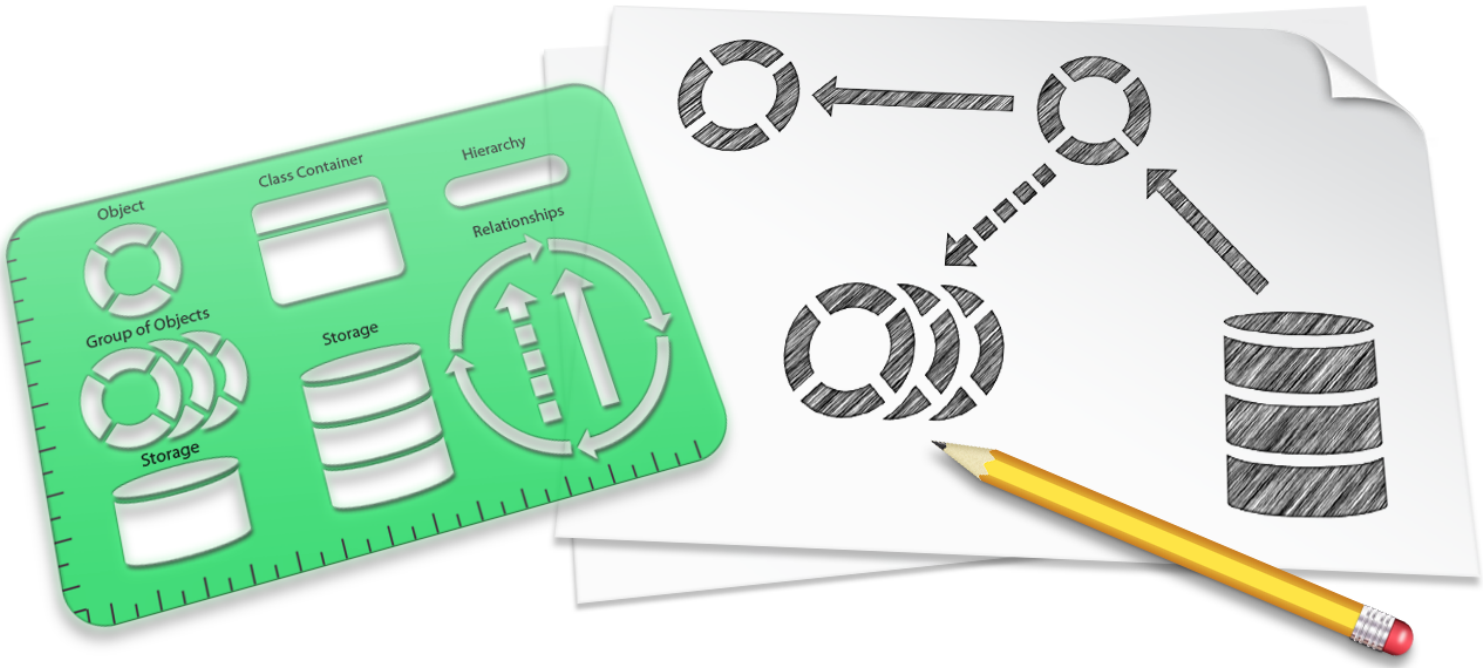
Introduction aux *Design Patterns*



Au menu

Au menu

Motivation

- Concevoir du logiciel est difficile et concevoir du logiciel réutilisable encore plus !
  - Nécessite de chercher :
    - ★ Bonne décomposition du problème
    - ★ Bonnes abstractions logicielles
    - ★ Flexibilité, extensibilité, modularité et élégance
  - Émerge souvent d'un processus itératif (nombreux essais et erreurs)
- Bonne nouvelle : des conceptions réussies existent
  - Présentent certaines caractéristiques récurrentes
  - Mais ne sont pratiquement jamais identiques
- Peut-on décrire, codifier et standardiser ces bonnes conceptions ?
  - Disposer de briques de conception réutilisables
  - Produire plus rapidement du logiciel meilleur

Un (tout petit) peu d'histoire

- Christopher Alexander
  - Architecte
  - Professeur émérite à Berkeley
  - Père des *design patterns* appliqués à l'architecture (2543 patterns)
  - *A Pattern Language: Towns, Buildings, Construction* (1977)

    Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (*GoF*)
  - Application des *design patterns* à la programmation par objet
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (1994)
    - ★ Catalogue de 23 *design patterns*

Qu'est-ce qu'un *design pattern* ?

Une définition

Un *design pattern* décrit une solution à un problème général et récurrent de conception dans un contexte particulier.

- En français, patron de conception
- Ce N'est PAS :
  - Une classe individuelle ou bibliothèque, telle que les listes ou les tables d'association
  - Une conception complète et concrète, ni une implémentation, mais plutôt une description abstraite sur comment résoudre un problème

Éléments d'un *design pattern*

- Nom
  - Concis et significatif
  - Partie utile du vocabulaire de conception
- Problème résolu et applicabilité
  - Quand appliquer le pattern : problème + contexte
- Solution
  - Représentée sous la forme d'un schéma (*e.g.*, diagrammes UML)
  - Participants (classes et objets) et leurs relations/responsabilités/ collaborations
  - Doivent être personnalisés
- Conséquences
  - Avantages et inconvénients d'utiliser le pattern
  - Impacts sur la réutilisation, la flexibilité, l'extensibilité, *etc.*
  - Peuvent être différentes en fonction des variations du pattern

Bénéfices des *design patterns*

- Vocabulaire de conception commun
  - Améliore la compréhension et la documentation de la conception
  - Améliore la communication entre les développeurs
- Capture de l'expérience de conception
  - Facilite la réutilisation de solutions éprouvées vis-à-vis du problème
  - Facilite l'implémentation, la maintenance et l'évolution du logiciel
  - Augmente la productivité et la qualité du logiciel
- Enseignement et apprentissage
  - Est plus facile de comprendre une architecture à partir de descriptions de design patterns que de lire du code
  - Permet de comprendre certains schémas et patrons présents dans les composants

Un peu de lecture

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson et Vlissides, Addison-Wesley, 1994
- *Pattern Hatching : Design Patterns Applied*, Vlissides, Addison-Wesley, 1998
- *Design Patterns Explained: a New Perspective on Object-Oriented Design*, Shalloway et Trott, Addison-Wesley, 2001
- *Head First Design Patterns*, Freeman et Freeman, O'Reilly, 2004

D'autres types de *patterns*

- Styles architecturaux
  - Caractérisent une famille de systèmes qui utilise les même types de composants, d'interactions, de contraintes structurelles et sémantiques, et d'analyses (*e.g.*, *pipes*, *filters*, *brokers*, *blackboard*, *MVC*)
  - Décrits au moyen des termes et des concepts du domaine d'application
  - Niveau d'abstraction/granularité $>$ *design patterns*
- Idiomes de programmation
  - Spécifiques à un langage de programmation particulier
  - Décrits au moyen des constructions du langage de programmation
  - Niveau d'abstraction/granularité $<$ *design patterns*
- *Framework*
  - Modèle d'architecture utilisé pour organiser une application complète
  - Par exemple, J2EE pour les applications Internet en Java
- Anti-patterns
  - Ce qu'il ne faut pas faire (mauvaises solutions)
- Schémas d'organisation
  - Tout ce qui entoure le développement d'un logiciel (humains)
- *Portland Pattern Repository* : `http://c2.com/cgi/wiki`

Comment utiliser les *design patterns* ?

