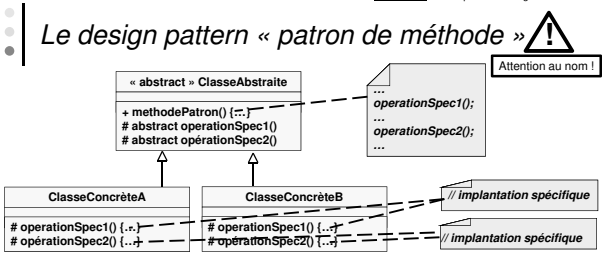
Modèles de conception réutilisables ou « Design Patterns »

Jean-Paul ARCANGELI
Jean-Paul.Arcangeli@irit.fr
UPS – IRIT

M1 MIAGE FI+FA – Ingénierie Logicielle

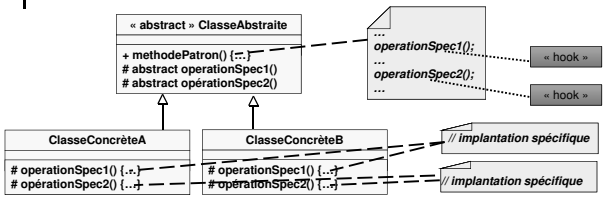
2021-2022

Le design pattern « patron de méthode »



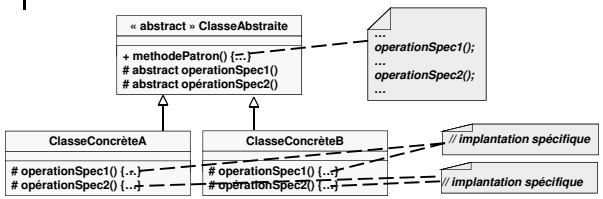
- Intention (objectif)
  - Partager du code commun
  - Reporter dans les sous-classes une partie d'une opération sur un objet
    - Algorithmes (méthodes) avec partie invariable et partie spécifique
- L'héritage supporte
  - La mise en facteur du code des parties communes (à des méthodes)
  - La spécialisation des parties qui diffèrent

Le design pattern « patron de méthode »



- methodePatron() : méthode générique
  - Partie commune (code partagé)
  - Fournit la structure générale de l'algorithme
  - Une partie de l'implantation se trouve dans la sous-classe (externalisation)
- operationSpec1() et operationSpec2() : parties spécifiques
  - Intégration *via* des « hooks » dans methodePatron()

Le design pattern « patron de méthode »



- Objectif : organisation du code (des classes)
  - Patron de méthode est un pattern « de niveau classe » (*vs* « de niveau objet »)
- S'occupe du comportement de l'entité
  - Rôle « comportemental » (*vs* « structurel » ou « créationnel »)
- Utilise le polymorphisme

Le modèle « Stratégie » (1/4)

- Stratégie
  - Pattern comportemental de niveau objet
- Intention
  - Découpler l'algorithme utilisé de la classe qui l'utilise
  - Permettre de définir des objets qui exécutent algorithmes inconnus à la conception et/ou interchangeables
    - Les algorithmes peuvent évoluer indépendamment des objets qui les utilisent

Le modèle « Stratégie » (2/4)

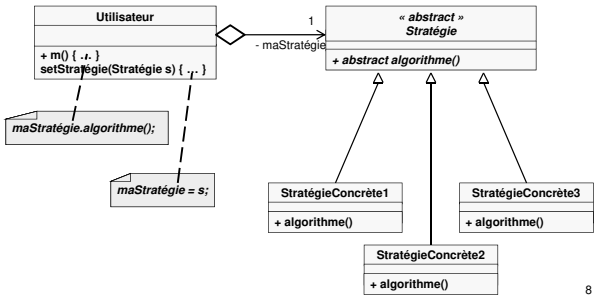
- Motivation
  - Pour éviter de coder « en dur » les algorithmes au sein des classes qui les utilisent
    - Pour séparer les codes des algorithmes et les codes des classes utilisatrices
    - Pour pouvoir ajouter de nouveaux algorithmes, ou modifier ou retirer des algorithmes existants
    - Pour pouvoir changer d'algorithme dynamiquement
  - Par exemple, dans une fenêtre de texte, pour gérer l'affichage des lignes et la césure des mots, on peut employer différents algorithmes
    - Une famille d'algorithmes pour l'affichage, conforme à une même interface « stratégie »
    - Un algorithme est encapsulé dans un objet de type « stratégie »
    - L'objet de type « stratégie » est un délégué de l'objet utilisateur

Le modèle « Stratégie » (3/4)

