Design Patterns
(patrons de conception)

GLO-3001
Architecture logicielle

Luc Lamontagne
Hiver2010

Contenu du module

- Introduction au concept de *design patterns*
- Patrons de base
- Classification par familles
  - De génération (*creational*)
  - De structuration (*structural*)
  - De comportement (*behavioral*)
- Patrons concurrents
- Autres patrons – Série POSA

Design Patterns

- Origine - travaux de Christopher Alexander
- Livre du *Gang of Four*
- Solutions abstraites qui fonctionnent bien dans différents contextes

  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 (GoF)

Design Patterns (suite)

- Notre but – maîtriser des patrons de conception réutilisables dans vos projets
  - Mémorisation
  - Compréhension
  - Application
- On étudie d'un point de vue Java
  - Plusieurs patrons déjà utilisés par le langage
- Autres langages et autres informations
  - voir *hillside.net* ou Wikipédia

Styles, patrons et idiomes

- Styles (*macro patrons*)
  - structure de haut niveau pour décrire un logiciel d'un point de vue global.
  - Par ex. modèle par couche (*layered*), *Model-view-Controller*
- Patrons (*micro patrons*)
  - correspond au *design patterns*
  - solutions abstraites applicables à différents contextes mais qui offrent les mêmes avantages chaque fois.
- Idiomes
  - décrit des bonnes pratiques liées:
    - à un formalisme de programmation (par exemple, la construction d'un bonne classe en OO)
    - à un langage en particulier (destruction d'objets en C++)
- Ce module porte sur les *micro patrons (design patterns)*.

Description d'un patron

- Nom
- Problème
  - Description du problème et de son contexte
  - Parfois une énumération des *vices* de conception
- Solution
  - Éléments de la conception et leurs relations
  - Responsabilités et collaboration
- Conséquences
  - Temps vs. espace mémoire
  - Problèmes liés au langage et à l'implantation

Catégories de patrons

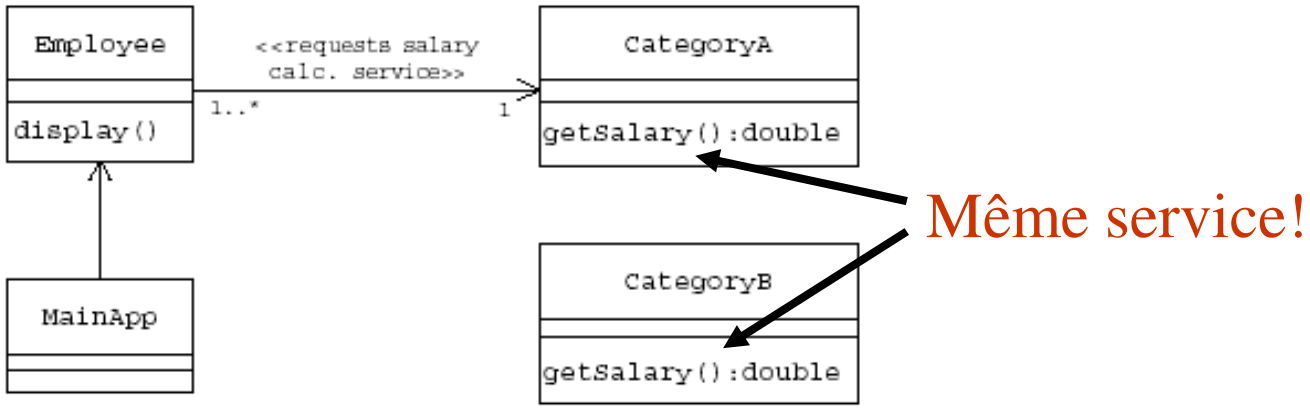
		Famille		
		Génération	Structuration	Comportement
Portée	Classe	Méthode *Factory*	Adapteur	Interpréteur Méthode *Template*
	Objet	*Factory* abstrait Prototype Singleton *Builder*	Adapteur *Bridge* Composite Décorateur Façade *Flyweight* *Proxy*	Chaîne de responsabilité Commande Itérateur Médiateur *Memento* Observateur État (state) Stratégie Visiteur

Patrons de base

Patterns	**Description**
Interface	Pour encapsuler un service offert par plusieurs classes
Classe abstraite parent	Pour permettre différentes définitions de services pour des classes similaires
Méthodes privées	Pour limiter l'accès au comportement interne d'une instance
Méthodes accesseurs	Pour contrôler l'accès aux propriétés (attributs) des instances d'une classes
Gestionnaire de constante	Pour regrouper dans une classe centralisée et faciliter la gestion des constantes d'une application
Objet immuable	S'assurer que l'état d'une instance ne peut pas être modifiée
Moniteur	Contrôler l'accès concurrent à une ressource partagée pour limiter les résultats non prévisibles

Interface

- Des objets (clients) invoquent les services d'autres objets (serveurs)
- Habituellement, on présume qu'un service est fourni par un seul type d'objet
- Service offert par plus d'un type d'objet ?
- Exemple :
  - Calcul du salaire de différents types d'employés



