

Patterns

Historique (Architecture)

- ◆ Notion de **patterns** est d'abord apparue en architecture
 - Christopher Alexander: « The Timeless Way of Building ». 1979.
 - Définition de Patterns pour:
 - Architecture des bâtiments
 - Conception des villes et de leur environnement
- ◆ Exemple:
 - pour se sentir à l'aise dans une pièce, il faut qu'il y ait des fauteuils confortables et que l'on puisse s'orienter face à la lumière
 - donc ... dans une pièce il faut avoir une fenêtre face à laquelle s'asseoir dans un fauteuil confortable

Historique (Logiciels)

- ♦ **Idée:** appliquer la notion de patterns à des architectures logicielles
- ♦ Construire ses logiciels en tenant compte de l'expérience collective des ingénieurs de logiciels

Réutiliser des solutions bien connues pour résoudre des problèmes bien connus

Exemple

◆ Problème

- supporter les différentes catégories d'interfaces graphiques
 - ex: différents affichages graphiques pour les mêmes statistiques

◆ Solution

- Séparer les responsabilités relatives aux données, de celles relatives à la fonctionnalité, et à l'affichage

◆ Pattern

- Modèle-Vue ou Modèle-Vue-Contrôleur

Patterns: Définition

Un pattern est une règle en trois parties exprimant une relation entre un contexte, un problème et une solution (Alexander)

Patterns: Définition (Architecture)

◆ Pattern

- relation entre un contexte, des aspects contradictoires, et une configuration spatiale
- règle qui montre comment la configuration spatiale peut être utilisée pour résoudre les aspects contradictoires, à chaque fois que le contexte le requiert

◆ Exemple: fenêtre

- Contexte: architecture d'intérieur d'un bureau
- Aspects contradictoires (problème): se tourner vers la lumière et être assis dans un fauteuil confortable
- Configuration Spatiale (solution): organiser un espace permettant de s'asseoir confortablement face à une fenêtre

Patterns: Définition

- ◆ Permettent de prendre en compte des **propriétés non fonctionnelles** comme:
 - robustesse au changement, à l'extensibilité
 - fiabilité, testabilité
- ◆ Ne fournissent pas une solution complète:
 - squelette ou schéma pour une solution générique à une famille de problèmes

Description de Patterns (Logiciels)

- ◆ Description uniforme pour tous les patterns
- ◆ Nom
 - significatif pour permettre une recherche efficace du pattern
- ◆ Contexte
 - situation qui présente un problème (peut être précis ou large)
- ◆ Problème
 - problème récurrent qui a lieu dans le contexte (problèmes avec ses aspects contradictoires)

Description de Patterns

◆ Solution

- solution éprouvée au problème
- configuration capable de balancer les aspects contradictoires
 - structure (aspects statiques): diagramme
 - comportement à l'exécution (aspects dynamiques): scénarios

◆ Règles d'implémentation

◆ Codes exemples

◆ Variantes

Observer (Modèle-Vue)

◆ Contexte

- Un composant utilise des données et informations procurées par un autre composant

