurChocolat ();  
        }  
        return uniqueInstance;  
    }  
    ....  
}
```

# Création d'une instance unique

42

```
public class BouilleurChocolat {  
    private boolean vide;  
    private boolean bouilli;  
    private static BouilleurChocolat uniqueInstance;  
  
    public static BouilleurChocolat getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new BouilleurChocolat ();  
        }  
        return uniqueInstance;  
    }  
    ....  
}
```

Le Pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.



# Singleton

43

- *Intention*
  - ▣ S'assurer qu'une classe a une seule instance, et fournir un point d'accès global à celle-ci.
  - ▣ Variable globale « améliorée »
- *Utilisations connus*
  - ▣ Un seul window manager, un seul point d'accès à une base de donnée, DefaultToolkit en AWT/Java, etc.
- *Patterns associés*
  - ▣ *Abstract Factory, Builder, Prototype*

# Singleton

44

- Problème
  - ▣ Il est souvent important pour une classe de n'avoir qu'une instance qui doit être accessible de manière connue (facilement accessible).
  - ▣ Lorsque l'instance unique doit être extensible par héritage, et que les clients doivent pouvoir utiliser cette instance étendue sans modifier leur code
  - ▣ Une variable globale n'est pas suffisamment flexible

# Singleton

45

## □ Solution:

- ▣ Assurer une instance unique en cachant le mécanisme de création (constructeur privé en Java)
- ▣ Garder une référence pour l'instance unique (attribut statique privé)
- ▣ Créer un point d'accès publique (une méthode qui retourne l'instance unique)

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

Singleton
-instance: Singleton
-Singleton() +getInstance(): Singleton

# Conséquences

46

- Accès contrôlé
- Pas de variable globale
- Permet la spécialisation des opérations et de la représentation
- Permet un nombre variable d'instances (variante de Singleton)
- Plus flexible que les méthodes de classe

# Implémentation

47

- **Assurer l'unicité**
  - ▣ Remarque : En Java la méthode « Clone » de la classe Object permet de cloner l'instance créée. Pour garantir l'unicité, une solution est de redéfinir cette méthode de sorte qu'elle retourne « this » (l'instance unique créer).
- Sous-classer la classe Singleton : L'attribut qui réfère à l'instance du Singleton doit être initialiser avec une instance de la sous-classe
  - ▣ Déterminer quel singleton utilisé (instance de sous-classe) dans la méthode **getInstance()**
  - ▣ Utiliser un registre de singletons (nom singleton, instance)

# Verrouillage à double tour

48

- Application « multithread »
  - Si le singleton n'a pas d'état, cela ne devrait pas être un problème
  - Préférer une instance créée au démarrage (lors de la déclaration de l'attribut) à une instance créée à la demande
- Verrouillage à double tour : Variante du pattern Singleton
  - Effectuer une synchronisation après le test de la valeur de l'instance et avant sa création

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton () {...}  
    private synchronized static void effectuerSync() {  
        if (instance==null) instance= new Singleton ();  
    }  
    public static Singleton getInstance() {  
        if (instance==null) effectuerSync();  
        return instance;  
    }  
}
```

□

# Exercice

49

- Proposez une implémentation d'une application qui veut créer un seul point au pilote de la carte son, en se basant sur le patron singleton.