1. Connaître les patterns et les problèmes récurrents qu'ils adressent
2. Durant la phase de conception, identifier les problèmes qui peuvent être résolus par un pattern
3. Consulter le pattern
4. Intégrer correctement le pattern dans le code
   - Déterminer quelles classes devraient remplacer les "stéréotypes" fournis par le pattern
   - Parfois, un pattern ne s'applique pas directement
     - ★ Besoin de l'adapter à la situation
     - ★ Besoin d'utiliser plusieurs patterns pour résoudre le problème

Classification des *design patterns* (*GoF*)

- But : ce que fait le pattern
  - Patterns de création
    - ★ Rendent un système indépendant de comment ses objets sont créés, initialisés et configurés
    - ★ Sont utiles lorsque le système évolue : les classes qui seront utilisées dans le futur peuvent ne pas être connues maintenant
  - Patterns de structure
    - ★ Permettent de composer des classes et des objets afin de former des structures plus importantes
    - ★ Réduisent le couplage entre des classes (découplage interface et implémentation)
  - Patterns de comportement
    - ★ S'intéressent aux interactions entre les objets
    - ★ Décrivent des flots de contrôle complexes

Classification des *design patterns* (*GoF*)

- Portée : à quoi s'applique le pattern
  - Patterns de classe
    - ★ Se focalisent sur les relations entre les classes et leurs sous-classes
    - ★ Utilisent l'héritage
  - Patterns d'objet
    - ★ Se focalisent sur les relations entre les objets
    - ★ Utilisent la composition

Exemple : un éditeur de texte

Caractéristiques

- *WYSIWYG*
- Interface graphique
  - Barre d'outils, barres de défilement, menus *etc.*
- Possibilité de mélanger librement texte et images
- Plusieurs systèmes de fenêtrage
- Vérification orthographique, césure, *etc.*

Exemple : un éditeur de texte

Jeu

1. Je vous énonce un problème de conception pour cet exemple
2. Je vous demande ce que vous proposez comme conception
   - C'est le moment où vous participez
3. Je vous donne des éléments de réponse et des détails sur le pattern

Au menu

Patterns de création

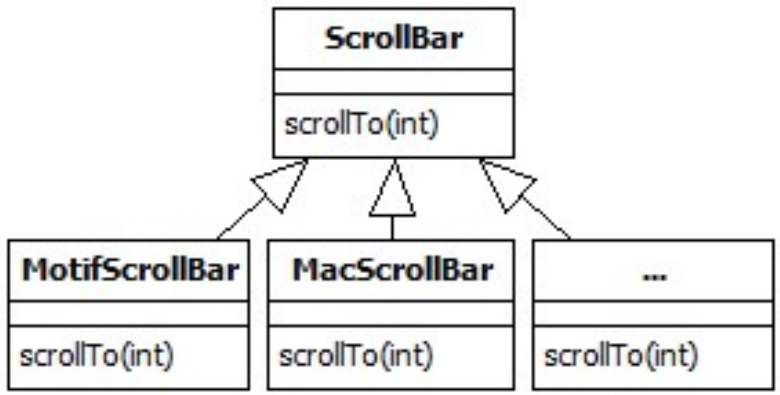
- Abstraient le processus d'instanciation (vs l'opérateur `new`)
  - En encapsulant la connaissance des classes concrètes que le système utilise
  - En cachant la manière dont les instances de ces classes sont crées et assemblées
  - Tout ce que le système connaît de ces objets est leurs interfaces telles que définies par des classes abstraites
- Offrent une grande flexibilité en *ce qui* doit être créé, *qui* doit le créer, *comment* le créer et *quand*
- Permettent de configurer statiquement ou dynamiquement un système avec des objets "produits"

Problème : *look-and-feel*

- Nous souhaitons que l'éditeur de texte supporte différents standards de *look-and-feel*
  - Apparence des barres de défilement, menus, bordures, *etc.*
  - Que faut-il écrire dans le code pour créer ces différents composants d'interface graphique (*widgets*) selon le *look-and-feel* courant :

```
ScrollBar sb = new ?
```

- Autrement dit, disposant d'une hiérarchie de classes, comment créer de manière flexible les instances ?
- Que proposez-vous comme conception ?



Pas très bon

- Très mauvais :

```
ScrollBar sb = new MotifScrollBar(...);
```

- Un peu meilleur :

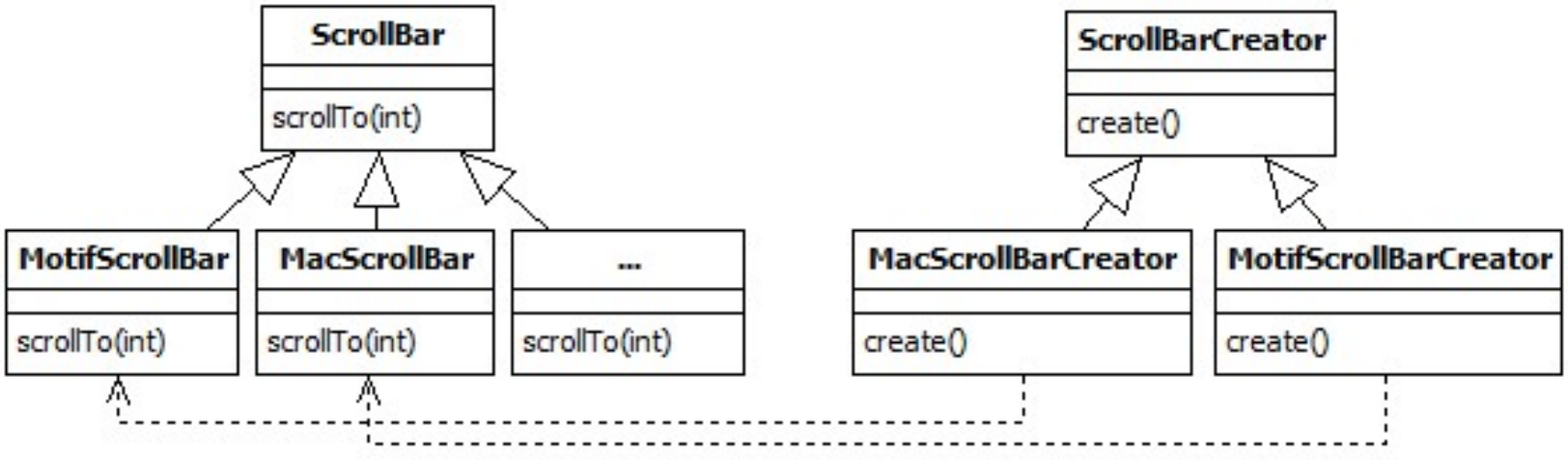
```
ScrollBar sb;
if (style == MOTIF) {
  sb = new MotifScrollBar();
} else if (style == ...) {
  sb = new ...
}
```

  - Conditionnels similaires pour les menus, les bordures, *etc.*

- Autres solutions :
  - Méthode de classe (pour les riches)
  - Introspection quand cela est permis

Abstraire la création d'objet

- Encapsuler ce qui varie dans une classe (ici, la création d'objet)
  - Pouvoir créer différents menus, barres de défilement, *etc.* selon le *look-and-feel* courant
- Définir une classe `ScrollBarCreator` avec une méthode abstraite d'instance `create`
- Définir des sous-classes de `ScrollBarCreator` pour chaque *look-and-feel*, redéfinissant la méthode `create`

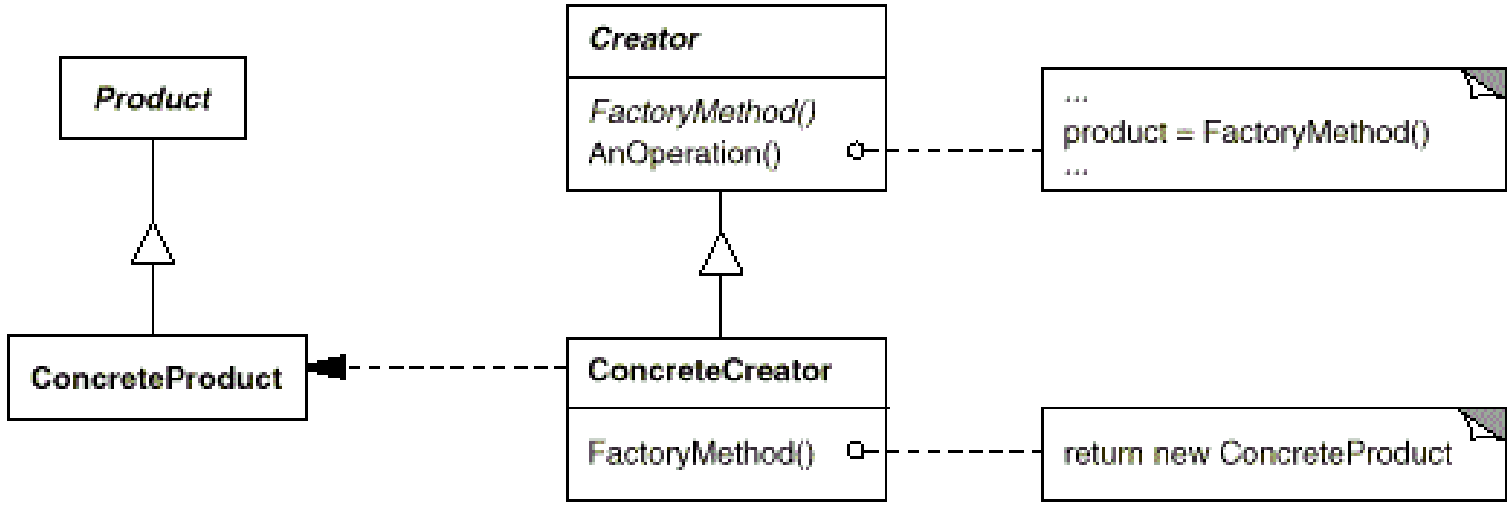


Méthode de fabrique (*Factory Method*)