◆ Problème

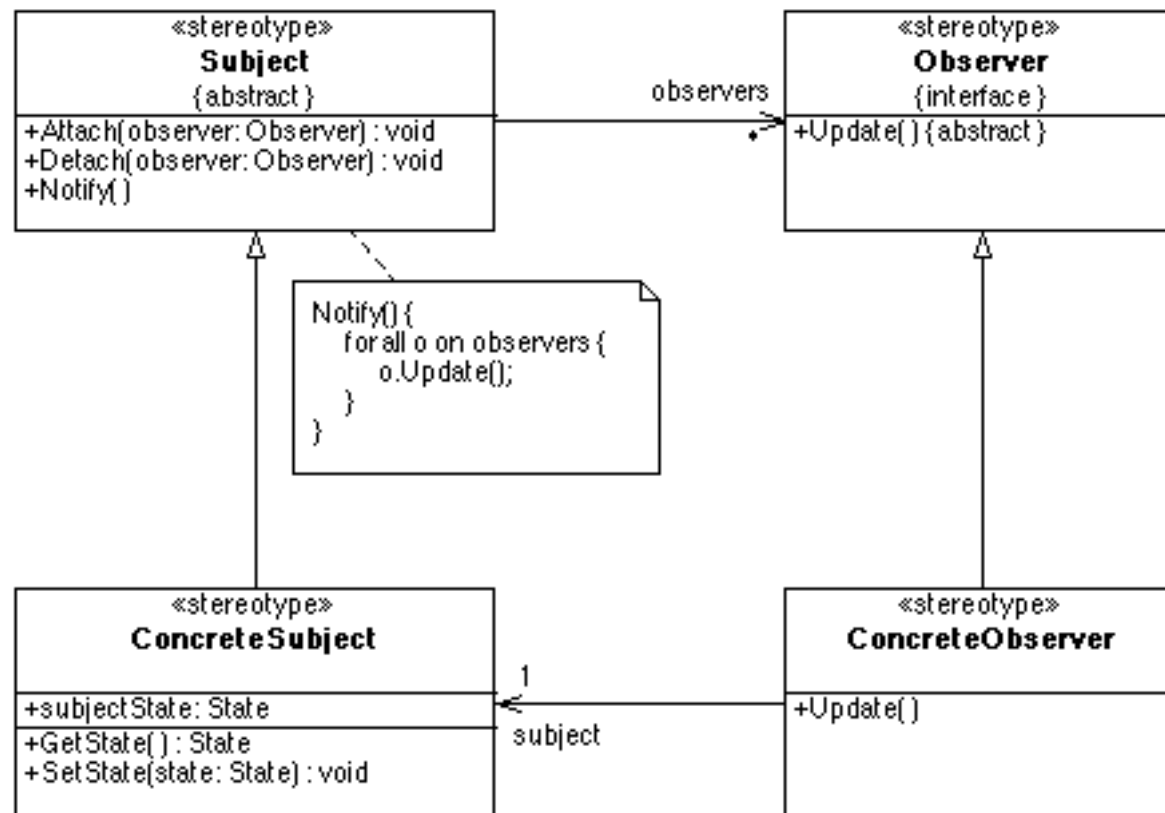
- Définir des dépendances entre objets, de telle sorte que lorsqu'un objet change d'état, tous ceux qui en dépendent sont mis à jour automatiquement.
- Il faut éviter que le composant fournissant les informations ne dépendent des autres composants.
- Exemple: mêmes données avec plusieurs affichages différents

Observer (Modèle-Vue)

◆ Solution

- Introduire un mécanisme de **propagation des changement** entre le détenteur d'informations (Subject) et les composants qui en dépendent (observers)
- Structure et collaborations:
 - Une interface Subject, observée par les observers et qui permet d'attacher ou de détacher des observers
 - Une interface Observer qui déclare une méthode update() de mise à jour des observers
 - Une classe ConcreteSubject qui implante l'objet observé, et qui informe les observers lors de modifications
 - Une classe ConcreteObserver qui maintient une référence sur un objet de la classe ConcreteSubject et implante l'observer pour que son état soit cohérent avec celui du sujet

Observer (Modèle-Vue)



Fournisseur
de données

Utilisateur
des données

Template Method

◆ Contexte

- Partager des algorithmes uniques dont l'implémentation concrète peut varier

◆ Problème

- Comment implanter les parties invariantes d'un algorithme et laisser les sous-classes implanter les parties variables
- Exemple:
 - l'ouverture d'un document se fait en plusieurs étapes: le fichier peut-il être ouvert, créer un objet représentant le document (l'information dans le fichier) spécifique à l'application, lire le document.
 - Mais chaque étape est particulière aux documents et à l'application.