- Participants
  - *Stratégie* : classe abstraite ou interface qui déclare une interface commune aux algorithmes (super-type)
  - *StratégieConcrète* : implémente l'algorithme conformément à l'interface *Stratégie*
  - *Utilisateur* : classe utilisatrice (cliente) de l'algorithme qui gère une référence sur un objet de type Stratégie
- Collaborations
  - un *utilisateur* transmet les requêtes à l'objet *stratégie* (sous-traitance)
  - l'objet *stratégie* peut éventuellement accéder à des données propres à l'*utilisateur* à travers une méthode dédiée

Le modèle « Stratégie » (4/4)

- Structure



Le(s) modèle(s) « Adaptateur »

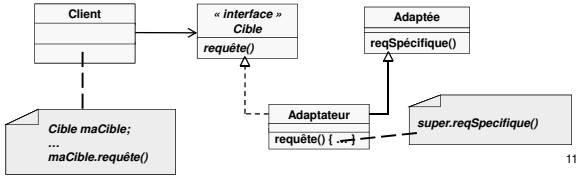
- Adaptateur
- Intention
  - Convertit l'interface d'une classe en une autre conforme aux besoins d'un client (c'est de l'adaptation de l'existant)
  - Permet l'interaction entre objets, instances de classes dont les interfaces sont différentes
- Motivation
  - La réutilisation de classe d'une boîte à outils peut être empêchée parce que l'interface ne correspond pas au besoin « métier » ou au système existant côté client (« ils ne parlent pas la même langue »)
  - On ne veut pas modifier le code des classes impliquées

Le(s) modèle(s) « Adaptateur »

- Participants
  - *Cible* : *interface* spécifique au domaine du client
  - *Client* : classe qui collabore avec une autre qui implémente *Cible*
  - *Adaptée* : classe ou interface existante réutilisée qui diffère de *Cible* et doit être adaptée
  - *Adaptateur* : classe qui adapte l'interface de l'adapté au client, donc implante l'interface *Cible* et
    - Soit étend la classe *Adaptée* (cas d'un adaptateur « de classe) »
    - Soit est associée à la classe *Adaptée* (cas d'un adaptateur « d'objet »)
      - Délégation
- Collaboration
  - Les clients invoquent les opérations d'une instance d'*Adaptateur* qui appelle les opérations de la classe *Adaptée*

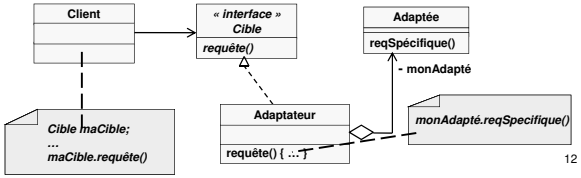
Le modèle « Adaptateur » de niveau classe

- Collaboration (suite)
  - Le client invoque directement l'objet adapté via une méthode ajoutée pour la conformité des interfaces
    - L'interface fournie par l'adaptateur étend l'interface fournie par l'adapté avec l'interface requise par le client
- Structure



Le modèle « Adaptateur » de niveau objet

- Collaboration (suite)
  - Un objet adaptateur sert d'intermédiaire entre l'objet client et l'objet adapté
    - Son interface est celle requise par le client
    - Il traduit la requête client en une requête pour l'adapté, invoque l'adapté, récupère le résultat, le transforme éventuellement et le retransmet au client
  - Les objets clients et adaptés s'ignorent mutuellement
- Structure



Le(s) modèle(s) « Adaptateur »

- Conséquences
  - Le rôle de l'adaptateur peut être plus ou moins complexe selon l'écart entre le client et l'adapté
  - Un adaptateur de niveau « classe »
    - Peut redéfinir certaines méthodes de la classe « adaptée »
    - N'introduit pas d'objet intermédiaire (donc plus efficace en temps d'exéc.)
  - Un adaptateur de niveau « objet »
    - Peut gérer plusieurs objets adaptés
    - Peut adapter des objets issus de sous-classe de *Adaptée*
    - Peut intercepter les requêtes ou les exceptions et les traiter lui-même…
    - Introduit un objet intermédiaire donc une indirection
    - Rq : cette sol. peut également fonctionner en Java si Cible est une classe
      - On remplace le lien d'implantation par un lien d'héritage (pour l'adaptateur de niveau « classe » : problème d'héritage multiple)
- Modèles apparentés
  - Pont, Décorateur, Procuration (Proxy), Façade

Le modèle « State » (1/5)