- Intention
  - Définit une interface pour la création d'un objet, mais le choix de la classe concrète de l'objet est délégué à des sous-classes
- Utilisation
  - Une classe ne peut pas anticiper la classe des objets qu'elle doit créer
  - Une classe attend de ses sous-classes qu'elles spécifient les objets qu'elle crée

Méthode de fabrique (*Factory Method*)

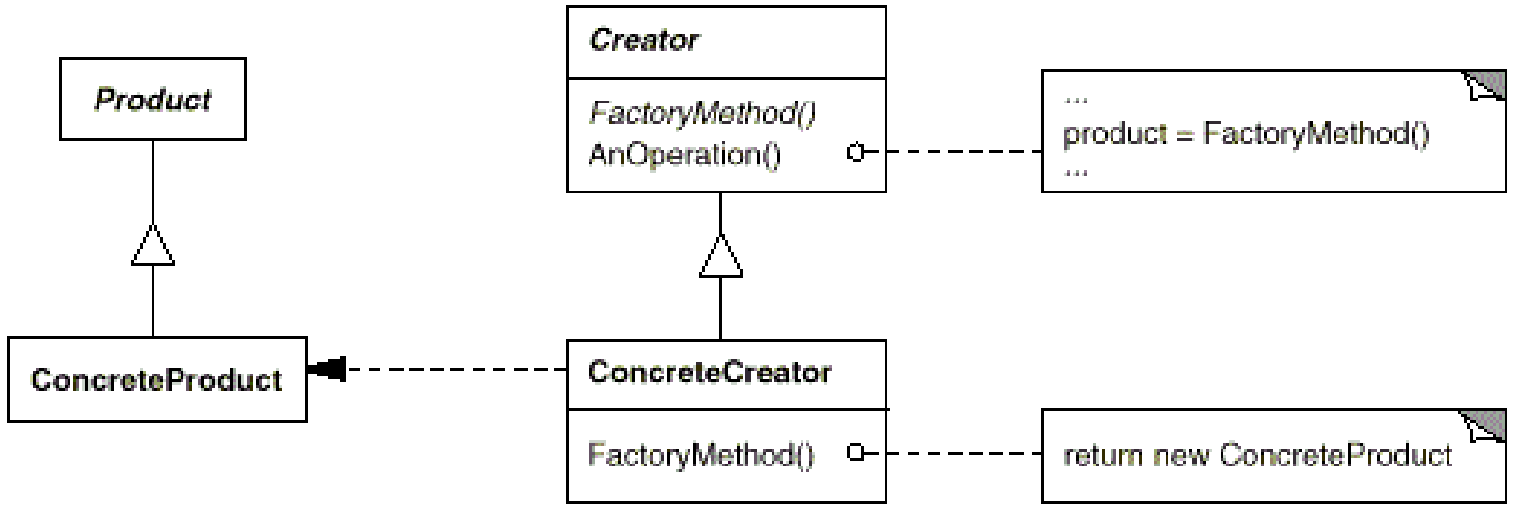
- Structure



- Participants
  - `Product` définit l'interface des objets que la méthode de fabrique crée
  - `ConcreteProduct` implémente l'interface `Product`
  - `Creator` déclare la méthode de fabrique qui retourne un objet de type `Product`
  - `ConcreteCreator` redéfinit la méthode de fabrique pour retourner une instance d'un `ConcreteProduct`

Méthode de fabrique (*Factory Method*)

- Structure



- Collaboration
  - La classe `Creator` compte sur ses sous-classes pour implémenter la méthode de fabrique afin qu'elle retourne une instance du `ConcreteProduct` approprié
  - La classe `Creator` est écrite sans savoir quelle véritable classe `ConcreteProduct` sera instanciée
    - ★ Déterminé seulement à l'exécution par la sous-classe `ConcreteCreator` qui est instanciée et utilisée par l'application (client)
    - ★ Mais ne signifie pas que cette sous-classe décide à l'exécution de la classe `ConreteProduct` à instancier

Méthode de fabrique (*Factory Method*)

- Avantages
  - En évitant de spécifier le nom de la classe concrète et les détails de son instanciation, le code client devient plus flexible et réutilisable
  - Le client est uniquement dépendant de l'interface `Product` et peut fonctionner avec n'importe quelle classe `ConcreteProcduct` qui implémente cette interface
- Inconvénients
  - Le client peut avoir à créer une sous-classe de la classe `Creator`, juste pour instancier un `ConcreteProduct` particulier

Méthode de fabrique (*Factory Method*)

- Implémentation
  - La classe `Creator` peut être :
    - ★ Abstraite (*doit* être spécialisée) et n'implémente pas la méthode de fabrique
    - ★ Concrète (*peut* être spécialisée) et fournit une implémentation par défaut de la méthode de fabrique
  - Une méthode de fabrique devrait-elle pouvoir créer différents types de produits ?
    - ★ Si oui, la méthode de fabrique prend un paramètre (possiblement utilisé dans un bloc `if-else`) pour décider quel objet créer

Retour à notre exemple

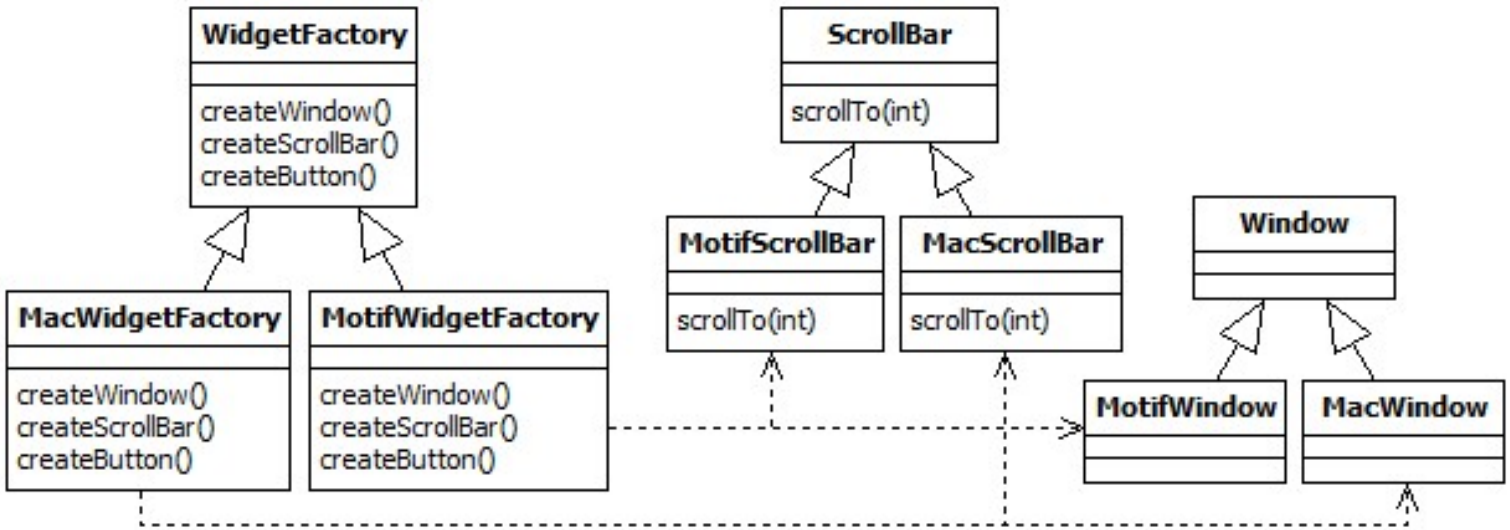
```
public abstract class ScrollBarCreator {
  public abstract ScrollBar create();
}

public class MotifScrollBarCreator extends ScrollBarCreator {
  public ScrollBar create() {
    return new MotifScrollBar();
  }
}

public class MacScrollBarCreator extends ScrollBarCreator {
  public ScrollBar create() {
    return new MacScrollBar();
  }
}
...
ScrollBarCreator sbc = new MotifScrollBarCreator();
ScrollBar sb = sbc.create();
```

Abstraire la création d'objet - suite

- La méthode de fabrique permet de changer une méthode de classe en une méthode d'instance
- Mais ce n'est pas suffisant pour notre problème
  - Ajout d'une hiérarchie de classes auxiliaires
  - Permet de gérer et séparer cette construction des classes utilitaires