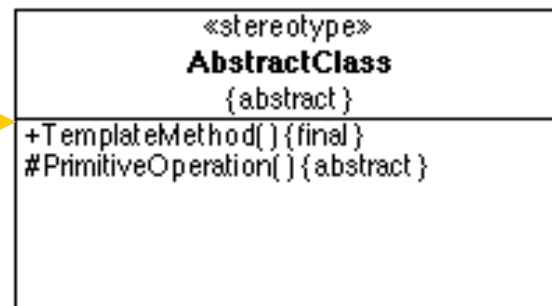
Template Method

◆ Solution

- Définir le **squelette de l'algorithme** dans une opération et implanter les **étapes** de l'algorithme dans les **sous-classes**
- Structure et collaborations:
 - Une classe abstraite et plusieurs sous-classes concrètes
 - La classe abstraite définit une (ou plusieurs) **Template méthodes** (qui ne sont pas redéfinies par les sous-classes).
 - La classe abstraite définit une ou plusieurs méthodes abstraites **Primitives** (redéfinies par chacune des sous-classes). Une **Template méthode** fait appel aux méthodes **Primitives**.

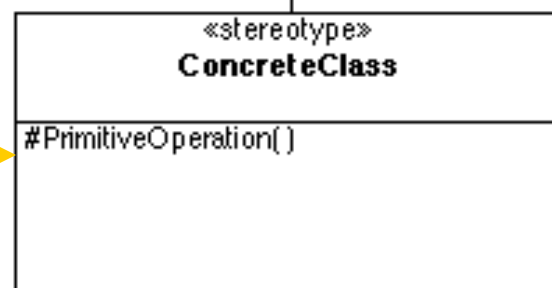
Template Method

Squelette



```
TemplateMethod() {
    ...
    PrimitiveOperation();
    ...
}
```

Etapes



Builder

- ◆ Contexte

- Création d'objets ayant une représentation interne et externe différente

- ◆ Problème

- Comment créer des objets qui peuvent être convertis dans plusieurs représentations différentes, et pour lesquels toutes les représentations ne sont pas connues à l'avance. On veut pouvoir ajouter de nouvelles représentations au fur et à mesure.
- Exemple: convertisseur de texte RTF vers ASCII, TEX, etc.