- Nom
  - State (modèle comportemental de niveau objet)
- Intention
  - Permettre à un objet d'adapter son comportement en fonction de son état interne
- Motivation
  - L'approche classique qui consiste à utiliser des conditions dans le corps des méthodes conduit à des méthodes complexes
  - En réifiant l'état sous forme d'objet (1 classe par état possible) et en déléguant le traitement de la méthode à cet objet, on rend le traitement spécifique à l'état courant de la machine à états
- Indications d'utilisation
  - Quand le comportement d'un objet dépend de son état et que l'implantation de cette dépendance à l'état par des instructions conditionnelles est trop complexe

Le modèle « State » (2/5)

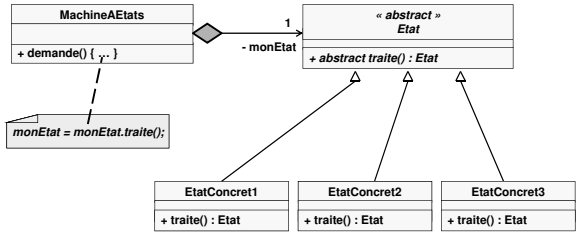
- Participants
  - *MachineAEtat* : classe concrète définissant des objets qui sont des machines à états (pouvant être décrits par un diagramme d'états-transitions). Cette classe maintient une référence vers une instance d'état qui définit l'état courant
  - *Etat* : classe abstraite qui spécifie les méthodes liées à l'état et qui gère l'association avec la machine à états
  - *Etatconcret1*... : (sous-)classes concrètes qui implantent le comportement de *MachineAEtat* pour chacun des ses états

Le modèle « State » (3/5)

- Collaborations
  - La *MachineAEtat* transmet la requête à l'objet *EtatConcret* qui la traite par « délégation » (sous-traitance) ; ce traitement provoque la mise à jour de l'état de MachineAEtat
    - C'est l'objet *EtatConcret* qui décide du nouvel état (EtatConcret) de *MachineAEtat* et initie la mise à jour
    - Le nouvel objet *EtatConcret* est retourné en résultat du traitement (cf. signatures des méthodes dans le diagramme de classes)
      - Alternativement (variante), *EtatConcret* peut utiliser une méthode de **callback** offerte par *MachineAEtat*

Le modèle « State » (4/5)

- Structure

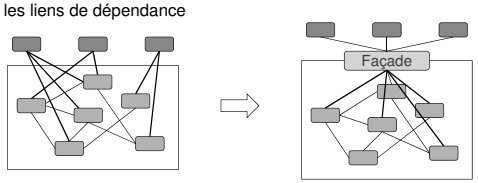


Le modèle « State » (5/5)

- Modèle apparenté
  - Stratégie
    - Fondamentalement, State diffère de Stratégie par son intention. State a pour intention de permettre à un objet d'adapter son comportement en fonction de son état et de changer cet état
    - Dans la solution (mise en œuvre), le changement de stratégie est externe (méthode setStratégie()) alors que c'est l'état lui-même qui provoque le changement d'état (opération interne)

Le modèle « Façade » (1/4)

- Façade
  - Pattern structurel de niveau objet
- Intention
  - Fournir une interface unifiée et simplifiée à l'ensemble des interfaces d'un sous-système (organiser et simplifier)
- Motivation
  - Réduire la complexité de la relation client/sous-système et organiser les liens de dépendance

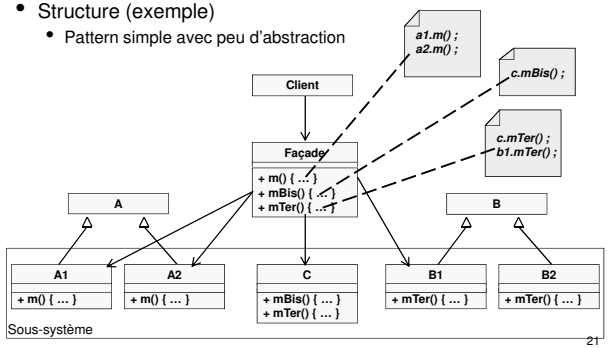


Le modèle « Façade » (2/4)

- Participants
  - La *façade* fournit aux clients une interface unifiée de plus haut niveau d'abstraction que celles des composants
    - À travers des procédures construites à partir des fonctionnalités du sous-système (de plus bas niveau)
  - Les classes du sous-système implémentent les différentes fonctionnalités
    - La classe Façade possède des liens vers les classes du sous-système
- Collaborations
  - Les clients communiquent avec le sous-système en envoyant des requêtes à la façade qui les répercute aux objets du sous-système

Le modèle « Façade » (3/4)