Abstraire la création d'objet - suite

- Définir une classe `WidgetFactory`
  - Abstrait la création d'une famille d'objets
    - ★ Une méthode `create` pour créer chaque *widget*
  - Différentes instances fournissent des implémentations alternatives à cette famille
    - ★ Des sous-classes de `WidgetFactory` pour chaque *look-and-feel*
  - Un objet `WidgetFactory` pour créer les différents *widgets* du *look-and-feel* courant
    - ★ Variable globale
    - ★ Peut être changé à l'exécution

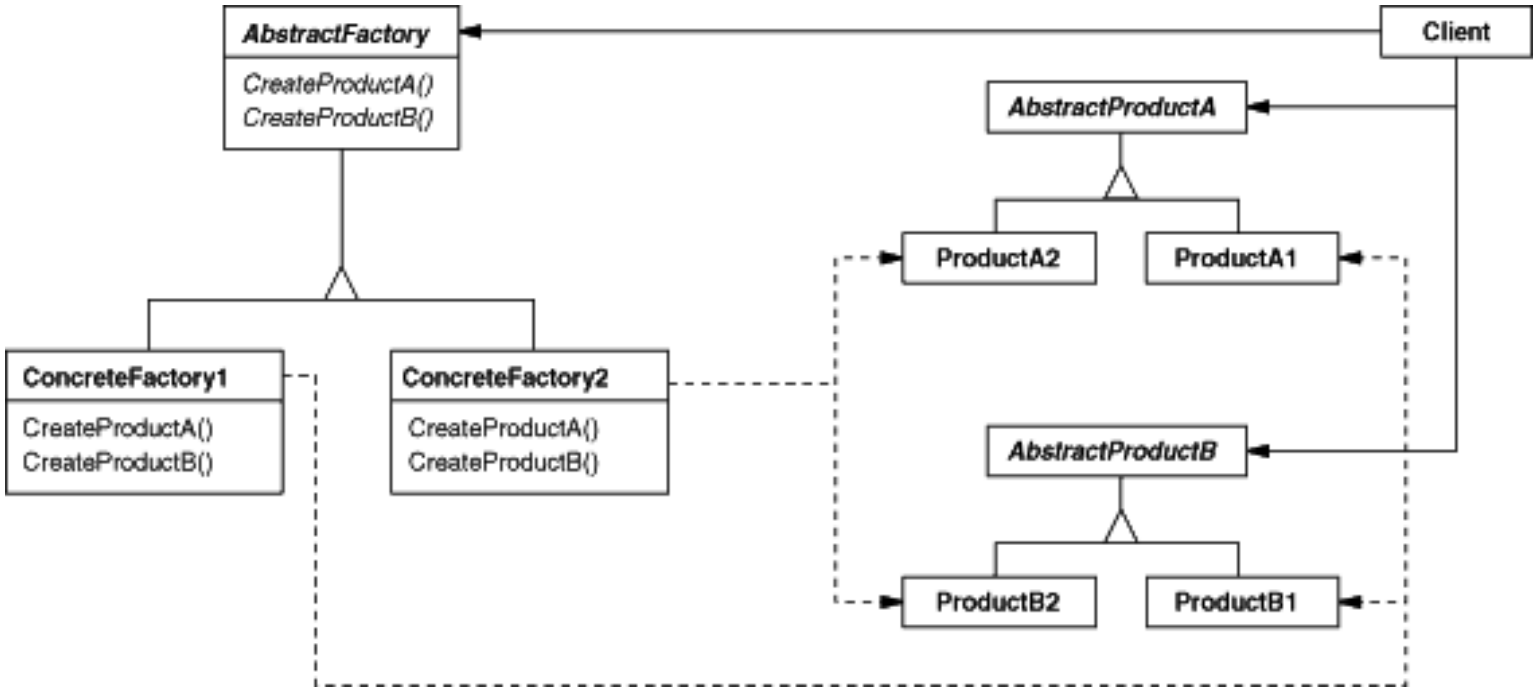


Fabrique abstraite (*Abstract Factory*)

- Intention
  - Fournit une interface pour créer des familles d'objets liés sans spécifier leurs classes concrètes
- Utilisation
  - Un système doit être indépendant de comment ses produits sont crées, composés et représentés
  - Un système doit être configuré avec l'une des multiples familles de produits
  - Une famille de produits est conçue pour être utilisée ensemble et il est nécessaire de respecter cette contrainte

Fabrique abstraite (*Abstract Factory*)

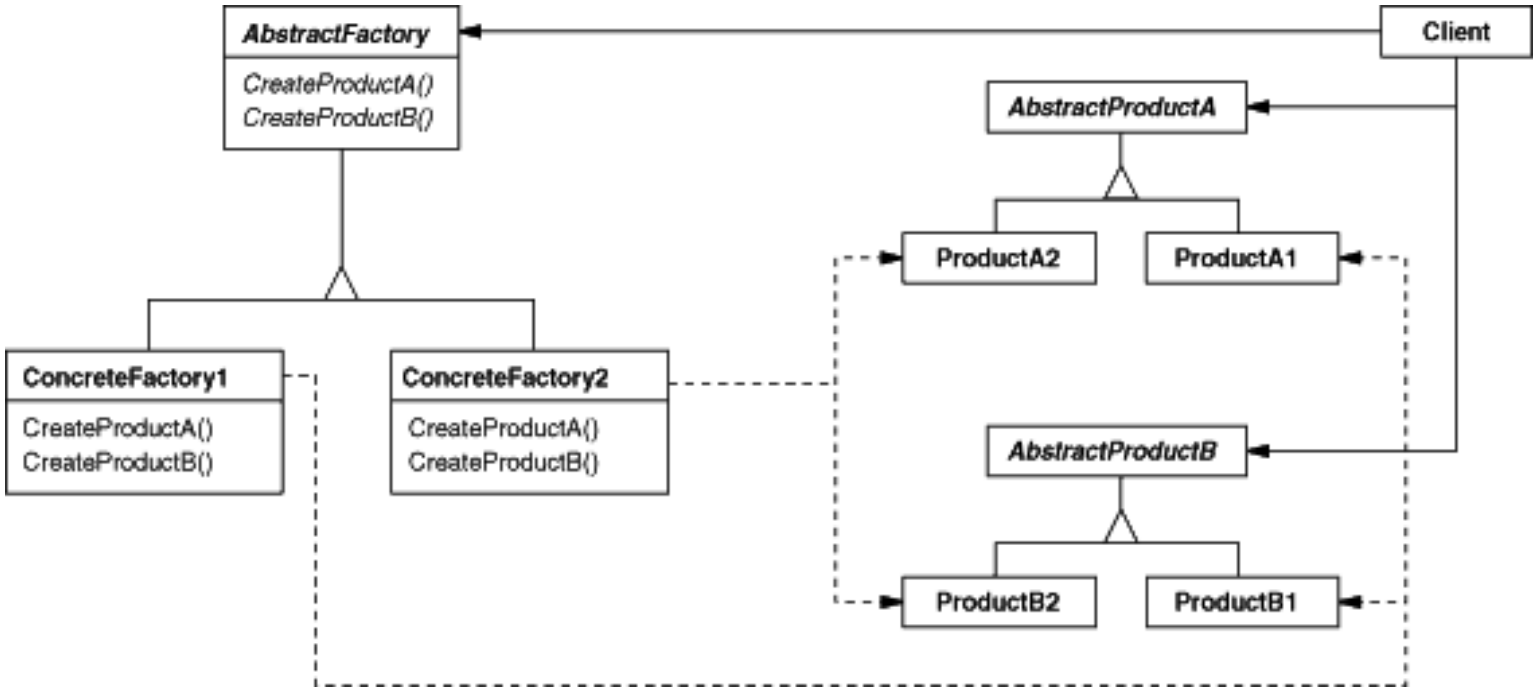
- Structure



- Participants
  - `AbstractFactory` déclare une interface pour des opérations qui créent des objets de produit abstrait
  - `ConcreteFactory` implémente les opérations pour créer des objets de produit concret

Fabrique abstraite (*Abstract Factory*)