# PATRONS DE CRÉATION

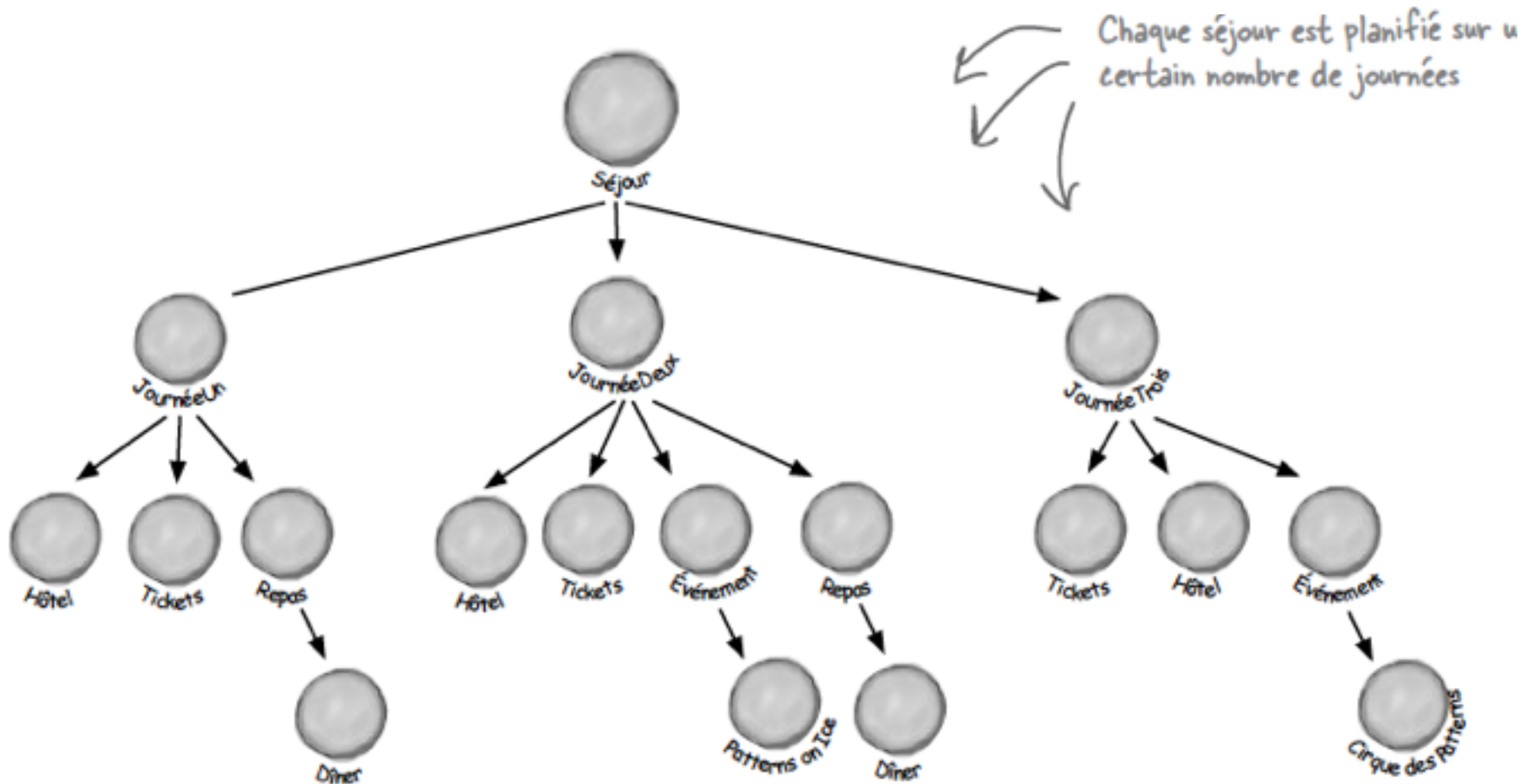
Builder



# Construction d'un produit par étape

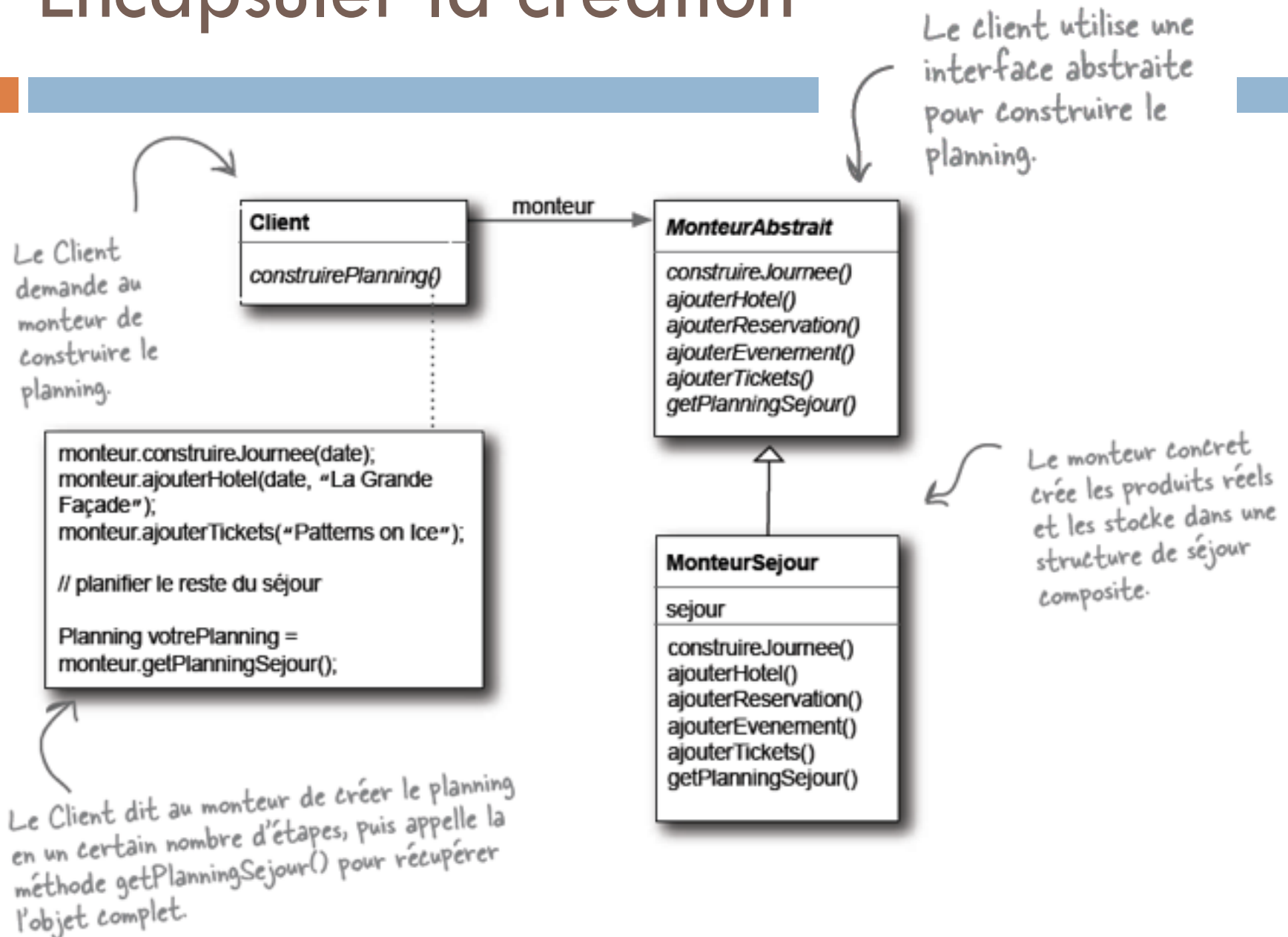
51

- Logiciel de planification de séjour: jour1, jour2, etc.