Interface

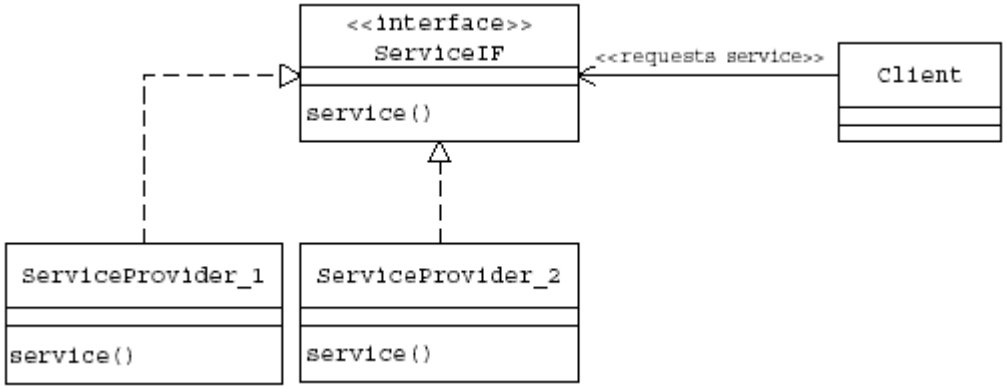
- Une solution possible
  - Un attribut qui indique la classe de l'instance
  - Lourd et difficile à maintenir pour un grand nombre de classes

```
public class CategoryA {
   double baseSalary;
   double OT;

   public CategoryA(double base, double overTime) {
       baseSalary = base;
      OT = overTime;
   }

   public double getSalary() {
      return (baseSalary + OT);
   }
}
```

```
public class Employee {
   CategoryA salaryCalculator;
   String name;
   public Employee(String s, CategoryA c) {
         name = s;
         salaryCalculator = c;
   }
   public void display() {
        System.out.println("Name=" + name);
        System.out.println("salary= " +
         salaryCalculator.getSalary());
   }
}
```

Interface

- Solution
  - On fait une abstraction du service
  - Chaque classe qui offre le service implémente cette abstraction
  - Éviter de modifier le code du client si un nouveau *serveur*
  - Fourni par le langage Java
  - Question: pourquoi ne pas utiliser l'héritage?



Interface

- En code

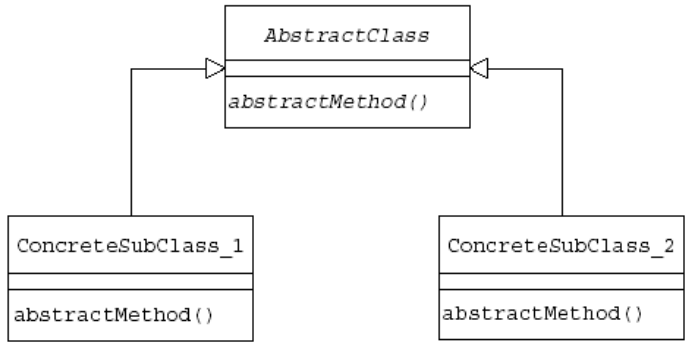
```
public interface SalaryCalculator (
        public double getSalary();
)

public class CategoryA implements SalaryCalculator {
   double baseSalary;
   double OT;

   public CategoryA(double base, double overTime) {
      baseSalary = base;
      OT = overTime;
   }

   public double getSalary() {
      return (baseSalary + OT);
   }
}
```

Classe abstraite parent

- Méthode abstraite
  - méthode déclarée, mais qui ne contient aucune implantation.
- Classe abstraite
  - classe avec une ou plusieurs méthodes abstraites.
- Méthode abstraite a plusieurs implantations possibles
  - partie variable du comportement de la classe abstraite.
- Une classe abstraite peut contenir des implantations d'autres méthodes
  - représente la partie invariable de la classe.

Classe abstraite parent

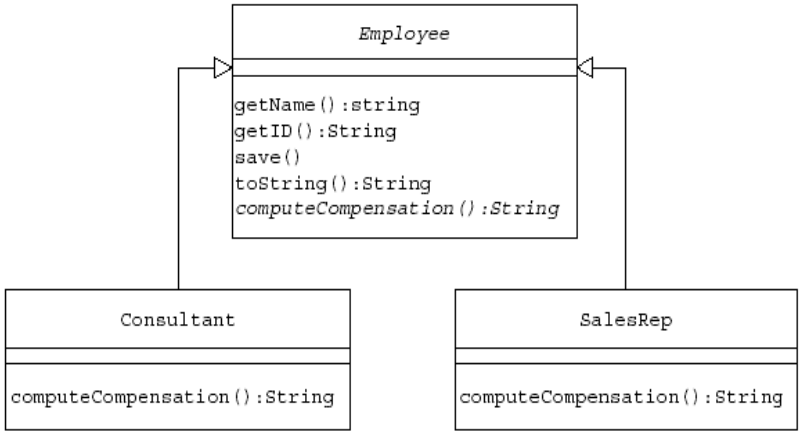
- Sous-classes
  - On en a plusieurs lorsque les méthodes abstraites sont implantées différemment
  - Doit implanter toutes les méthodes abstraites
  - Hérite des méthodes implantés par la classe parent
  - Seulement les sous-classes peuvent être instanciée
  - En Java, héritage multiple interdit.