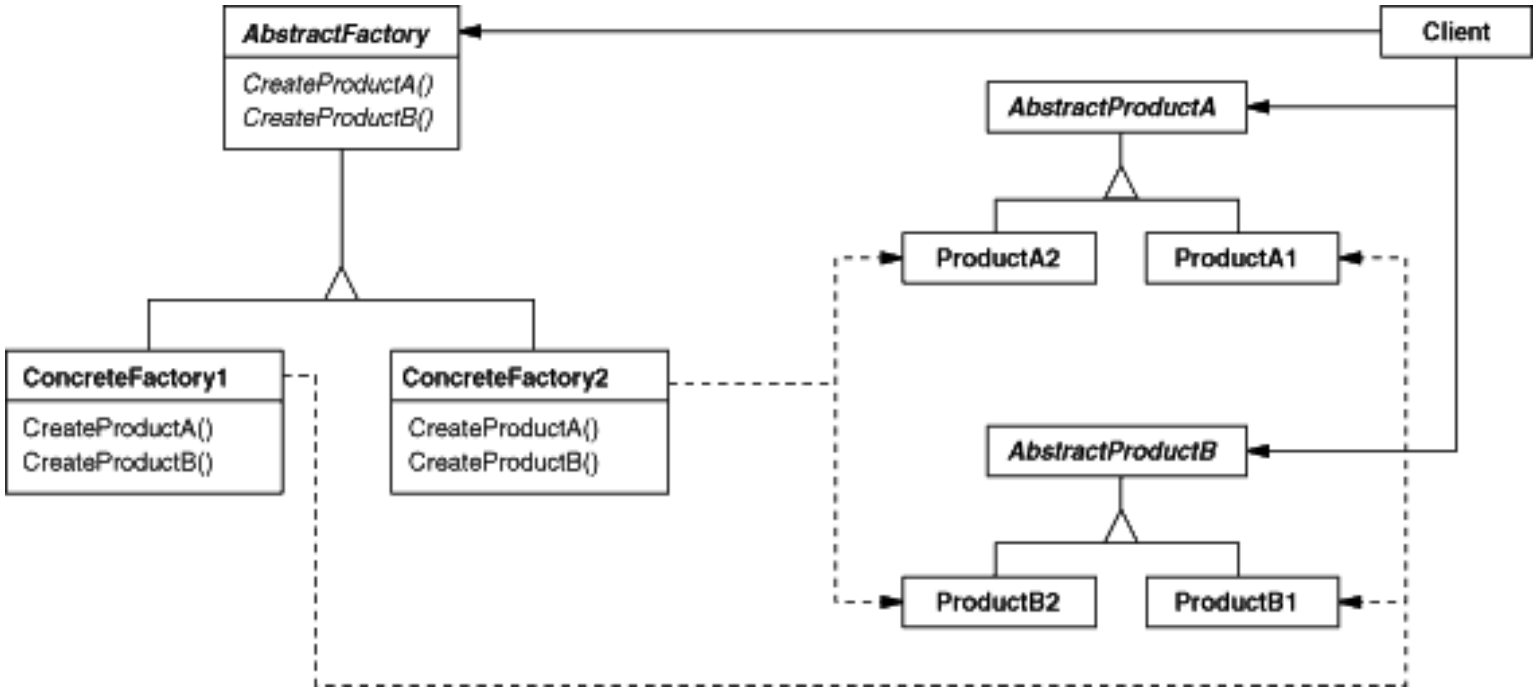
- Structure



- Participants
  - `AbstractProduct` déclare une interface pour un type de produit
  - `ConcreteProduct` implémente l'interface `AbstractProduct` et définit un produit devant être créé par la fabrique concrète correspondante
  - `Client` utilise uniquement les interfaces déclarées par les classes `AbstractFactory` et `AbstractProduct`

Fabrique abstraite (*Abstract Factory*)

- Structure



- Collaboration
  - Cette fabrique concrète crée des produits ayant une implémentation particulière
  - Pour créer des produits différents, les clients doivent utiliser une fabrique concrète différente
  - La classe `AbstractFactory` défère la création des produits à ses sous-classes concrètes

Fabrique abstraite (*Abstract Factory*)

- Avantages
  - Isole les clients des classes concrètes, car ils manipulent seulement des interfaces abstraites
  - Rend facile le changement des familles de produits, car une fabrique concrète particulière supporte une famille complète de produits
  - Favorise la cohérence entre produits
    - ★ Impose l'utilisation de produits d'une seule et même famille
- Inconvénients
  - Difficile d'ajouter de nouveaux produits, car requiert de changer l'interface `AbstractFactory` et donc d'étendre l'interface de toutes les classes de fabrique concrète

Fabrique abstraite (*Abstract Factory*)

- Implémentation
  - En général, une seule instance d'une fabrique concrète particulière est nécessaire à l'exécution
    - ★ Utilisation du pattern Singleton
  - Une méthode de fabrique par type de produit (la surcharger pour spécifier les objets réels à créer)
  - S'il y a un grand nombre de familles de produits, la fabrique concrète peut être implémentée en utilisant le pattern Prototype
    - ★ Est initialisée avec un prototype de chaque produit de la famille et crée un nouveau produit par clonage de son prototype
    - ★ Élimine la nécessité d'une nouvelle classe de fabrique concrète pour chaque nouvelle famille de produits

Retour à notre exemple (1)

```
// Method 1: use factory methods
public abstract class WidgetFactory {
  public abstract Window createWindow();
  public abstract ScrollBar createScrollBar();
  public abstract Button createButton();
}

public class MotifWidgetFactory extends WidgetFactory {
  public Window createWindow() {return new MotifWindow();}
  public ScrollBar createScrollBar() {return new MotifScrollBar();}
  public Button createButton() {return new MotifButton();}
}
...
// The client code is the same no matter how the factory creates
// the product
WidgetFactory wf = new MotifWidgetFactory();
Window w = wf.createWindow();
ScrollBar sb = wf.createScrollBar();
Button b = wf.createButton();
```

Retour à notre exemple (2)

```
// Method 2: use factories
public abstract class WidgetFactory {
  protected WindowFactory windowFactory;
  protected ScrollBarFactory scrollBarFactory;
  protected ButtonFactory buttonFactory;
  public Window createWindow() {return windowFactory.createWindow();}
  public ScrollBar createScrollBar() {
    return scrollBarFactory.createScrollBar();
  }
  public Button createButton() {return buttonFactory.createButton();}
}

public class MotifWidgetFactory extends WidgetFactory {
  public MotifWidgetFactory() {
    windowFactory = new MotifWindowFactory();
    scrollBarFactory = new MotifScrollBarFactory();
    buttonFactory = new MotifButtonFactory();
  }
}
```

Retour à notre exemple (3)

```
// Method 3: use factories with no required subclasses
// (pure composition)
public class WidgetFactory {
  private WindowFactory windowFactory;
  private ScrollBarFactory scrollBarFactory;
  private ButtonFactory buttonFactory;
  public WidgetFactory(WindowFactory wf, ScrollBarFactory sbf,
                       ButtonFactory bf) {
    windowFactory = wf;
    scrollBarFactory = sbf;
    buttonFactory = bf;
  }
  ... // setters
  public Window createWindow() {return windowFactory.createWindow();}
  public ScrollBar createScrollBar() {
    return scrollBarFactory.createScrollBar();
  }
  public Button createButton() {return buttonFactory.createButton();}
}
```

Méthode de fabrique vs Fabrique abstraite

- Méthode de fabrique
  - Pattern de création au niveau classe
  - Utilise l'héritage (sous-classe) pour décider de l'objet à instancier
- Fabrique abstraite
  - Pattern de création au niveau objet
  - Utilise la composition pour déléguer la responsabilité de l'instanciation d'objets à un autre objet
    - ★ L'objet délégué utilise souvent des méthodes de fabrique pour effectuer l'instanciation

Problème : unicité d'une fabrique

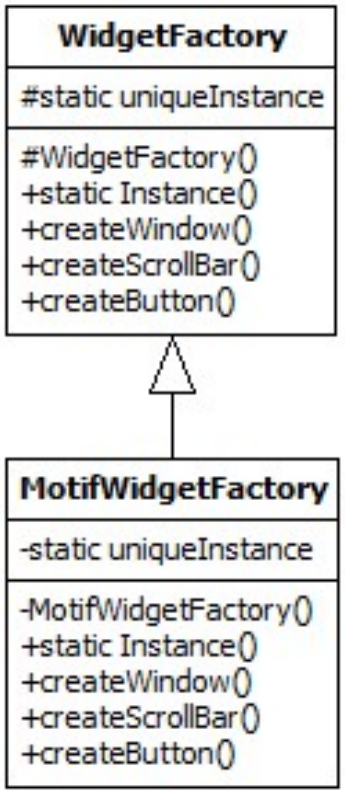
- Nous avons seulement besoin d'une seule instance de fabrique concrète pour créer la famille de *widgets* correspondant au *look-and-feel* choisi
  - Maintient de la cohérence graphique
- Autrement dit, comment garantir qu'il ne peut exister qu'une seule instance d'une classe, facilement accessible ?
- Que proposez-vous comme conception ?