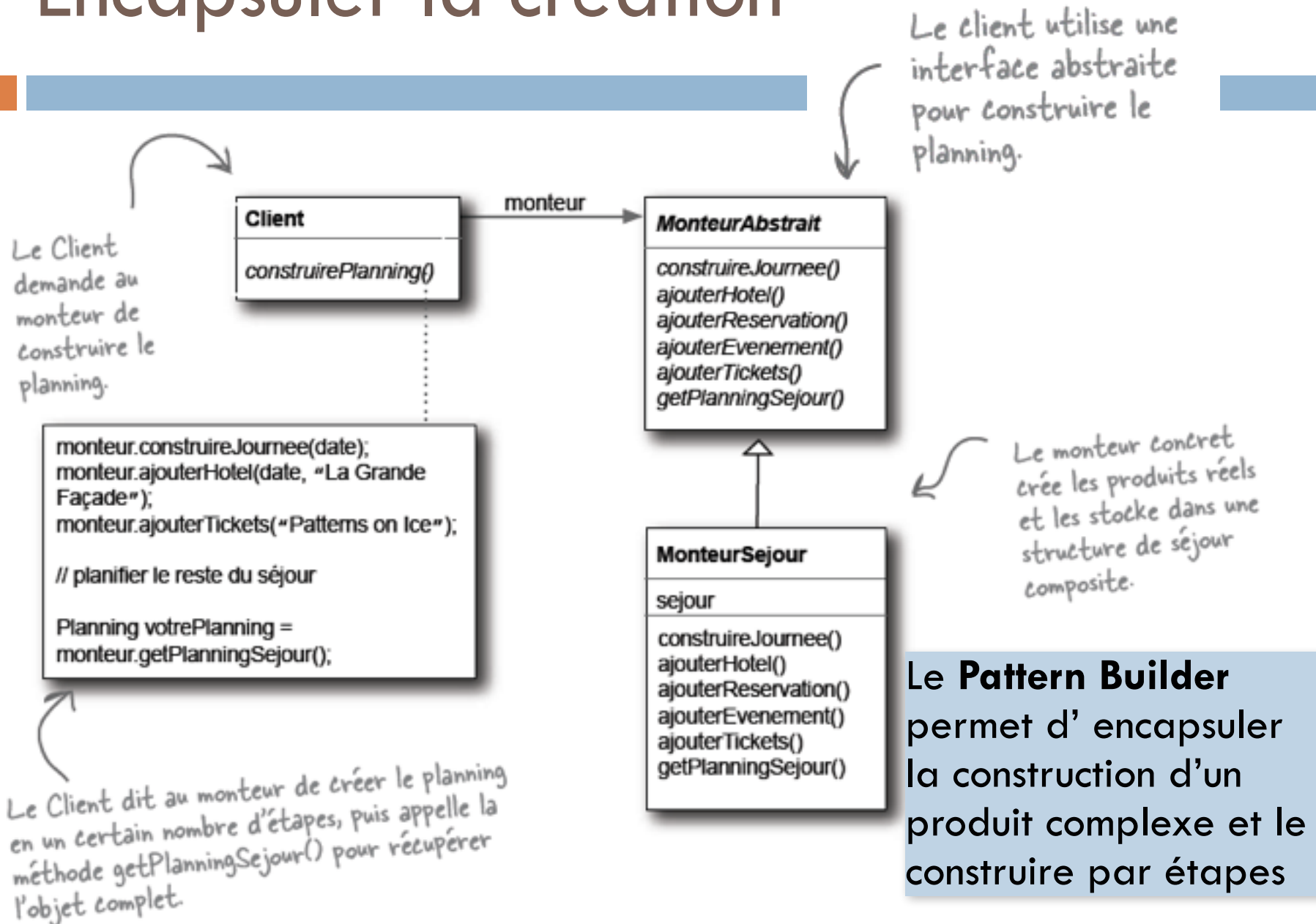
# Encapsuler la création

52



# Encapsuler la création

52



# Pattern Builder

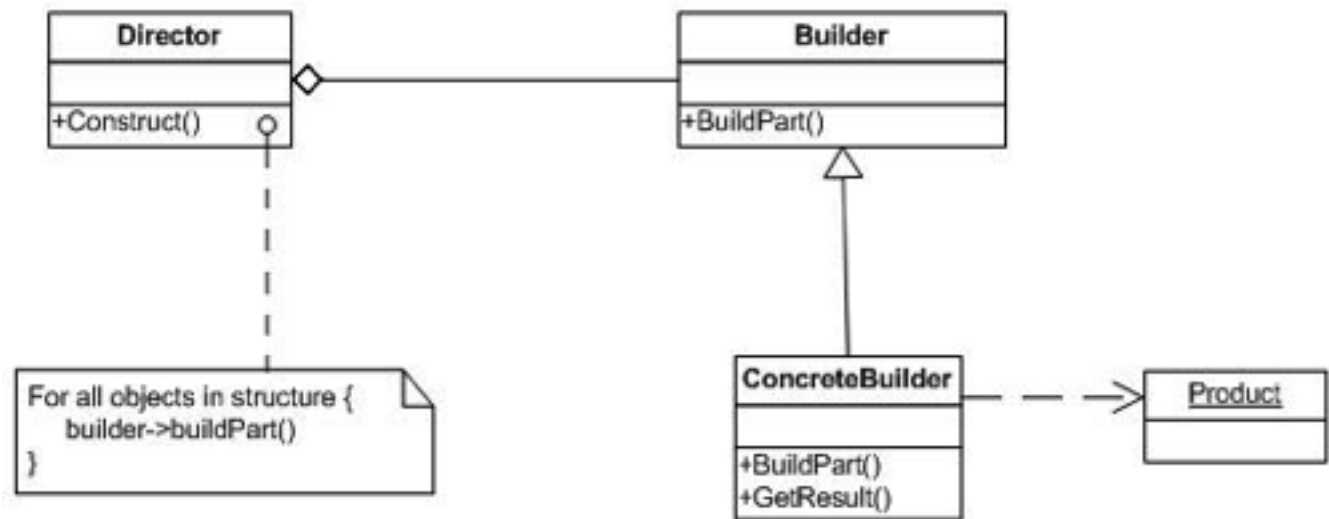
53

- Intention
  - ▣ Séparer la construction d'un objet complexe de sa représentation de façon à pouvoir construire différentes représentations à partir d'un même processus de création.
- *Utilisations connues*
  - ▣ Utilisé par les parser, création de graphe, création d'un menu..
- *Synonymes*
  - ▣ *Monteur, Fabrique concrète*
- *Patrons associés*
  - ▣ *Un Builder est en général un **Singleton***
  - ▣ ***Abstract factory** est souvent similaire mais la démarche est différente*
  - ▣ *Le produit généré est souvent un **Composite***

# Builder

54

- Problème
  - ▣ l'algorithme pour créer un objet doit être indépendant des parties qui le compose et de la façon de les assembler
  - ▣ le processus de construction permet différentes représentations de l'objet construit
- Solution



# Builder: Participants

55

- **Builder**
  - ▣ Spécifie une interface abstraite afin de créer les parties d'un objet Produit.
- **BuilderConcret**
  - ▣ Construit et assemble les pièces du produit en implémentant l'interface Builder.
  - ▣ Définit et garde une trace de la représentation qu'il crée.
  - ▣ Fournit une interface pour retrouver le produit
- **Director**
  - ▣ Construit un objet en utilisant l'interface Builder.
- **Produit**
  - ▣ Représente l'objet complexe en construction. BuilderConcret construit la représentation interne du produit et définit le processus par lequel il est assemblé.
  - ▣ Comprend les classes qui définissent les éléments constitutifs, y compris les interfaces pour l'assemblage des pièces en un résultat final.

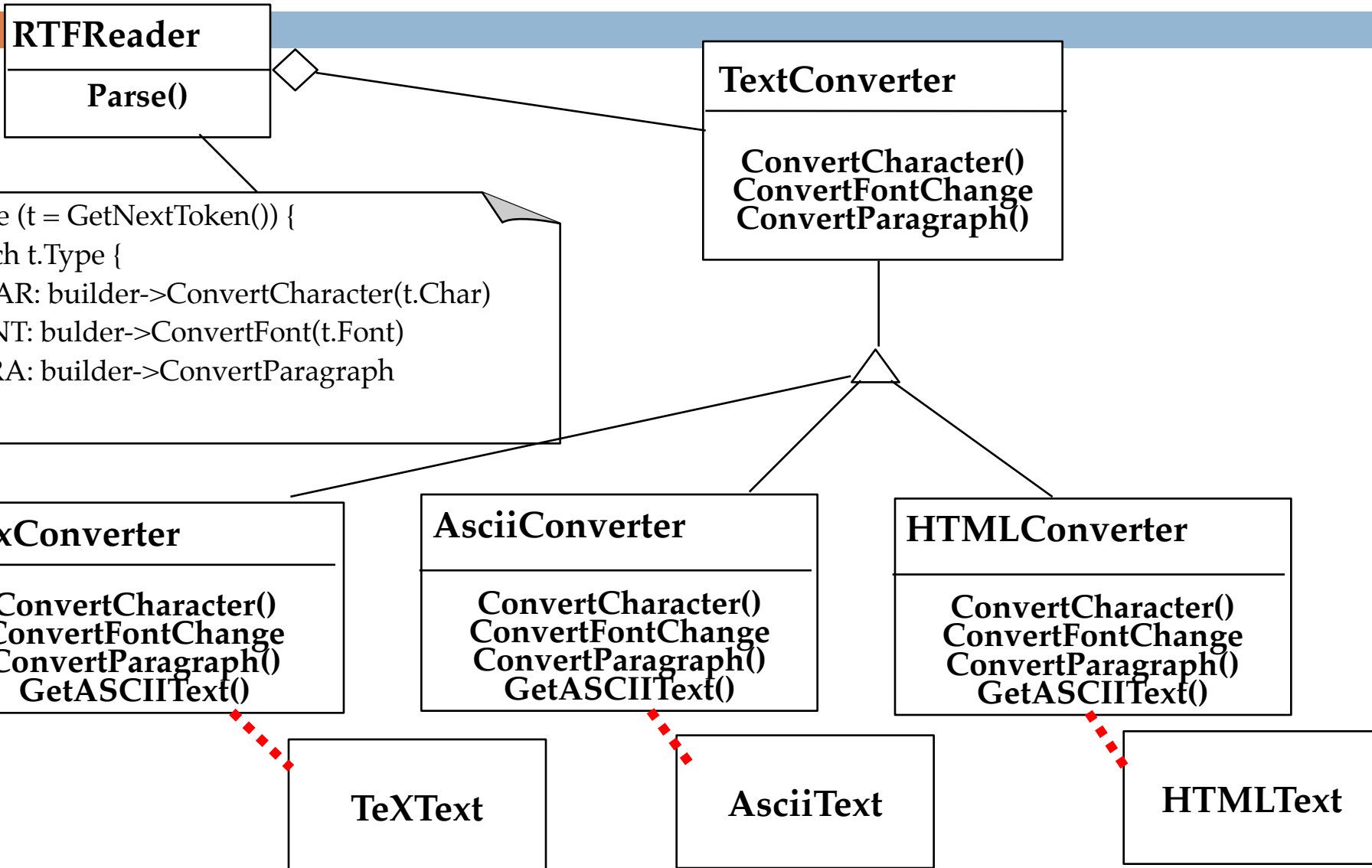
# Exemple réel

56

- 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.
  - La concaténation de chaînes de caractères au moyen de l'opérateur `" + "` est traduite via `StringBuilder`

# Builder : Example

57





# Builder : Exemple

58

## //Abstract Builder

```
class abstract
TextConverter{
    abstract
    convertCharacter(char c);
    abstract
    convertParagraph();
}
```

## // Product

```
class ASCIIText{
    public void append(char c){ //
Implement the code here }}
```

class

void

void

## //Concrete Builder

```
class ASCIIConverter extends
TextConverter{
    ASCIIText asciiTextObj;//resulting
    product
```

```
/*converts a character to target
representation and appends to the
resulting*/
```

```
object void convertCharacter(char c)
{
    char asciiChar = new
Character(c).charValue();
    //gets the ascii character
    asciiTextObj.append(asciiChar);
}
void convertParagraph(){
    ASCIIText getResult(){
        return asciiTextObj;
    }
}
```

# Builder : Exemple

59

## //Director

```
class RTFReader{
    private static final char EOF='0';
    //Delimiter for End of File
    final char CHAR='c';
    final char PARA='p';
    char t;
    TextConverter builder;
    RTFReader(TextConverter obj){
        builder=obj;
    }
    void parseRTF(Document doc){
        while ((t=doc.getNextToken())!=
EOF){ switch (t){
            case CHAR:
builder.convertCharacter(t);
            case PARA:
builder.convertParagraph();
        }
    }
}
```

## //Client

```
public class Client{
    void createASCIIText(Document doc)
    {
        ASCIIConverter  asciiBuilder  =
new ASCIIConverter();
        RTFReader  rtfReader  =  new
RTFReader(asciiBuilder);
        rtfReader.parseRTF(doc);
        ASCIIText      asciiText      =
asciiBuilder.getResult();
    }
    public static void main(String
args[]){
        Client client=new Client();
        Document doc=new Document();
        client.createASCIIText(doc);
        system.out.println("This is an
example of Builder Pattern");
    }
}
```