- Structure (exemple)
  - Pattern simple avec peu d'abstraction



Le modèle « Façade » (4/4)

- Conséquences
  - La façade masque le sous-système au client donc le rend plus facile à utiliser
    - Elle permet de contrôler l'accès aux opérations (rendre invisible certaines opérations en dehors du sous-système)
    - Elle n'empêche pas (forcément) le client d'accéder directement aux composants du sous-système si besoin
  - On peut définir différentes façades « métier » pour un même sous-système
  - La façade peut ajouter une « plus-value » c-à-d. offrir des services de haut niveau
  - La façade réduit le couplage entre les classes client et sous-système
    - Une évolution du sous-système n'impacte pas directement le client
- Modèles apparentés
  - Adaptateur, Médiateur, Proxy

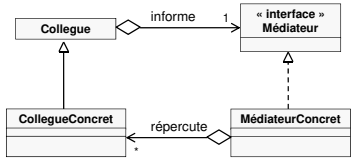
Le pattern Médiateur 1/5

- Médiateur
  1. Modèle « comportemental » de niveau « objet »
  2. Alias : aucun

Le pattern Médiateur 2/5

- Problème et contexte
  3. Intention
     - Définit un objet qui encapsule les modalités d'interaction (gestion et contrôle) d'un ensemble d'objets
     - Favorise le couplage faible en permettant aux objets de ne pas se référencer les uns les autres
  4. Motivation (justification)
     - La conception objet favorise la distribution des comportements. Elle peut conduire à des structures d'interconnexion complexes, donc à des difficultés en cas de modification
     - Exemple : boîtes de dialogue (widgets) dans une interface graphique
  5. Indications d'utilisation
     - Interconnexions complexes dans un ensemble d'objet
     - Réutilisation difficile des classes du système (du fait des références multiples)

Le pattern Médiateur 3/5



- Solution
  6. Structure
  7. Constituants (ou participants)
     - *Médiateur* définit l'interface du médiateur pour les objets *Collegue*
     - *MédiateurConcret* implante la coordination et gère les associations
     - *Collegue* regroupe les attributs, associations et méthodes communes des objets en interaction (classe abstraite)
     - *CollegueConcret* implante les objets en interaction
  8. Collaborations
     - Les collègues émettent et reçoivent des requêtes du médiateur. Le médiateur assure le routage des requêtes entre collègues
     - *A compléter par des diagrammes de séquence ou de communication*

Le pattern Médiateur 4/5

- Conséquences et réalisation
  9. Conséquences
     - Le médiateur centralise la logique d'interaction. Il remplace des interactions N-N entre collègues par des interactions 1-N
       - La complexité n'est pas distribuée mais centralisée dans le médiateur
       - La logique d'interaction est séparée de la logique métier des collègues
     - La présence d'un médiateur réduit le couplage entre collègues
     - Surcoût des indirections à l'exécution (c'est le prix à payer !)
       - Au besoin, les liens directs restent possibles
  10. Implémentation
     - L'interface (ou classe abstraite) Médiateur n'est pas obligatoire lorsque le médiateur est unique
     - La communication entre collègues et médiateur peut se faire par évènements
  11. Exemples de code

Le pattern Médiateur 5/5

- Compléments
  12. Utilisations remarquables
  13. Modèles apparentés
     - Le modèle Médiateur diffère de Façade… *(à développer)*
     - Le modèle Observateur peut être utilisé pour la communication entre collègues et médiateur
     - On peut implanter le médiateur comme un Singleton

Le modèle « Observateur » (1/6)

- Observateur
- Alias
  - Souscription-diffusion (*publish-subscribe*)
- Intention
  - Définit une relation un-à-plusieurs (1-N) entre des objets de telle sorte que lorsqu'un objet (le « sujet ») change d'état, tous ceux qui en dépendent (les « observateurs ») en soient notifiés et mis à jour « automatiquement »
  - Maintien d'une cohérence d'état entre objets
- Motivation
  - Ne pas introduire de couplage fort entre les classes sujet et observateur
  - Pouvoir attacher et détacher dynamiquement les observateurs
  - *Par exemple, pour afficher différentes représentations d'un jeu de données (des graphiques extraits d'un tableur par exemple)*

Le modèle « Observateur » (2/6)