Pas très bon

- Les variables globales permettent d'accéder à un objet n'importe où dans le programme mais n'empêche pas des instanciations multiples de cet objet
- Une constante dans une interface ?
  - Est très souvent un détail d'implémentation (vs service)
  - La promeut dans l'API publique de la classe implémentant l'interface
  - Anti-pattern
- Qualificatif `static`
  - Solution un peu rigide

Diagramme de classes

- La solution passe par la création d'une instance, plus flexible et plus évolutive grâce à l'héritage

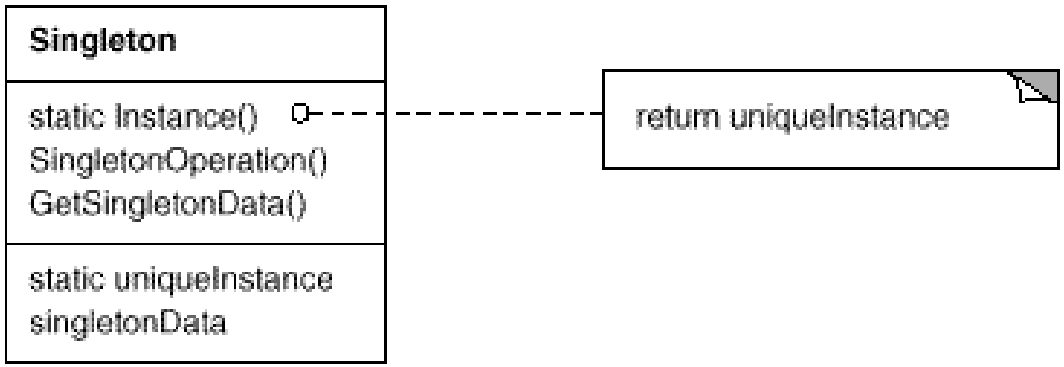


Singleton (*Singleton*)

- Intention
  - Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance
- Utilisation
  - Il ne doit y avoir exactement qu'une seule instance d'une classe
  - Cette instance unique doit être facilement accessible aux clients
  - Cette instance unique doit être extensible par héritage, et les clients doivent pouvoir utiliser cette instance étendue sans modifier leur code

Singleton (*Singleton*)

- Structure



- Participants
  - Une seule classe concrète (aucune classe abstraite/interface)
  - `Singleton` définit une méthode de classe `Instance` permettant d'accéder à son unique instance
    - ★ Peut être chargée de créer sa propre instance unique
    - ★ `uniqueInstance` est une variable de classe de type `Singleton`
    - ★ `singletonData` est une variable d'instance (état)
    - ★ `SingletonOperation` est une méthode d'instance (comportement)
- Collaboration
  - Les clients accèdent à l'instance d'un Singleton *via* la méthode de classe `Instance`

Singleton (*Singleton*)

- Avantages
  - Permet un accès contrôlé à une unique instance
  - Permet la réduction de l'espace des noms (vs variables globales)
  - Peut être étendu par héritage
  - Permet un nombre variable d'instances (toutes invisibles aux clients)
  - Est plus flexible que les méthodes de classe

Singleton (*Singleton*)

- Implémentation
  - Le constructeur est privé/protégé
    - ★ Évite les créations intempestives
  - L'instance unique de la classe est stockée dans une variable statique privée/protégée
  - Une méthode publique statique de classe (`Instance`)
    - ★ Crée l'instance au premier appel (*lazy instantiation*)
    - ★ Retourne cette instance
  - Héritage de la classe `Singleton`
    - ★ Soit, la méthode `Instance` détermine la sous-classe à instancier (*via* un argument ou une variable d'environnement)
    - ★ Soit, chaque sous-classe fournit une méthode de classe `Instance`
  - La concurrence peut compliquer les choses...
    - ★ *Thread safe* (`synchronised`/*eager instantiation*)

Implémentation

```
public class Singleton {

  private static Singleton uniqueInstance = null;

  public static Singleton Instance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton(); // Lazy instanciation
    }
    return uniqueInstance;
  }

  // No client can instantiate a Singleton object
  private Singleton() {...}
  ...
}
...
Singleton s = Singleton.Instance();
```

Implémentation *thread safe*

```
// The singleton instance is created in a static initializer
// This is guaranteed to be thread safe
// Alternative: make the Instance method synchronized (expensive!)
public class Singleton {

  // Eager instantiation
  private static Singleton uniqueInstance = new Singleton();

  public static Singleton Instance() {
    return uniqueInstance;
  }

  // No client can instantiate a Singleton object
  private Singleton() {...}
  ...
}
...
Singleton s = Singleton.Instance();
```

Retour à notre exemple (1)

```
//Method 1: the Instance method determine the subclass to instantiate
public abstract class WidgetFactory {
  private static WidgetFactory uniqueInstance = null;
  public static WidgetFactory Instance()
    if (uniqueInstance == null) {
      return Instance("motif"); // As default
    return uniqueInstance;
  }
  public static WidgetFactory Instance(String str) {
    if (uniqueInstance == null) {
      if (str.equals("motif")) {
        uniqueInstance = new MotifWidgetFactory();
      } else if (...) {...}
    }
    return uniqueInstance;
  }
  // Constructors cannot be private here
  protected WidgetFactory() {...}
}
// Client code to create factory the first time
WidgetFactory factory = WidgetFactory.Instance("motif");
// Client code to access the factory
WidgetFactory factory = WidgetFactory.Instance();
```

Retour à notre exemple (1)

- Les constructeurs des sous-classes (*e.g.*, `MotifWidgetFactory`) ne peuvent pas être privés, car `WidgetFactory` doit pouvoir les instancier
- Donc, des clients pourraient potentiellement créer d'autres instances de ces sous-classes
- La méthode `Instance(String)` viole le principe *Ouvert-Fermé*, car elle doit être modifiée pour chaque nouvelle sous-classe de `WidgetFactory`
- Nous pourrions utiliser le nom des classes comme argument de la méthode, donnant ainsi un code plus simple :

```
public static WidgetFactory Instance(String str) {
  if (uniqueInstance == null) {
    uniqueInstance = Class.forName(str).newInstance();
  }
  return uniqueInstance;
}
```

Retour à notre exemple (2)

```
// Method 2: each subclass provide a static Instance method
public abstract class WidgetFactory {
  protected static WidgetFactory uniqueInstance = null;
  public static WidgetFactory Instance() {
    return uniqueInstance;
  }
  // Constructor cannot be private here
  protected WidgetFactory() {...}
}
public class MotifWidgetFactory extends WidgetFactory {
  private static WidgetFactory uniqueInstance = null;
  public static WidgetFactory Instance() {
   if (uniqueInstance == null)
      uniqueInstance = new MotifWidgetFactory();
    return uniqueInstance;
  }
  // Private subclass constructor!
  private MotifWidgetFactory() {...}
}
// Client code to create factory the first time
WidgetFactory factory = MotifWidgetFactory.Instance();
// Client code to access the factory
WidgetFactory factory = WidgetFactory.Instance();
```

Retour à notre exemple (2)

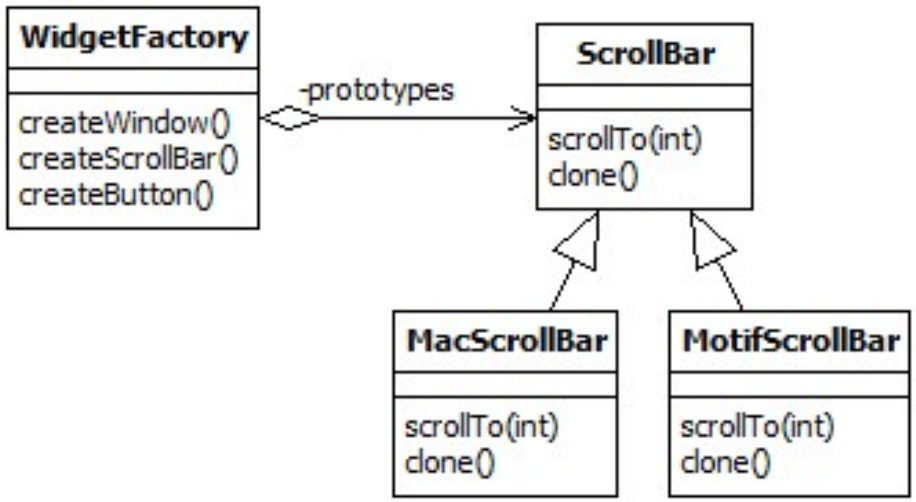
- Chaque sous-classe est une implémentation d'une classe Singleton
- Les constructeurs des sous-classes sont maintenant privés
  - Seulement une instance peut être créée
- Un client peut obtenir une référence `null` s'il invoque `WidgetFactory.Instance()` avant que l'unique instance de la sous-classe soit d'abord créée
- `uniqueInstance` est maintenant protégée

Problème : réduction du nombre de fabriques

- Nous souhaitons éviter d'avoir un grand nombre de sous-classes (*i.e.*, fabriques concrètes) pour créer les différentes familles de *widgets*
- Autrement dit, comment abstraire la création d'objet en évitant de faire proliférer le nombre de classes dans le système ?
- Que proposez-vous comme conception ?