# Conséquences

60

- Variation possible de la représentation interne d'un produit
  - ▣ L'interface `AbstractBuilder` permet de cacher la représentation et la structure interne du produit ainsi que la manière dont il est assemblé. Pour changer la représentation interne du produit il suffit de définir un nouveau type de Builder.
- Isolation du code de construction et du code de représentation du reste de l'application (améliore la modularité)
  - ▣ Chaque `BuilderConcret` contient tout le code pour créer et assembler un certain type de produit. Le code n'est écrit qu'une fois ; puis différents Directeurs peuvent le réutiliser afin de construire des Produits dérivants d'un même ensemble de pièces.
- Meilleur contrôle du processus de construction
  - ▣ Produit construit pas à pas sous la supervision du directeur
  - ▣ Accès au produit une fois le processus terminé

# Implémentation

61

- La classe abstraite **AbstractBuilder** définit une opération pour chaque composant que le Directeur pourrait lui demander de créer.
  - Par défaut, ces opérations ne font rien.
  - Une classe BuilderConcret surcharge les opérations concernant les composants dont la création l'intéresse.
- L'interface **AbstractBuilder** doit être assez générale pour permettre la construction de produits pour tout type de BuilderConcret.
  - Habituellement dans le processus de construction et d'assemblage, les résultats des demandes de construction sont simplement ajoutés les uns aux autres.
  - Parfois, vous pourriez avoir besoin d'accéder à des parties du produit construit plus tôt. Les structures arborescentes sont un exemple. Dans ce cas, le Builder va renvoyer des noeuds enfants au Directeur, qui va ensuite les retourner au Builder pour construire les noeuds parents.

# Implémentation

62

- **Pourquoi ne pas rendre abstraites les classes de produits également ?**
  - Dans la plupart des cas, les produits construits par les BuilderConcret diffèrent tellement dans leur représentation qu'il y a très peu de gain à donner aux différents produits une classe parente commune.
  - Dans l'exemple du RTF, les objets TexteASCII et TexteWidget sont trop différents pour avoir une interface commune, et n'en ont pas besoin.
  - Le client est en position de savoir quelle sous-classe de Builder est utilisée et peut gérer ses produits dans ce sens (configurer le Directeur avec le bon BuilderConcret).
- **Par défaut, les méthodes du Builder sont vides.**
  - En C++, les méthodes de construction ne sont intentionnellement pas déclarées comme de pures fonctions membres virtuelles. Au lieu de cela, elles sont définies comme des méthodes vide, laissant le client surcharger uniquement celles qui l'intéressent.

# PATRONS DE CRÉATION

Prototype

# Motivation: Editeur de music

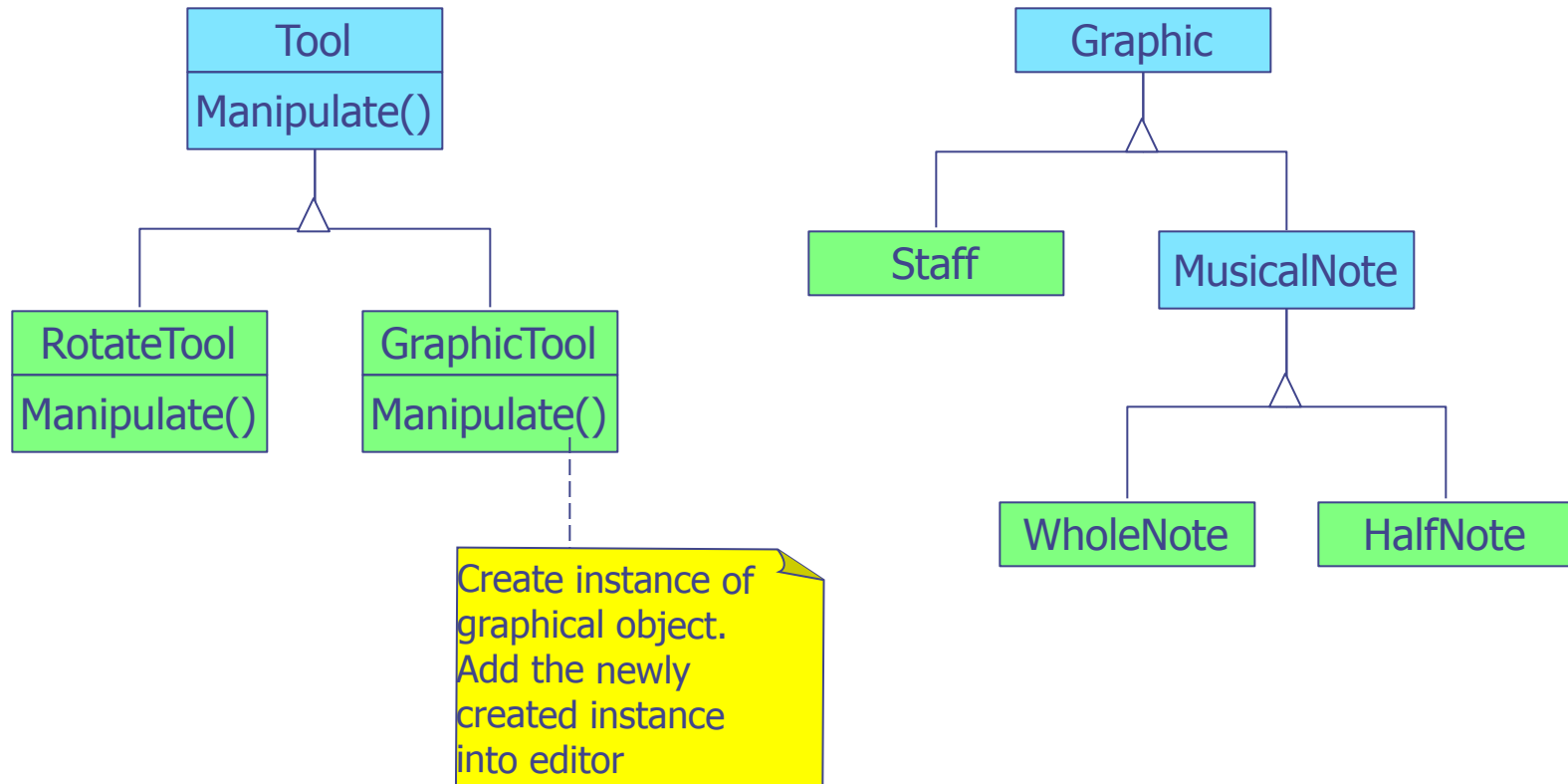
64

- Vous pourriez construire un éditeur de partitions musicales par la personnalisation d'un framework général d'éditeurs graphiques
  - ajouter de nouveaux objets qui représentent des notes, bâtons, etc.
- Le framework de l'éditeur peut avoir une palette d'outils pour ajouter ces objets musicaux.
- La palette pourrait inclure des outils de sélection, de déplacement, et de manipulation de toute autre objet musical.
  - Les utilisateurs pourront cliquer sur l'outil de note et l'utiliser pour l'ajouter à la partition
  - Ils peuvent aussi utiliser l'outil de déplacement pour déplacer une note vers le haut ou vers le bas ce qui modifie la partition.