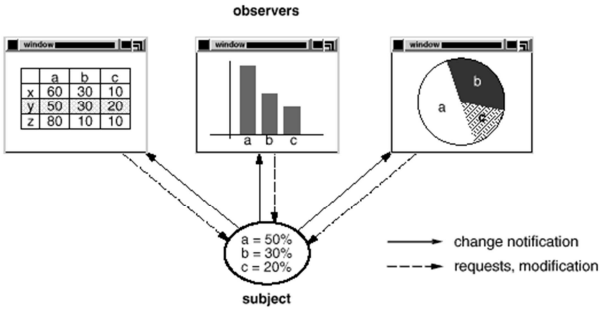


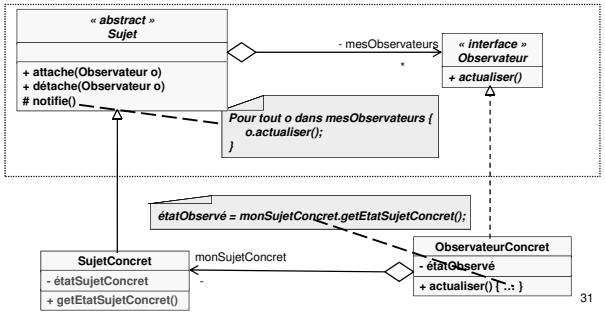
Figure extraite de
« Design Patterns, Elements of Reusable Object-Oriented Software »,
E. Gamma, R. Helm, R. Johnson & J. Vlissides, 1995

Le modèle « Observateur » (3/6)

- Participants
  - *Sujet* : classe abstraite en association avec *Observateur*
    - Offre une interface pour attacher et détacher les observateurs
    - Implémente la notification (protocole de diffusion)
    - Peut aussi être une interface ou une classe concrète
  - *Observateur* : interface qui spécifie la réception de la notification
  - *SujetConcret* : mémorise l'état et envoie la notification
    - Offre une méthode d'acquisition d'état aux observateurs (mode *pull*)
    - Un objet *SujetConcret* a la référence de ses *ObservateurConcrets*
  - *ObservateurConcret* : gère la référence au sujet concret et, éventuellement, mémorise l'état en cohérence avec le sujet
    - Sollicite le sujet pour acquérir l'état (en mode *pull*)

Le modèle « Observateur » (4/6)

- Structure



Le modèle « Observateur » (5/6)

- Conséquences
  - On peut modifier sujets et observateurs indépendamment
    - Pas de lien de la classe SujetConcret vers la classe ObservateurConcret
    - On peut ajouter de nouveaux observateurs sans avoir à modifier le sujet
      - Initialement, on a identifié que les observateurs pouvaient varier
  - Un observateur peut observer plusieurs sujets (relation N-1 possible)
  - Communication possible en mode *push*
    - Mais interface de notification spécifique (côté observateur)
  - D'autres modèles sont possibles en termes de synchronisation (*i.e.* évènementiel) et d'interaction

Le modèle « Observateur » (6/6)

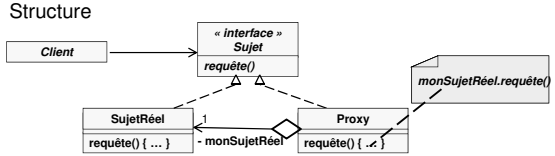
- Implémentation
  - Il existe une implémentation native en Java
    - Classe java.util.Observable et interface java.util.Observer
    - Deprecated in Java 9 (=> *Listeners*)
- Utilisations remarquables
  - Dans la mise en œuvre des IHM
    - En particulier dans le modèle MVC
- Modèles apparentés
  - Médiateur
    - Dans certains cas, Observateur peut supporter la mise en œuvre de Médiateur

Le modèle Proxy (1/3)

- Intention
  - Contrôler l'accès à un objet S (Sujet) au moyen d'un autre objet P (intermédiaire) qui se substitue à S
  - Cacher au client de S tout ce qui concerne l'identité et la localisation de S et la réalisation de l'appel
- Motivation
  - L'accès à un objet doit parfois être contrôlé. Par exemple, dans une exécution répartie, l'appel de méthode d'un objet client sur un objet distant n'est pas directement possible. On veut rendre possible cet appel tout en cachant au client la complexité de l'opération
- Indications d'utilisation
  - Quand l'accès à un objet doit être contrôlé, soumis à un pré-traitement (ou un post-traitement) externe au sujet lui-même
  - Utilisations remarquables : proxy distant, virtuel, de protection, de synchronisation, intelligent…

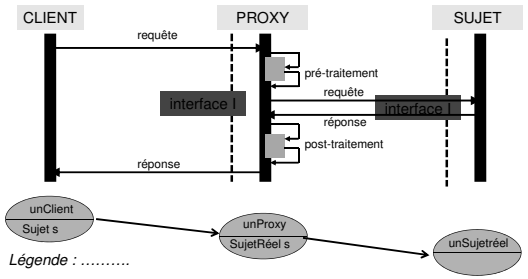
Le modèle Proxy (2/3)