Diagramme de classes

- L'objet fabrique et le prototype sont le même objet

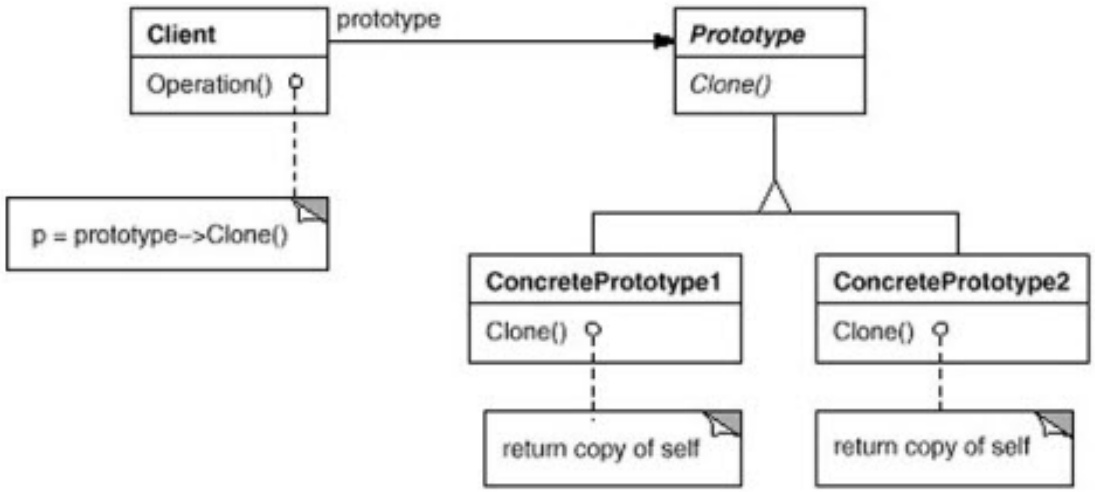


Prototype (*Prototype*)

- Intention
  - Spécifie le type des objets à créer en utilisant une instance (le prototype) et crée de nouveaux objets par copie de ce prototype (clonage)
- Utilisation
  - Un objet est plus coûteux à créer qu'à dupliquer (cloner)
  - Les classes à instancier sont spécifiées à l'exécution (*e.g.*, par chargement dynamique)
  - La construction d'une hiérarchie de fabriques parallèle à une hiérarchie de produits doit être évitée
  - Les instances d'une classe peuvent seulement avoir un petit nombre de combinaisons différentes d'état ou diffèrent uniquement par le comportement (le prototype sert de modèle)

Prototype (*Prototype*)

- Structure



- Participants
  - `Prototype` déclare une interface pour se cloner
  - `ConcretePrototype` implémente une opération pour se cloner
  - `Client` crée un nouvel objet par clonage d'un prototype
- Collaboration
  - Les clients demandent à un prototype de se cloner

Prototype (*Prototype*)

- Mêmes avantages et inconvénients que les patterns Fabrique abstraite et Monteur
- Avantages
  - Permet d'ajouter et supprimer des produits (prototypes) à l'exécution
  - Crée de nouveaux objets "différents" sans créer de classe
  - Facilite le chargement dynamique de classes
- Inconvénients
  - Peut être difficile d'implémenter l'opération de clonage
    - `Serializable` ?
    - Accès aux données ?
    - Références circulaires ?

Prototype (*Prototype*)

- Implémentation
  - Utilisation d'un gestionnaire de prototypes
    - ★ Stockage associatif, opération d'enregistrement et de suppression
  - Copie profonde vs copie superficielle
    - ★ Cloner un objet implique-t-il de cloner ses variables d'instance ou est-ce que le clone et l'original partagent les variables ?
  - Initialisation de l'état interne des clones

Retour à notre exemple (1)

```
public abstract class ScrollBar implements Cloneable {
   ...
}

public class MotifScrollBar extends ScrollBar {
   ...
   public Object clone() throws CloneNotSupportedException() {
      MotifScrollBar msb = (MotifScrollBar) super.clone();
      ...
      return msb;
   }
}
...
ScrollBar sb = scrollBarPrototype.clone();
sb.initialize(...);
```

Retour à notre exemple (2)

```
//Concrete factory using prototypes
public class WidgetFactory {
  private Window windowPrototype;
  private ScrollBar scrollBarPrototype;
  private Button buttonPrototype;
  public WidgetFactory(WindowPrototype wp, ScrollBarPrototype sbp,
                       ButtonPrototype bp) {
    windowPrototype = wp;
    scrollBarPrototype = sbp;
    buttonPrototype = bp;
  }
  ... // setters
  public Window createWindow() {...}
  public ScrollBar createScrollBar() {
    return (ScrollBar) scrollBarPrototype.clone();
  }
  public Button createButton() {
    Button b = (Button) buttonPrototype.clone();
    b.initialize(...);
    return b;
  }
}
```

Prototype vs Fabrique abstraite

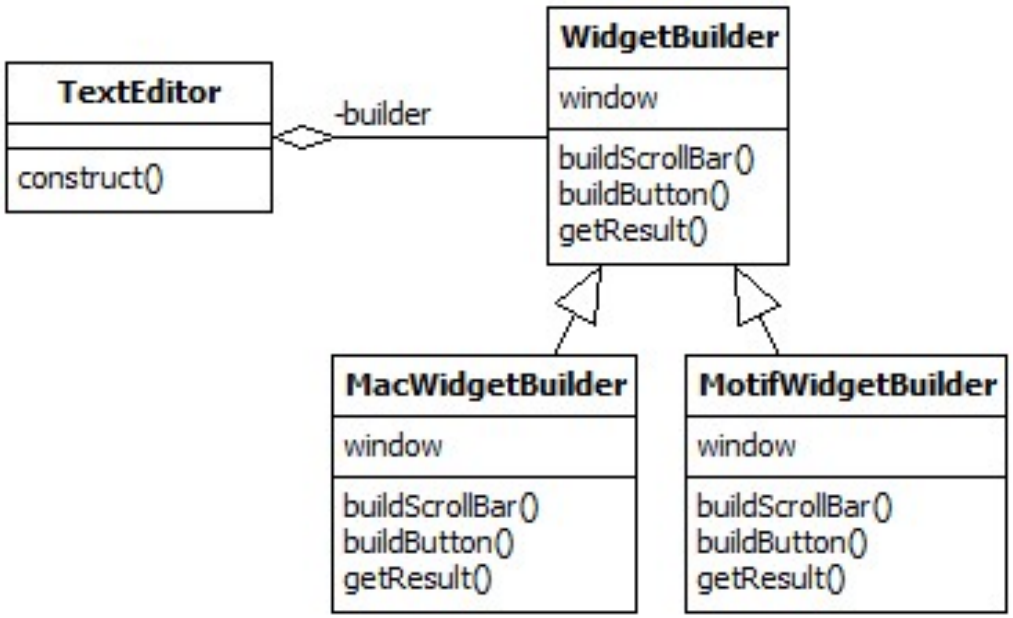
- En concurrence d'une certaine manière
- Une fabrique abstraite peut stocker un ensemble de prototypes à partir duquel elle va créer des produits par clonage

Problème : séparation construction/représentation

- Nous souhaitons capturer la manière dont les *widgets* sont créés et assemblés pour former la fenêtre graphique correspondante au *look-and-feel* choisi, ainsi que cacher sa représentation interne
  - Réutilisation d'une partie du processus de construction
- Autrement dit, comment séparer le processus de construction d'un objet complexe de ses représentations ?
- Que proposez-vous comme conception ?