# Motivation: Editeur de music

65

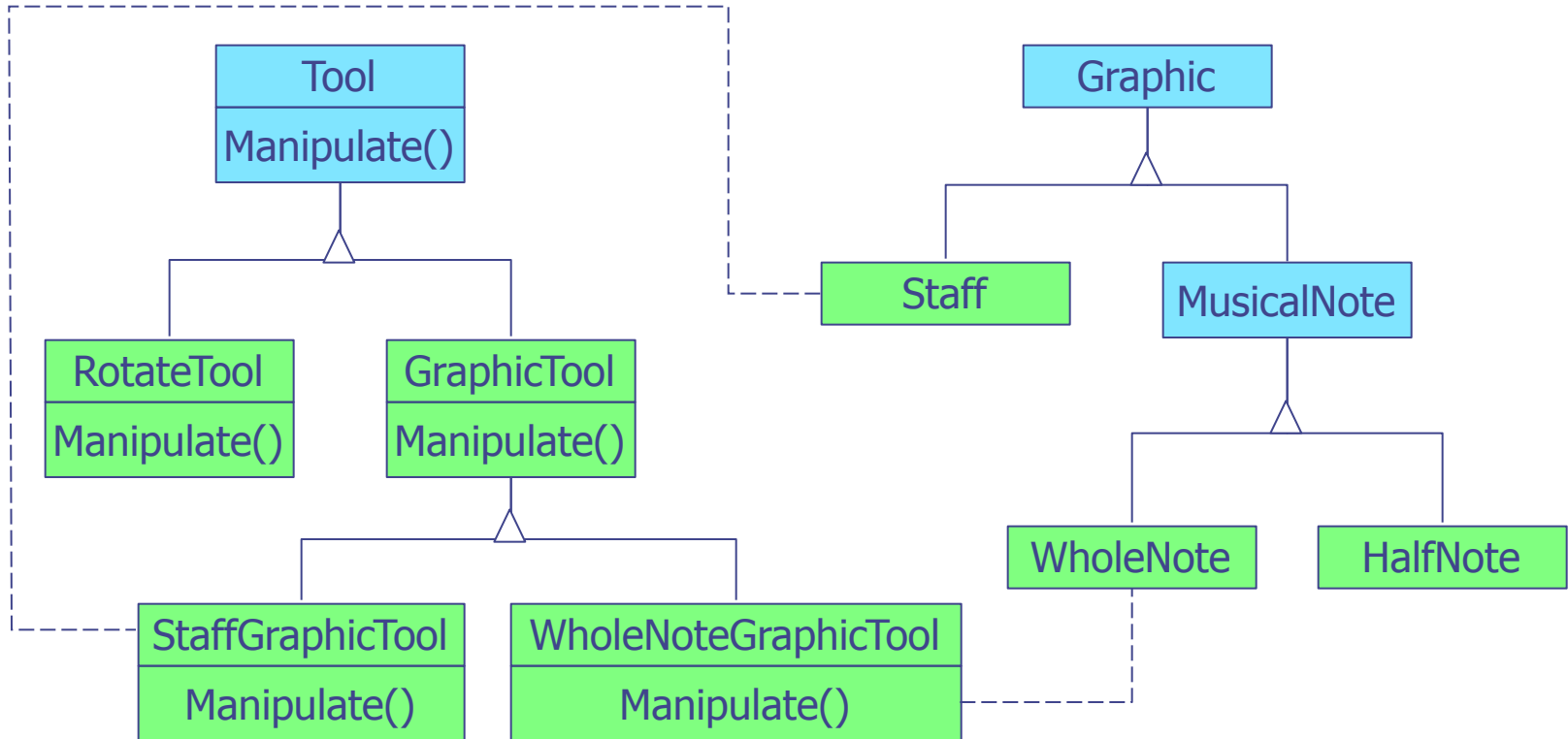
- La classe GraphicTool appartient au Framework, tandis que les classes graphiques sont spécifiques à l'application, de sorte que la classe GraphicTool ne sait pas comment créer des objets graphiques et fonctionne sur eux.





# Solution 1

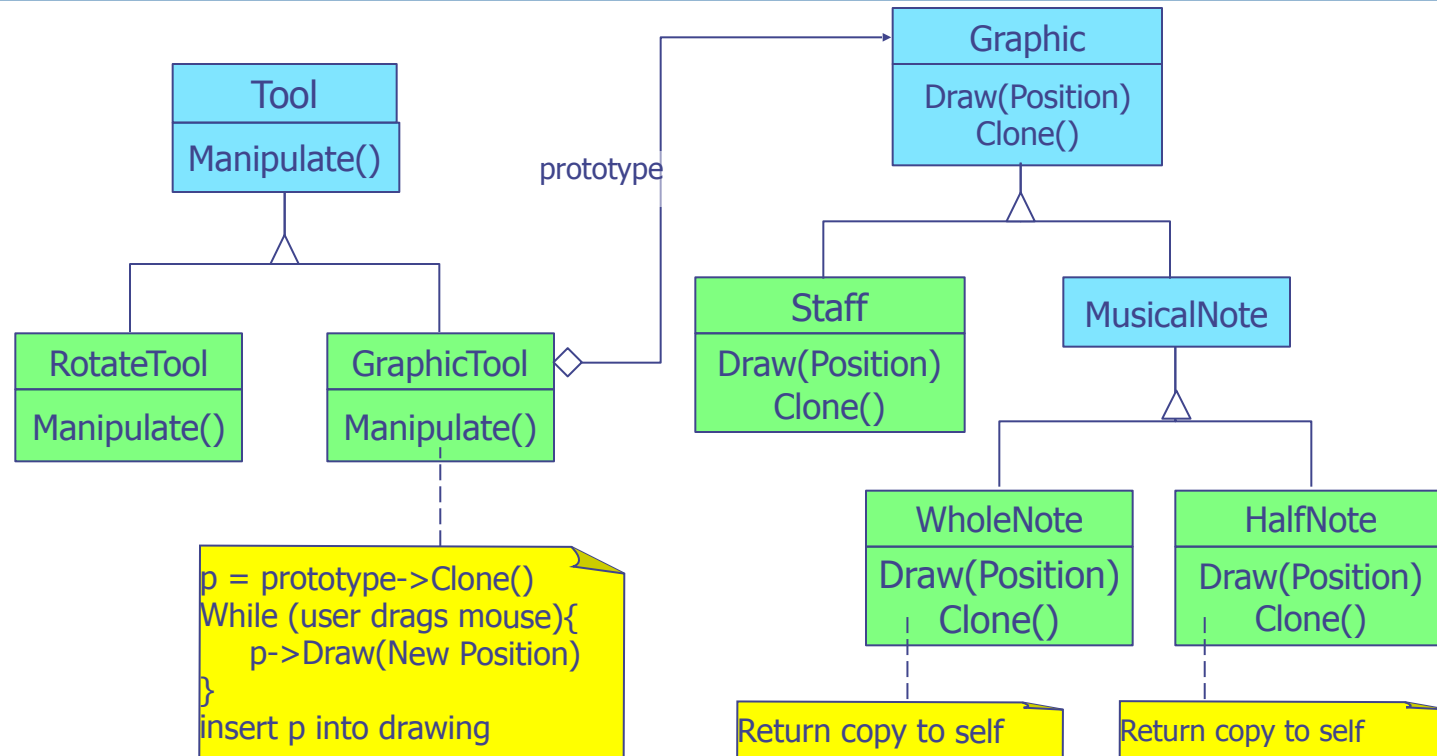
66



- Cette approche va créer plusieurs sous-classes qui diffèrent uniquement dans l'objet musical qu'elle instancie

# Solution 2

67



- Pour créer un nouvel objet graphique, l'instance « prototype » de la classe graphique spécifique est passée comme paramètre au constructeur de la classe **GraphicTool**.
- La classe **GraphicTool** peut alors utiliser ce prototype pour cloner un nouvel objet et l'opérer.
- Si toutes les sous-classes graphiques prennent en charge une opération de clonage, alors la classe **GraphicTool** peut cloner n'importe quel type de graphique.