- Participants
  - Sujet : interface commune entre le proxy et le sujet réel
  - SujetRéel : classe concrète du sujet réel, représentée te contrôlé par le Proxy
  - Proxy : classe concrète de l'objet qui se substitue au sujet réel
- Structure



Le modèle Proxy (3/3)

- Collaborations
  - Le proxy (représentant du sujet) reçoit les appels pour le sujet, et lui fait suivre sous condition…



Le modèle « Décorateur » (1/6)

- Décorateur
  - Pattern structurel de niveau objet
- Alias
  - Enveloppe
- Intention
  - Permet de remplacer un objet de base par un autre objet (avec conformité de type) tout en lui ajoutant des compétences supplémentaires (de manière dynamique)
  - Donne une alternative souple à l'héritage (via la délégation)

Le modèle « Décorateur » (2/6)

- Motivation
  - Le décorateur est un objet qui offre l'interface de l'objet décoré mais qui enveloppe ce dernier et lui ajoute une fonctionnalité
    - L'objet décoré est un délégué du décorateur
  - On peut imbriquer récursivement les décorateurs
  - Par exemple, si on veut agrémenter une fenêtre de texte (qui gère l'affichage et les évènements) d'une barre de défilement, d'un encadrement particulier et/ou d'un compteur de caractères…
    - À chaque « agrément » (barre, cadre, compteur…) correspondra un décorateur de type « fenêtre »
    - Les décorateurs seront composés par délégation pour fabriquer par exemple une fenêtre de texte avec compteur et barre de défilement…

Le modèle « Décorateur » (3/6)

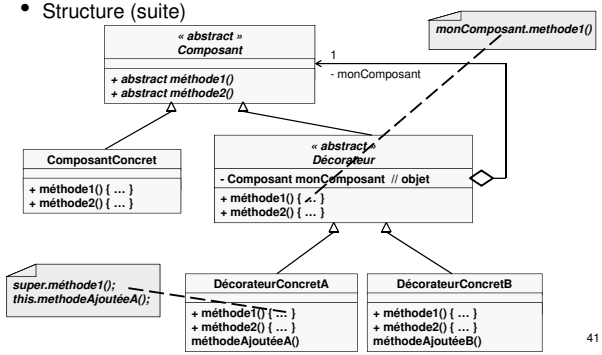
- Indication d'utilisation
  - Pour pouvoir « ajouter » (ou « retirer ») des opérations à des objets sans avoir à modifier les classes existantes
    - Au moment de la configuration (déploiement)
  - Quand l'héritage n'est pas souhaitable, pas possible ou limité

Le modèle « Décorateur » (4/6)

- Participants
  - *Composant* : classe abstraite (ou interface) qui spécifie l'interface des objets qui peuvent être décorés
  - *Composant concret* : classe qui définit un objet à décorer
  - *Décorateur* : classe abstraite qui implante l'interface de *Composant* et qui gère une référence à un *Composant*
  - *Décorateur concret* : ajoute une responsabilité au composant et redéfinit les méthodes de l'interface
- Collaboration
  - Le décorateur transmet les requêtes à l'objet décoré et peut y ajouter des opérations complémentaires

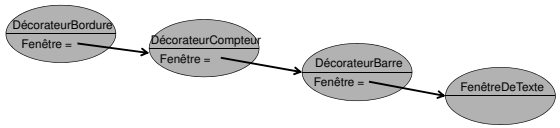
Le modèle « Décorateur » (5/6)

- Structure (suite)



Le modèle « Décorateur » (6/6)

- Structure
  - Diagramme d'objets (exemple)

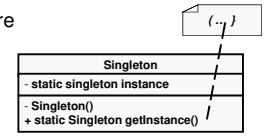


Le modèle « Singleton » (1/7)

- Nom
  - Singleton (modèle créateur)
- Alias
  - *Néant*
- Intention
  - Assurer qu'une classe n'ait qu'une instance et fournir un point d'accès global à celle-ci
- Motivation
  - Pour certaines classes, il est important de n'avoir qu'une seule instance : par exemple d'un serveur d'impression…
  - Pour cela, la classe assure l'unicité de l'instance et fournit un moyen pour y accéder
- Indications d'utilisation
  - S'il ne doit y avoir qu'une instance au plus de la classe

Le modèle « Singleton » (2/7)

- Constituants (ou participants)
  - Une seule classe
    - La classe elle-même et elle seule
  - Cas d'exception (c'est le seul design pattern parmi ceux du GoF)
- Structure