Builder

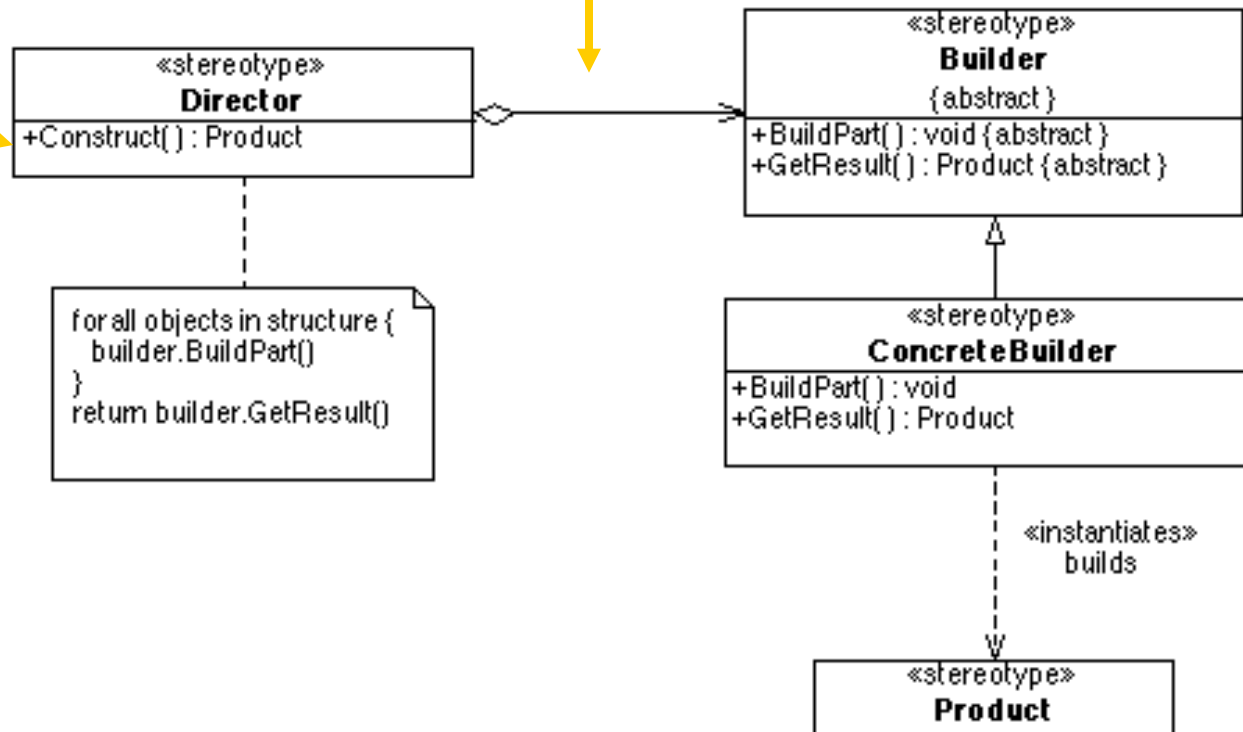
◆ Solution

- Séparer la construction d'un objet complexe de sa représentation, de telle sorte que le **même processus de construction** soit capable de créer **différentes** représentations de l'objet
- Structure et Collaboration
 - Une interface **Builder** qui définit les parties à créer
 - Une classe **Director** capable de créer l'objet en utilisant une interface **Builder**
 - Des classes **ConcreteBuilder** qui implantent les parties à créer (chaque classe permettant une représentation différente)

Builder

Choix de
la représentation

Création
de l'objet



Factory

- ◆ Contexte

- création d'objets par des sous-classes

- ◆ Problème

- une classe ne peut anticiper la classe d'objets à créer
 - exemple:
 - une application sait quand elle doit créer un document (choix de l'utilisateur dans un menu)
 - mais elle ne sait pas quel type de document (l'utilisateur choisit dans le menu le type de document).