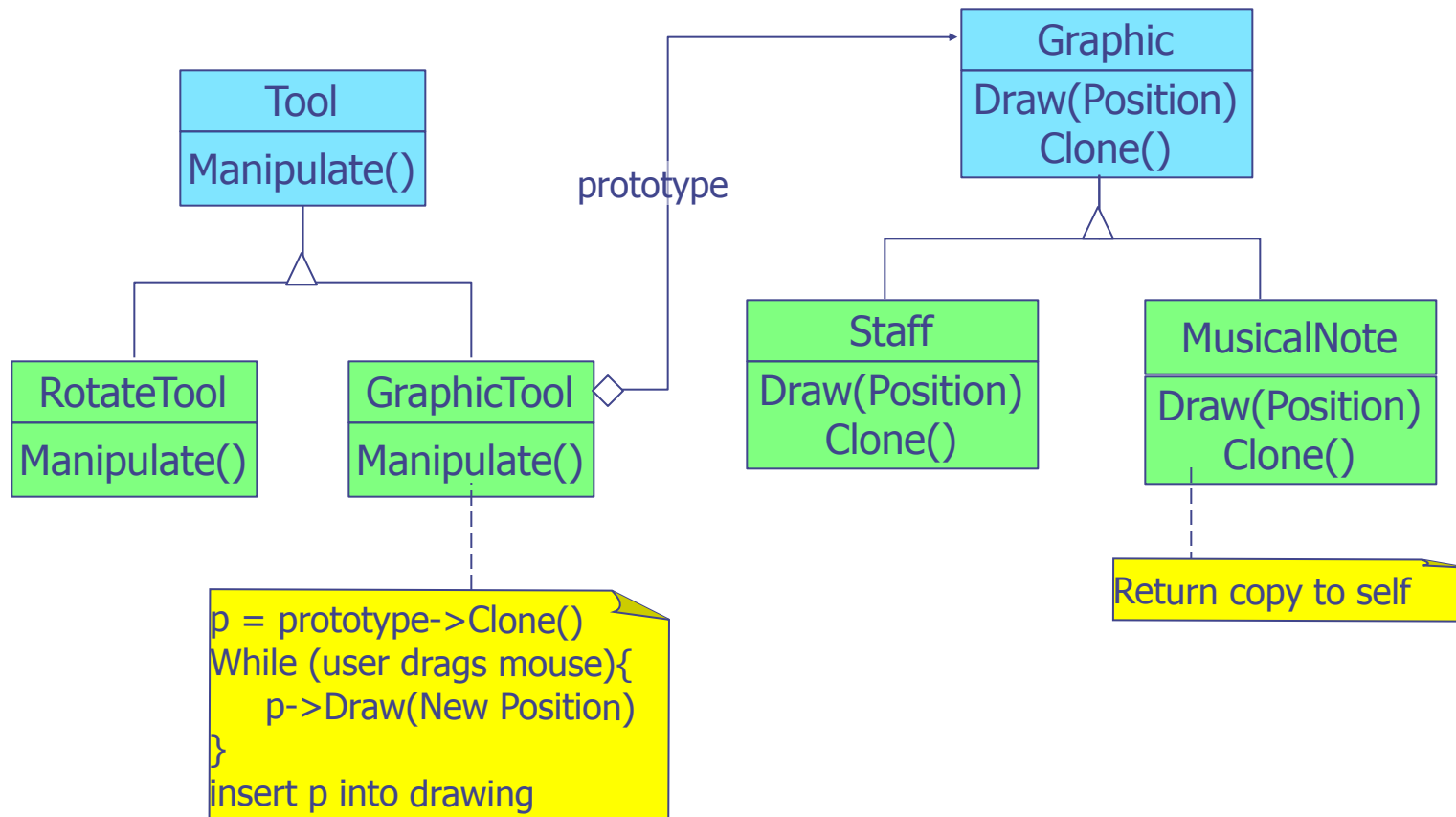
# Solution 2

68

- Le pattern Prototype modèle peut également être utilisé pour réduire le nombre de classes.
- Les classes WholeNote et Halfnote diffèrent selon les bitmaps et les durées.
  - Au lieu de définir une sous-classe pour chaque type de notes, nous pouvons rendre MusicalNote classe concrète, et les objets de note peut être instances de la même classe (MuscialNote) initialisé avec des réglages différents.
- Par conséquent, pour la classe GraphicTool, créer un nouvel objet WholeNote consiste à cloner un prototype de MuscialNote initialisée comme WholeNote