- Collaborations
  - Les clients accèdent à l'instance par le seul intermédiaire de la méthode (possiblement « synchronisée ») *getInstance()*

Le modèle « Singleton » (3/7)

```
public class MonSingleton {
  // l'unique instance
  private static MonSingleton instance = null;
  // le constructeur privé
  private MonSingleton(){
  }
  // méthode (de classe) pour la création d'instance
  public static MonSingleton getInstance() {
      // création « paresseuse de l'instance »
      if (instance == null) {
            instance = new MonSingleton();
      }
      return instance;
  }
}
```

Le modèle « Singleton » (4/7)

- Conséquences
  - La classe elle-même contrôle précisément comment et quand les clients accèdent à l'instance
    - Si l'instance doit être crée systématiquement, on peut la créer au chargement de la classe (simplification !)
  - Le modèle peut être adapté pour contrôler un nombre fixé d'instances
    - Allocation/gestion de « pools » d'objets
  - On peut sous-classer la classe Singleton et préserver le polymorphisme
  - …

Le modèle « Singleton » (5/7)

- Pour aller un peu plus loin…
  - Si on se place dans le contexte de la programmation concurrente (c'est-à-dire « multi-thread ») ?
    - C'est-à-dire si plusieurs accès différents et simultanés (concurrents) sont possibles ?
  - Il faut contrôler l'accès à la méthode getInstance()
    - En Java, la méthode getInstance() doit être « synchronized »

⇒ Il faut adapter la solution (d'où l'importance du contexte !)

Le modèle « Singleton » (6/7)

```
public class MonSingleton {
   // l'unique instance
   private static MonSingleton instance = null;
   // le constructeur privé
   private MonSingleton(){
   }
   // méthode (de classe) pour la création d'instance
   public static synchronized MonSingleton getInstance() {
       // création « paresseuse de l'instance »
       if (instance == null) {
             instance = new MonSingleton();
       }
       return instance;
   }
}
```


Le modèle « Singleton » (7/7)

- Mise en œuvre dans un contexte *multi-thread*
  - Utilisation de « *synchronized* »
    - Mais surcoût à l'exécution !
  - Au besoin, on peut déplacer le verrouillage dans le corps de la méthode
    - Verrouillage seulement quand l'instance n'existe pas
    - Double vérification (mais, en pratique, ça ne suffit pas, cf. « Tête La première, Design Patterns » p. 182)

```
if (instance == null) {
      synchronized (MonSingleton.class) {
            if (instance == null) {
                  instance = new MonSingleton();
            }
      }
}
```

Tous ces éléments documentent le pattern

Le modèle « Factory Method » (1/8)

- Nom
  - Alias « Fabrication »
  - Catégorie « créateur », de niveau classe
- Intention
  - Définir une méthode qui fabrique (crée) des objets sans connaître le type réel (la classe) de ces objets
  - Reporter la création effective dans les sous-classes
    - Utilisation de l'héritage

Le modèle « Factory Method » (2/8)

- Motivation
  - Ne pas faire directement appel à *new*
  - Par exemple, dans une bibliothèque graphique pour pouvoir créer des formes géométriques qui seront définies par l'utilisateur de la bibliothèque (le concepteur de la bibliothèque ignore la classe concrète des objets à créer)
  - Le pattern « Factory Method » introduit une classe abstraite qui offre une méthode abstraite de fabrication
    - Comme patron de méthode !
    - Implantée dans une sous-classe à qui est délégué le choix du type de l'objet à créer
    - Retourne un produit
      - Une classe abstraite (un super-type) représente le type du produit

Le modèle « Factory Method » (3/8)

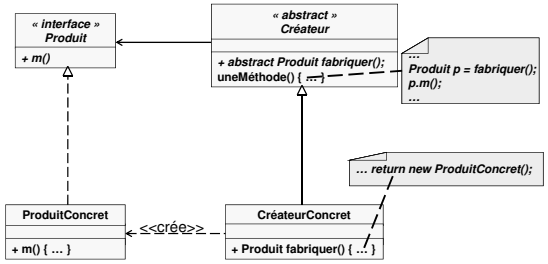
- Indication d'utilisation
  - Une classe ne connait pas la classe (concrète) des objets à créer
  - Une classe attend de ses sous-classes qu'elles précisent les objets qu'elles créent

Le modèle « Factory Method » (4/8)