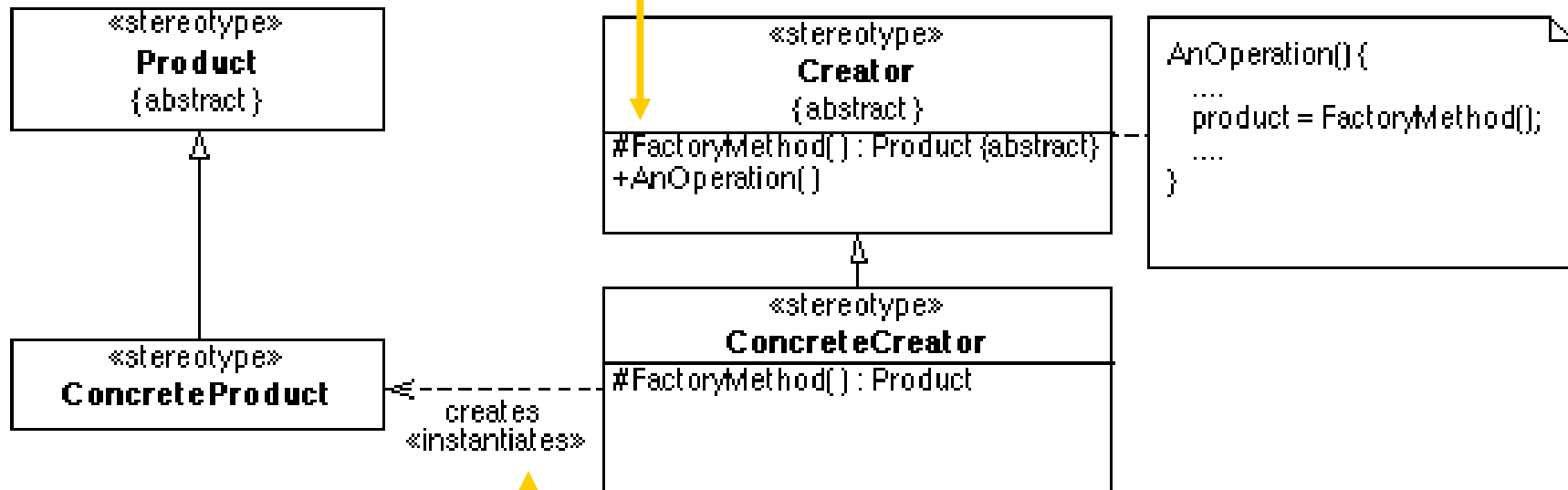
Factory

◆ Solution

- Définir une interface pour créer un objet. Ce sont les sous-classes qui instancient réellement l'objet.
- Structure et Collaborations
 - Une interface **Product** qui définit les objets à créer
 - Une classe **ConcreteProduct** qui implante l'interface **Product**
 - Une classe abstraite **Creator**, qui ne sait pas à l'avance de quelle classe sont les objets qu'il faut créer et qui définit une méthode abstraite, la **Factory** méthode, pour créer ces objets
 - Une classe **ConcreteCreator** qui implante la **Factory** méthode de sorte qu'elle retourne une instance de **ConcreteProduct**

Factory

Interface de
Création



Création
réelle

Proxy

- ◆ Contexte

- Accéder indirectement à des objets

- ◆ Problème

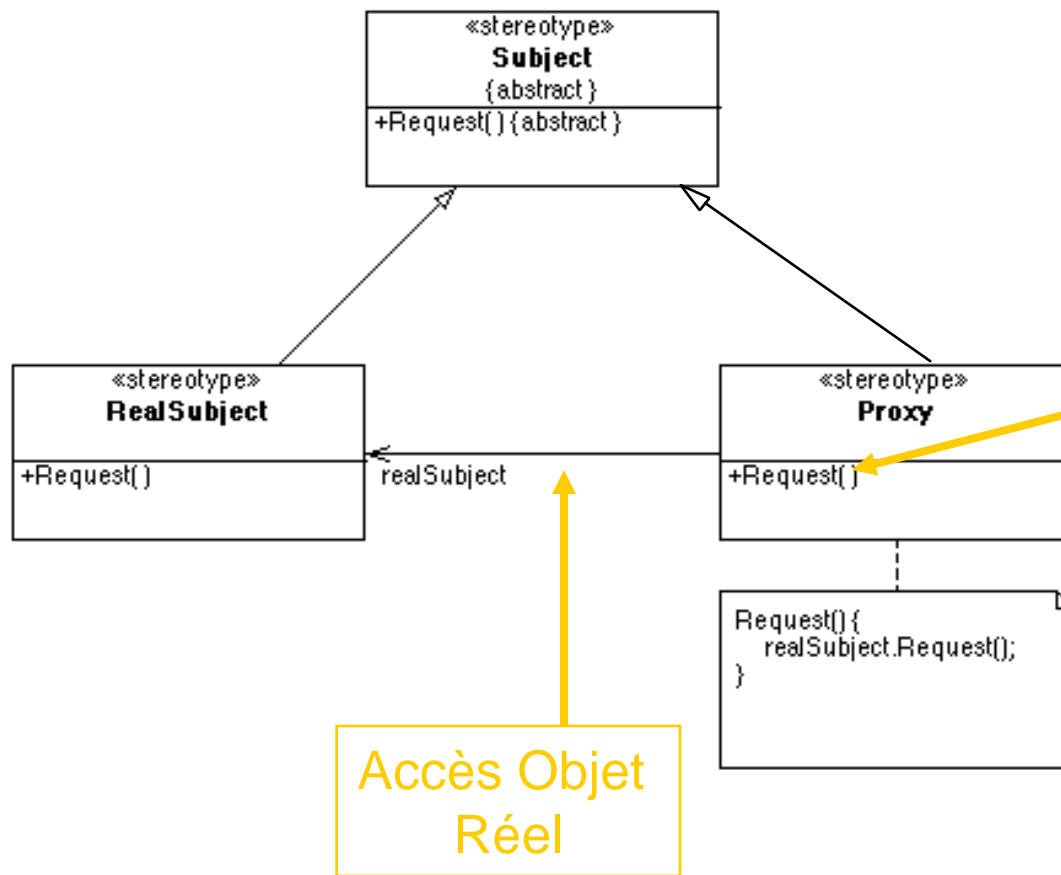
- On a besoin d'accéder à un objet mais on ne veut pas d'un pointeur direct sur l'objet (raisons de sécurité, d'efficacité)
 - Exemple:
 - La création d'un objet qui coûte cher est repoussée au moment où l'on a vraiment besoin de l'objet.
 - Ouvrir un document sans ouvrir immédiatement tous les graphiques qu'il contient. Attendre d'arriver sur les pages qui contiennent des graphiques.