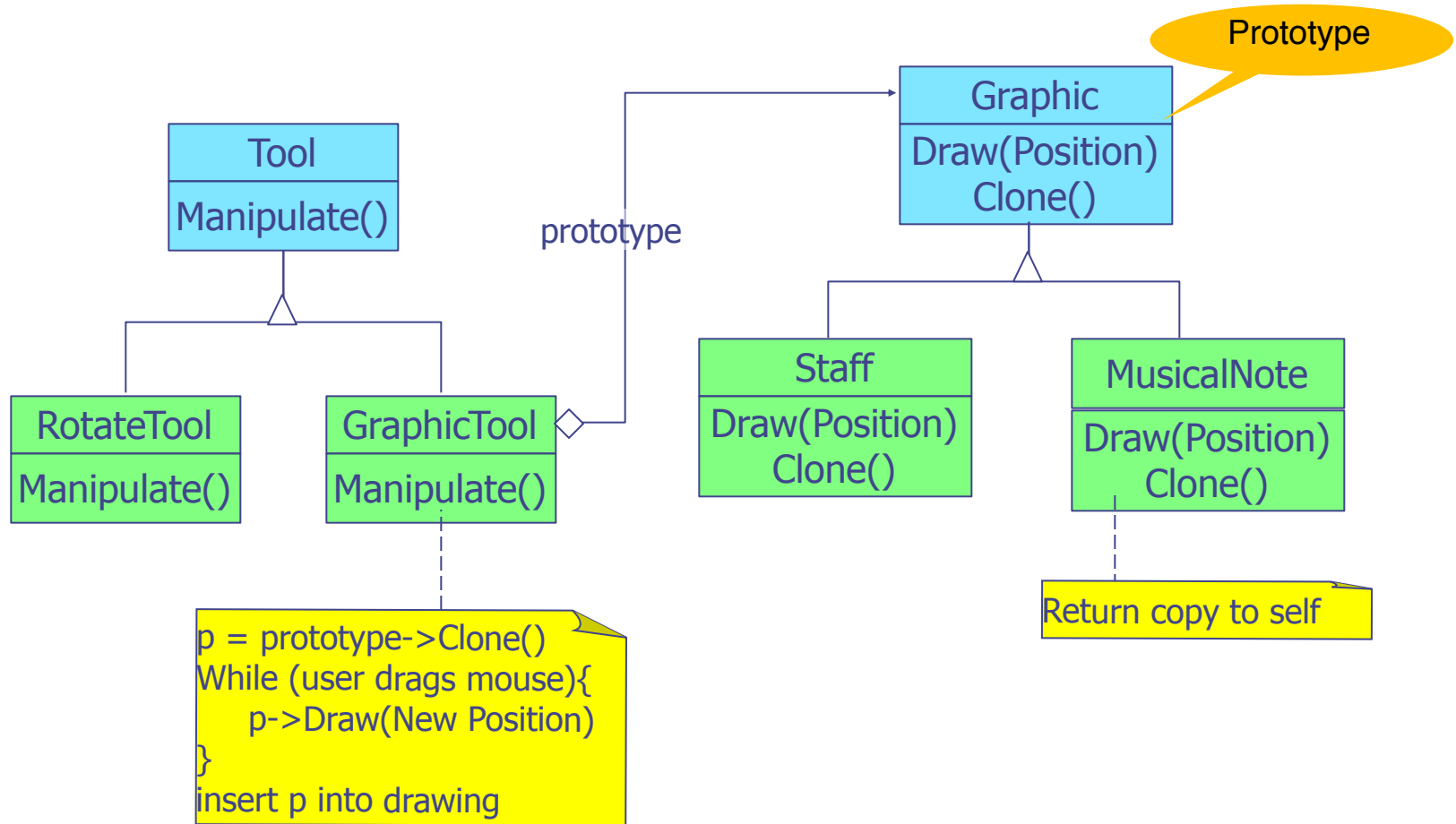
# Solution2

69



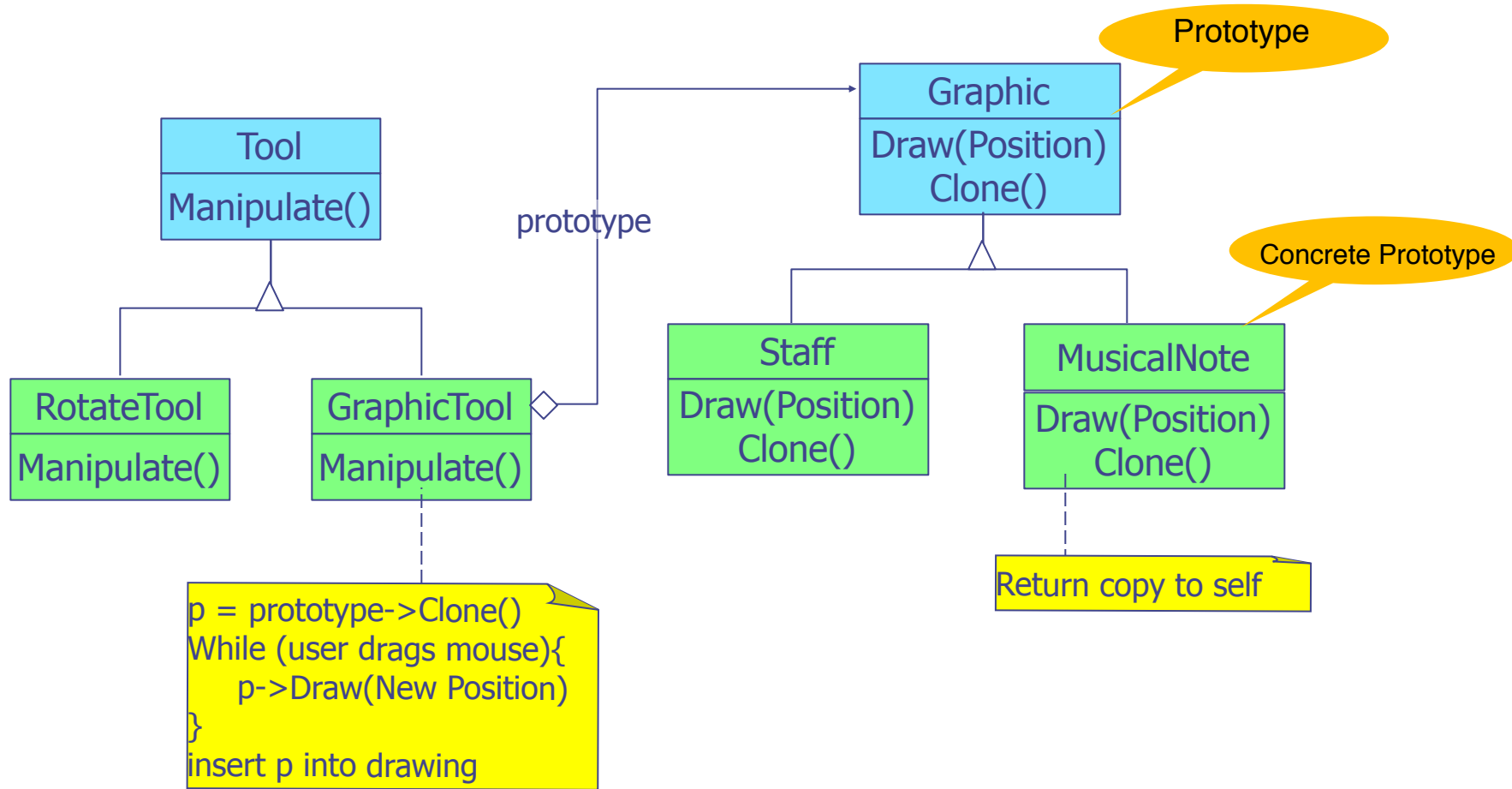
# Solution2

69



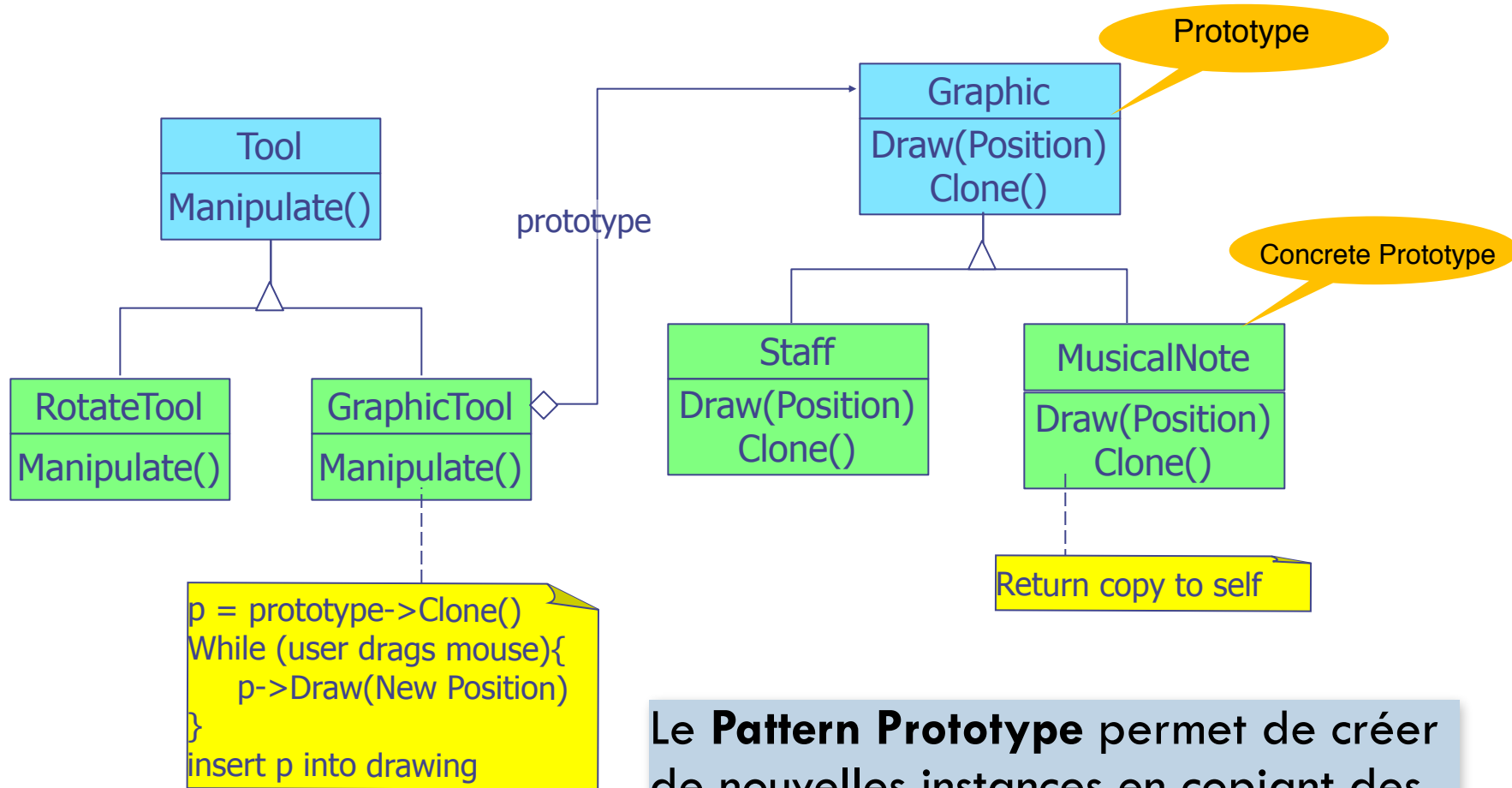
# Solution2

69



# Solution2

69



Le **Pattern Prototype** permet de créer de nouvelles instances en copiant des instances existantes.

# Prototype

70

- Intention
  - ▣ Spécifie le type des objets à créer à partir d'une instance (le prototype), et crée de nouveaux objets en copiant ce prototype (clonage).
- *Utilisations connues*
  - ▣ Prototypes de matériels informatiques, éditeurs (ex. UML, etc.), gestion des monstres dans les jeux vidéos, Simulation de la division cellulaire, ..
- *Synonymes :*
  - ▣ *clone, objet type*
- *Patrons en relation*
  - ▣ *Abstract Factory, Composite, Decorator*