- Participants
  - *Produit* : définit l'interface des objets à créer
  - *ProduitConcret* : implémente l'interface *Produit*
  - *Créateur* : déclare l'interface de fabrication qui retourne un *Produit*
  - *CréateurConcret* : définit (ou redéfinit) la fabrication pour retourner une instance de *ProduitConcret*
- Collaboration
  - C'est la sous-classe *CréateurConcret* qui réalise la fabrication d'un *ProduitConcret* (vu comme un *Produit*)

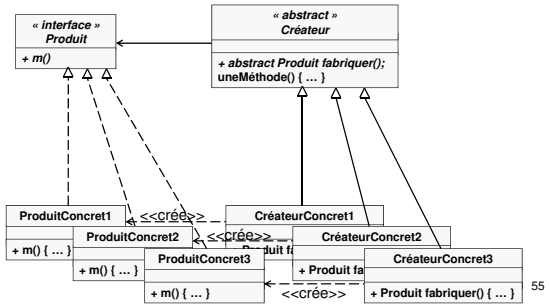
Le modèle « Factory Method » (5/8)

- Structure



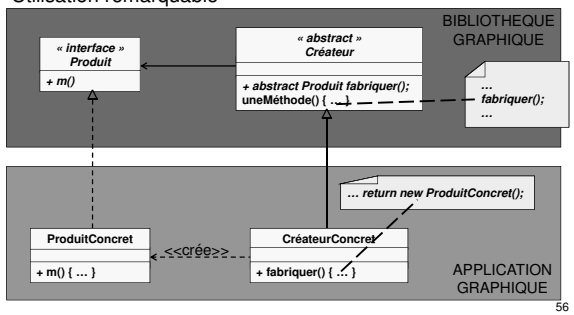
Le modèle « Factory Method » (6/8)

- Exemple



Le modèle « Factory Method » (7/8)

- Utilisation remarquable



Le modèle « Factory Method » (8/8)

- Implémentation
  - *Créateur* peut aussi définir une implémentation par défaut de *fabriquer()*
    - *Créateur* peut donc être une classe concrète
  - La requête de fabrication peut être paramétrée afin de permettre la création de plusieurs (nombreux) types de produits concrets

Le modèle « Fabrique abstraite » (1/4)

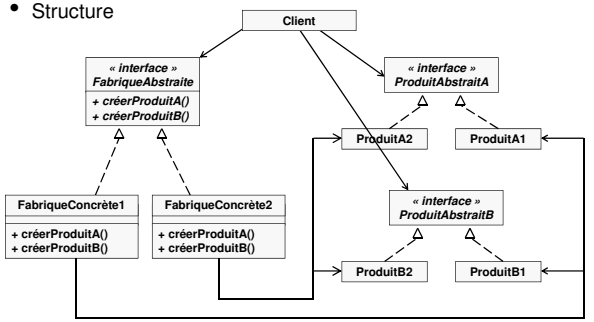
- Fabrique abstraite
  - Catégorie « créateur », de niveau objet
- Intention
  - Permet la création d'objets regroupés en familles sans avoir à spécifier (connaître) leurs classes concrètes
  - Au moyen d'un objet « fabrique »
- Motivation
  - Par exemple, quand on choisit un certain style graphique pour une IHM, on veut créer des objets graphiques (fenêtres, boutons…) conformes à ce style là (famille)
  - On veut éviter de coder « en dur » dans la classe « cliente » la création d'objets (new) en faisant référence à leurs classes concrètes
    - Rendre la classe « cliente » indépendante de la façon dont les objets sont créés, afin de faciliter l'évolution

Le modèle « Fabrique abstraite » (2/4)

- Participants
  - *FabriqueAbstraite* est une interface qui spécifie les méthodes de création des différents objets
  - *FabriqueConcrète1*, *FabriqueConcrète2*… sont les classes concrètes qui implantent *FabriqueAbstraite* pour chaque famille
  - *ProduitAbstraitA*, *ProduitAbstraitB*… sont des interfaces ou des classes abstraites qui définissent les objets de la famille A, B…
  - *ProduitConcretA1*, *ProduitConcretA2* implante (ou hérite de) *ProduitAbstraitA* pour chaque famille de produits
  - *Client* est la classe qui utilise l'interface de *FabriqueAbstraite*
- Collaborations
  - La classe *Client* utilise une instance d'une des fabriques concrètes pour créer les produits (via l'interface de *FabriqueAbstraite*)

Le modèle « Fabrique abstraite » (3/4)

- Structure



Le modèle « Fabrique abstraite » (4/4)

- Conséquences
  - L'interface de « Fabrique abstraite » change dès qu'on introduit un nouveau produit ☹
    - mais l'objectif n'est pas là !
  - Mais pas quand on ajoute une nouvelle famille ☺