Proxy

◆ Solution

- Contrôler l'accès à un objet en utilisant un **objet intermédiaire**: le proxy
- Structure et Collaboration
 - Une interface **Subject** qui est une interface commune au sujet réel et au proxy
 - Une classe **Proxy** qui maintient une référence sur l'objet réel, et contrôle l'accès à l'objet réel
 - Une classe **RealSubject** qui implante l'objet réel représenté par le proxy
 - Le client accède au proxy qui lui-même accède ensuite à l'objet réel

Proxy



Mediator

- ◆ Contexte

- Groupe d'objets fortement couplés

- ◆ Problème

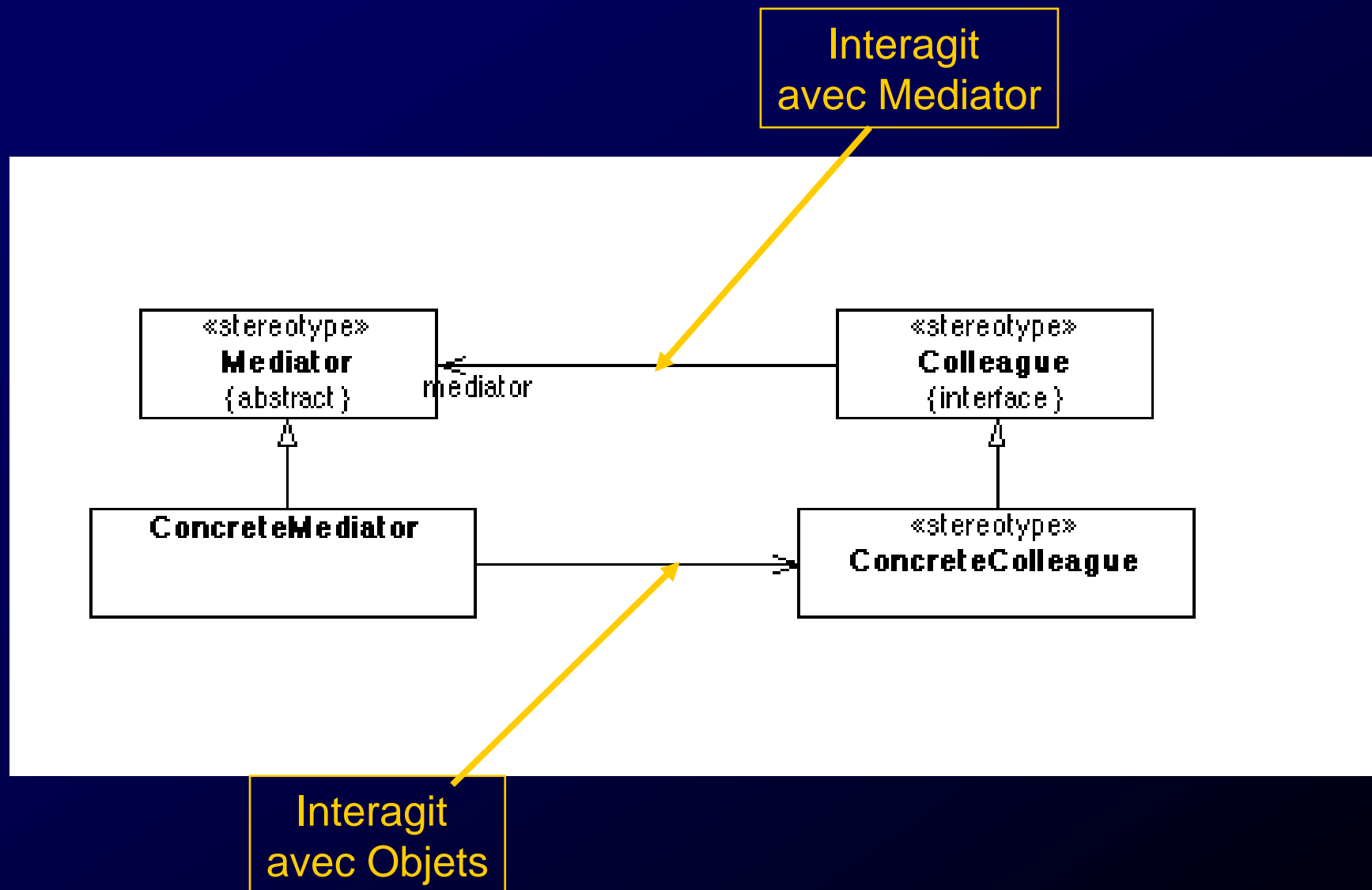
- Comment réutiliser des objets faisant partie d'un groupe d'objets qui dépendent fortement et de manière complexe les uns des autres
- Exemple:
 - Une interface graphique ayant plein d'objets qui dépendent tous les uns des autres (cliquer sur un bouton rend disponible ou indisponible d'autres éléments de l'interface)