Diagramme de classes

- Un monteur est passé en paramètre de la méthode `construct`, simplifiant son code

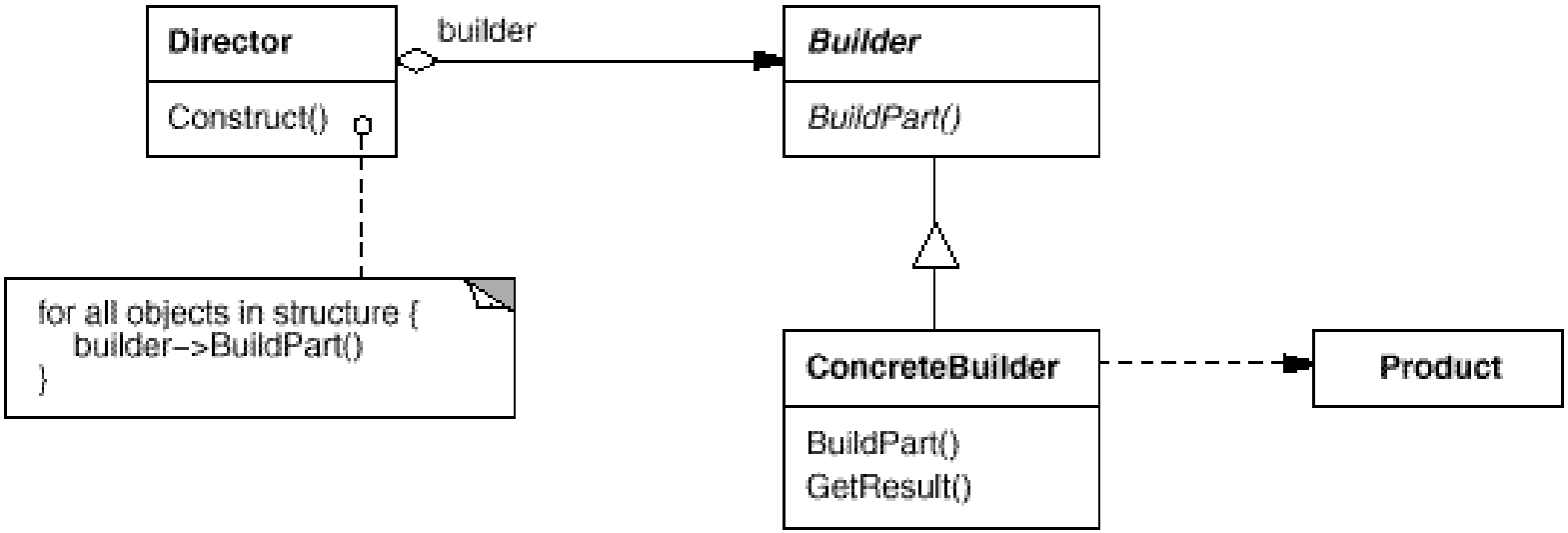


Monteur (*Builder*)

- Intention
  - Dissocie la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction permette de créer différentes représentations de cet objet
- Utilisation
  - L'algorithme pour créer un objet complexe doit être indépendant des parties qui composent l'objet et de comment elles sont assemblées (*i.e.*, leurs relations)
  - Le processus de construction doit permettre différentes représentations de l'objet construit
  - Construction de structures composites

Monteur (*Builder*)

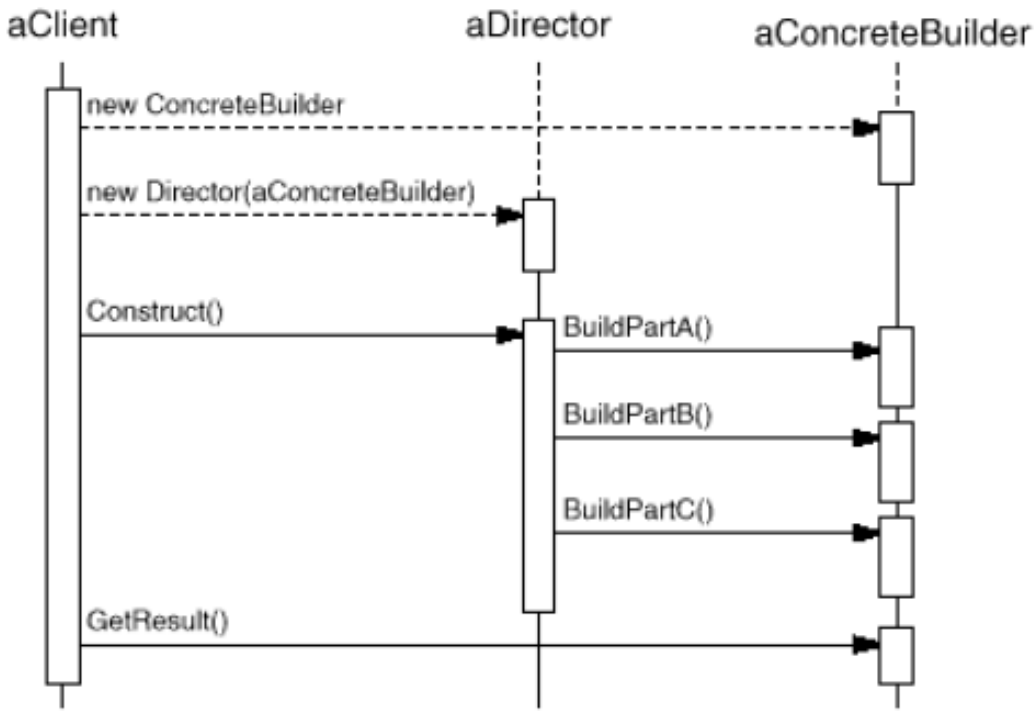
- Structure



- Participants
  - `Builder` spécifie l'interface pour créer les parties d'un objet `Product`
  - `ConcreteBuilder` construit et assemble les parties du produit en implémentant l'interface `Builder`, et fournit une interface pour récupérer le produit final
  - `Director` construit un objet en utilisant l'interface `Builder`
  - `Product` représente l'objet complexe en cours de construction

Monteur (*Builder*)

- Collaboration



1. Le client crée un objet de type `Director` et le configure avec le monteur désiré
2. Le directeur notifie le monteur chaque fois qu'une partie du produit doit être construite
3. Le monteur crée et ajoute les parties au produit
4. Le client récupère le produit depuis le monteur

Monteur (*Builder*)

- Avantages
  - Permet de faire varier la représentation interne d'un produit
    - ★ Implémentation des produits et de leurs composants cachée au directeur
    - ★ La construction d'un autre objet revient à définir un nouveau monteur
  - Isole le code de construction et de représentation du reste de l'application
    - ★ Modularité améliorée : les monteurs sont indépendants
  - Permet un meilleur contrôle du processus de construction et sur la structure interne du produit résultant
    - ★ Construction étape par étape sous le contrôle d'un directeur

Monteur (*Builder*)

- Implémentation
  - Quelles parties du processus du construction reste dans le directeur et laquelle est déléguée au monteur ?
    - ★ Trouver ce qui varie, l'extraire et le cacher
  - Interface d'assemblage et de construction
    - ★ Assez générale pour permettre la construction de produits pour toutes sortes de monteurs concrets
    - ★ Accès à des parties construites antérieurement : le monteur retournera des objets au directeur qui les lui repassera pour construire la suite

Retour à notre exemple (1)

```
public abstract class WidgetBuilder {
  protected Window window;
  public abstract void buildScrollBar();
  public abstract void buildButton();
  public abstract Window getResult();
}
public class MotifWidgetBuilder extends WidgetBuilder {
  public MotifWidgetBuilder() {window = new Window();}
  public void buildScrollBar() {
    ScrollBar sb = new MotifScrollBar();
    ... // add sb to window + others treatments
  }
  public void buildButton() {
    Button b = new MotifButton();
    ... // add b to window + others treatments
  }
  public Window getResult() {return window;}
}
...
WidgetBuilder wb = new MotifWidgetBuilder();
wb.buildScrollBar();
...
Window w = wb.getResult();
```

Retour à notre exemple (2)

```
// Builder using factories
public class WidgetBuilder {
  protected WidgetFactory factory;
  protected Window window;
  public WidgetBuilder(WidgetFactory wf) {factory = wf;}
  public void buildScrollBar() {
    ScrollBar sb = factory.makeScrollBar();
    ... // add sb to window + others treatments
  }
  public void buildButton() {
    Button b = factory.makeButton();
    ... // add b to window + others treatments
  }
  public Window getResult() {return window;}
}
...
WidgetBuilder wb = new WidgetBuilder(new MotifWidgetFactory());
wb.buildScrollBar();
...
Window w = wb.getResult();
```

Monteur vs Fabrique abstraite

- Monteur
  - Se focalise sur la construction d'un objet complexe étape par étape
  - Retourne le produit comme une étape finale
- Fabrique abstraite
  - Se focalise sur la notion de familles de produits (qu'ils soient simples ou complexes)
  - Retourne le produit immédiatement