# Prototype

71

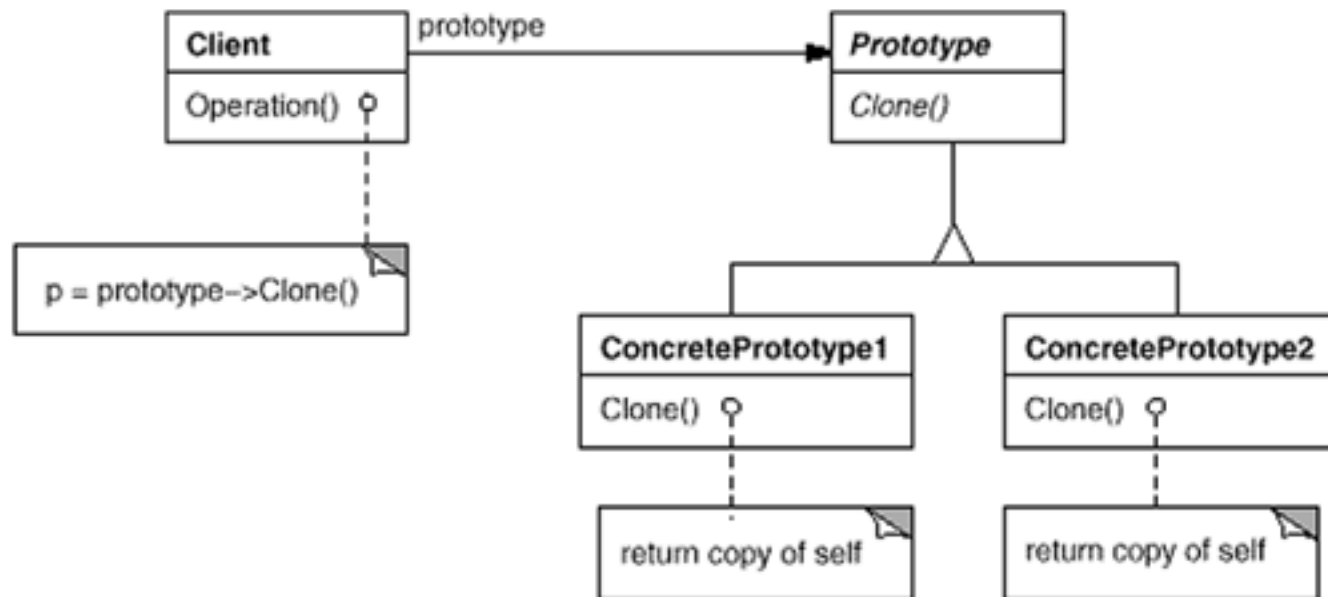
## □ Problème

- ▣ Si les instances d'une classe peuvent prendre un état parmi un ensemble de configurations (=prototypes). C'est souvent le cas pour des composites.
- ▣ Pour limiter le nombre de sous-classes, une par configuration type ( un nombre réduit d'états : un prototype par état)
- ▣ Pour éviter de construire une hiérarchie de “factories” parallèle à la (aux) hiérarchie(s) de produits
- ▣ Quand la classe à instancier n'est connue qu'à l'exécution
- ▣ Quand la création d'une instance d'une classe donnée est coûteuse ou compliquée
- ▣ Quand un système doit être indépendante de la façon dont ses produits sont créés, imposées, et représentés.
  - Un outil générique manipule certains objets spécifiques à un(e) domaine (application)
- ▣ Quand des types de produits (configurations) doivent pouvoir être créées (ou ajoutées) par l'utilisateur
  - Prototype offre d'une certaine façon des capacités de programmation à l'utilisateur lui-même (« programmation dynamique de classes »)

# Prototype

72

## □ Solution



# Participants

73

- **Prototype**
  - Déclare une interface pour se “cloner”
- **ConcretePrototype**
  - Implante une opération pour se “cloner”,
- **Client**
  - Crée un nouvel objet en demandant au prototype de “se cloner”

# Example

74

```
Ordinateur o1 = new Ordinateur("Dell",  
"clodion01");  
o1.addComposant(new UC(2.2, 100));  
o1.addComposant(new Ecran(19));  
o1.addComposant(new  
Clavier("azerty"));  
o1.addComposant(new HD(100));  
Ordinateur o2 = new Ordinateur("Dell",  
"clodion02");  
o2.addComposant(new UC(2.2, 100));  
o2.addComposant(new Ecran(19));  
o2.addComposant(new  
Clavier("azerty"));  
o2.addComposant(new HD(100));
```

```
//prototype!  
Ordinateur proto = new  
Ordinateur("Dell", "proto");  
proto.addComposant(new UC(2.2,  
100));  
proto.addComposant(new Ecran(19));  
proto.addComposant(new  
Clavier("azerty"));  
proto.addComposant(new HD(100));  
o1=proto.clone("clodion01");  
o2=proto.clone("clodion02");
```

# Conséquences

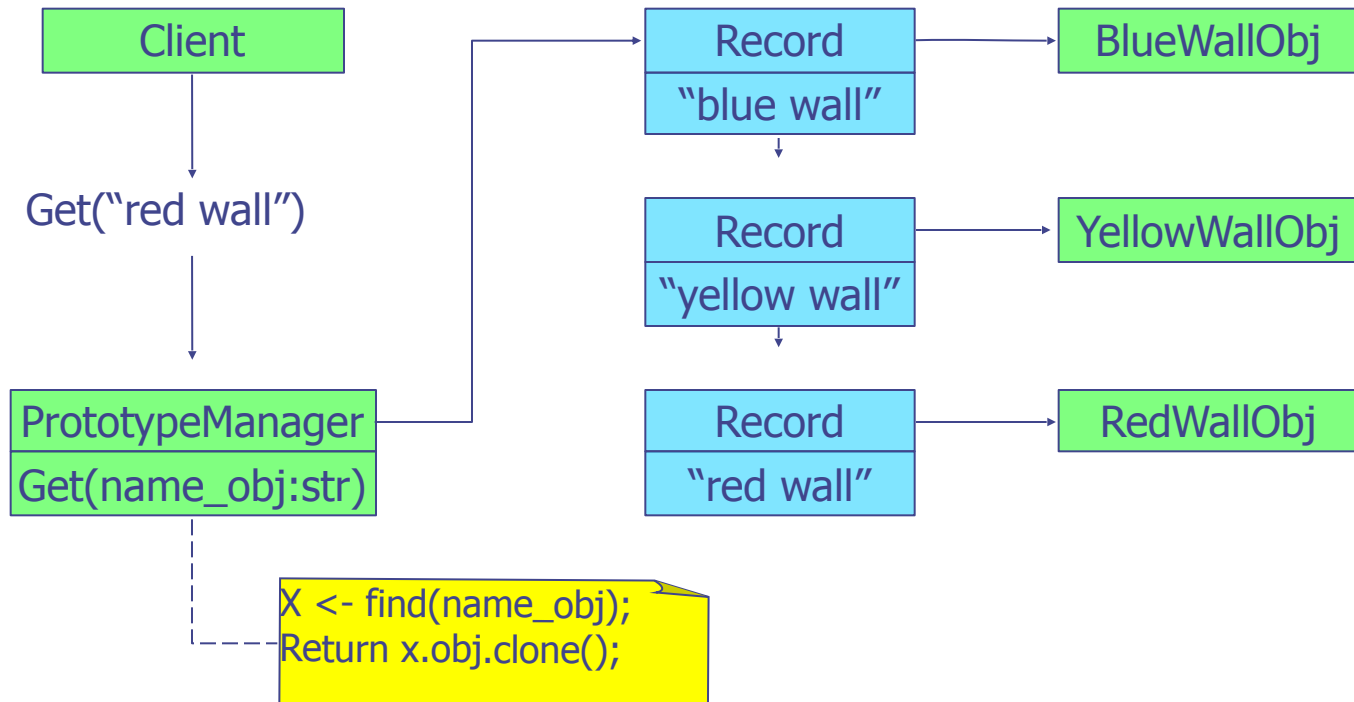
75

- Pouvoir ajouter et retirer des produits durant l'exécution,
- Pouvoir spécifier de nouveaux objets en variant les valeurs (plutôt que les types)
- Réduction dans les hiérarchies de classes (notamment les hiérarchies parallèles)
- Configurer une application avec des classes de façon dynamique
- Abstraction de la construction
  - 2 instances différentes 2 initialisations différentes
- Utile lorsque la phase d'initialisation est coûteuse
  - Plus rapide de recopier une instance

# Implémentation

76

- La partie la plus difficile du pattern Prototype est la mise en œuvre du fonctionnement de Clone
- Possibilité d'utiliser un prototype manager pour gérer différents prototypes jouant différents rôles,



# Récapitulatif

77

- On utilise le pattern Abstract Factory :
  - ▣ Quand on doit créer une famille d'objets qui se ressemblent.
  - ▣ Quand les objets d'une famille peuvent évoluer.
- On utilise le pattern Builder :
  - ▣ Quand on doit créer un objet complexe qui nécessite plusieurs étapes.
- On utilise le pattern Factory Method :
  - ▣ Quand on doit créer un objet concret en fonction d'un paramètre donné.
  - ▣ Quand on a besoin de faire évoluer ou ajouter des classes concrètes sans modifier la classe abstraite.
- On utilise le pattern Prototype :
  - ▣ Quand on doit créer de nouveaux objets à partir d'objets existants soit par copie, soit par clonage.
  - ▣ Quand la création d'un objet est plus coûteuse que de le copier.
- On utilise le pattern Singleton :
  - ▣ Quand un objet ne doit posséder qu'une seule et unique instance sous peine de générer une erreur.
  - ▣ Quand un objet créé deux fois est susceptible de générer une erreur.