Mediator

◆ Solution

- Définir un objet qui **encapsule** la manière dont un ensemble d'objets interagit. Cet objet, le Mediator, permet à cet ensemble d'objets d'être plus indépendants et rend la structure et les dépendances entre objets plus simples à comprendre.
- Structure et Collaborations
 - Une interface Mediator qui contrôle et coordonne les interactions d'un groupe d'objets (Colleagues)
 - Une classe ConcreteMediator qui implante la collaboration entre objets
 - Des classes Colleagues qui définissent les objets qui doivent communiquer entre eux

Mediator



Catégories de Patterns

◆ Patterns Architecturaux

- Patterns de hauts niveau, qui définissent la structure organisationnelle du système
- Ex: Modèle-Vue

◆ Design Patterns

- Patterns de niveau moyen, utilisés comme schémas pour raffiner des sous-systèmes. Ils permettent de résoudre des problèmes de conception.
- Types: création / structure / comportement
- Ex: Factory, Builder / Proxy / Mediator, Template

◆ Idiômes

- Patterns de bas niveau spécifiques à un langage de programmation

Niveau
d'abstraction



Relations entre Patterns

- ◆ Raffinement

- L'implantation d'un pattern soulève elle-même de nouveaux problèmes. Un autre pattern peut résoudre ce problème:
 - donc ... utiliser un pattern pour implanter un autre pattern

- ◆ Variantes

- Solutions à des problèmes similaires

- ◆ Combinaisons de patterns pour résoudre un problème

- Utiliser des patterns au même niveau d'abstraction

Systeme de Patterns

◆ Systeme ou langage de Patterns

- collection de patterns
- combinaisons de patterns
- dépendances et relations entre patterns
- classifieur:
 - les catégories de patterns
 - les catégories de problèmes

◆ Langage

- Patterns = Mots du langage
- Règles d'implémentation et combinaisons = Grammaire

Patterns: Contexte d'Utilisation

◆ Méthodologies

- définissent des étapes de développement
- général, indépendant du problème à résoudre

◆ Styles Architecturaux

- définition de systèmes logiciels en termes d'organisation structurelle (composants, relations)

◆ Patterns:

- complètent l'analyse et la conception avec des techniques concrètes (micro-méthode)
- plus abstraits que classes, composants et frameworks (micro-architecture)

◆ Frameworks et Composants

◆ Langages Orienté-Objets

Bibliographie

- ♦ E. Gamma, R. Helm, R. Johnson, J. Vlissides: « *Design Patterns - Elements of Reusable Object-Oriented Software* ». Addison-Wesley, 1995.
- ♦ F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: « *A System of Patterns - Pattern-Oriented Software Architecture* ». John Wiley & Sons. 1999.