Classe abstraite parent

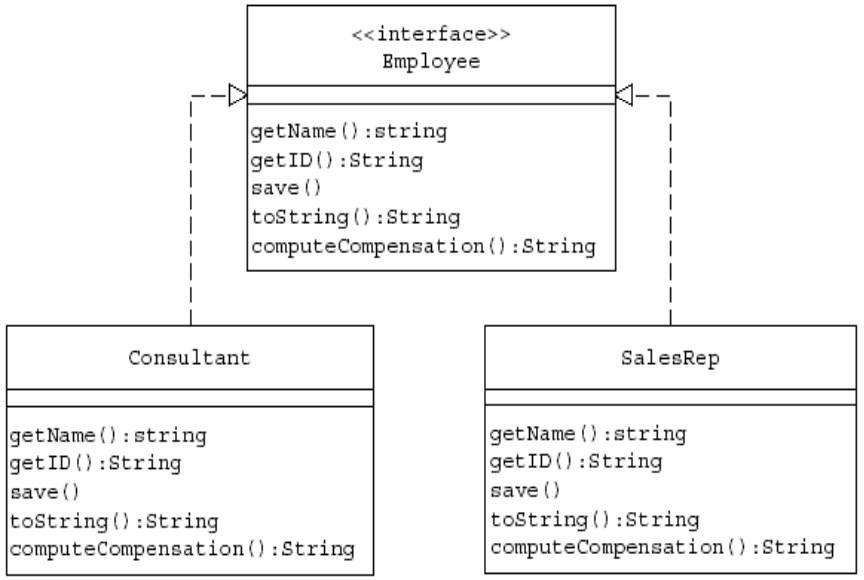
```
public abstract class Employee {
  String name;
  String ID;

  //invariable parts
  public Employee(String empName, String empID) {
    name = empName;
    ID = empID;
  }

  public String getName() {    return name;  }

  public String getID() {    return ID;  }

  public String toString() {
    String str = " Emp Name:: " + getName() + " EmpID:: " +     getID();
    return str;
  }

  public void save() {
    FileUtil futil = new FileUtil();
    futil.writeToFile("emp.txt",this.toString(), true, true);

  }

  //variable part of the behavior
  public abstract String computeCompensation();
}
```

Classe abstraite vs. interface



a) Classe abstraite

b) Interface



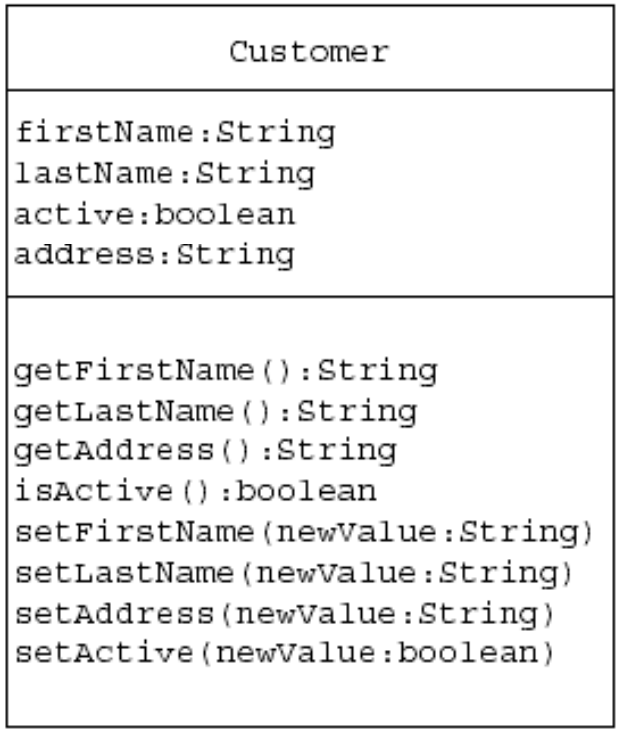
Redondance!

Méthodes accesseurs

- Le plus fréquent patron en OO
- État d'un objet
  - La valeur des variables de l'instance
  - publique (ex. Numéro étudiant) ou privée (ex: NAS)
- Clients externe → interdire l'accès direct aux variables de l'instance
  - On défini une méthode pour accéder chaque variable
- Changement d'état (source → cible)
  - Si impossible, aviser le client externe
  - Correspond à une émission (*throw*) d'exception
- Instance utilise ses propres méthodes accesseurs
  - Limite le nombre de modification et favorise la maintenabilité
  - Permet le changement de représentation interne (Vector → List)
- Cache si l'attribut est direct ou dérivé
  - Exemple : moyenne cummulative d'un étudiant

Méthodes accesseurs

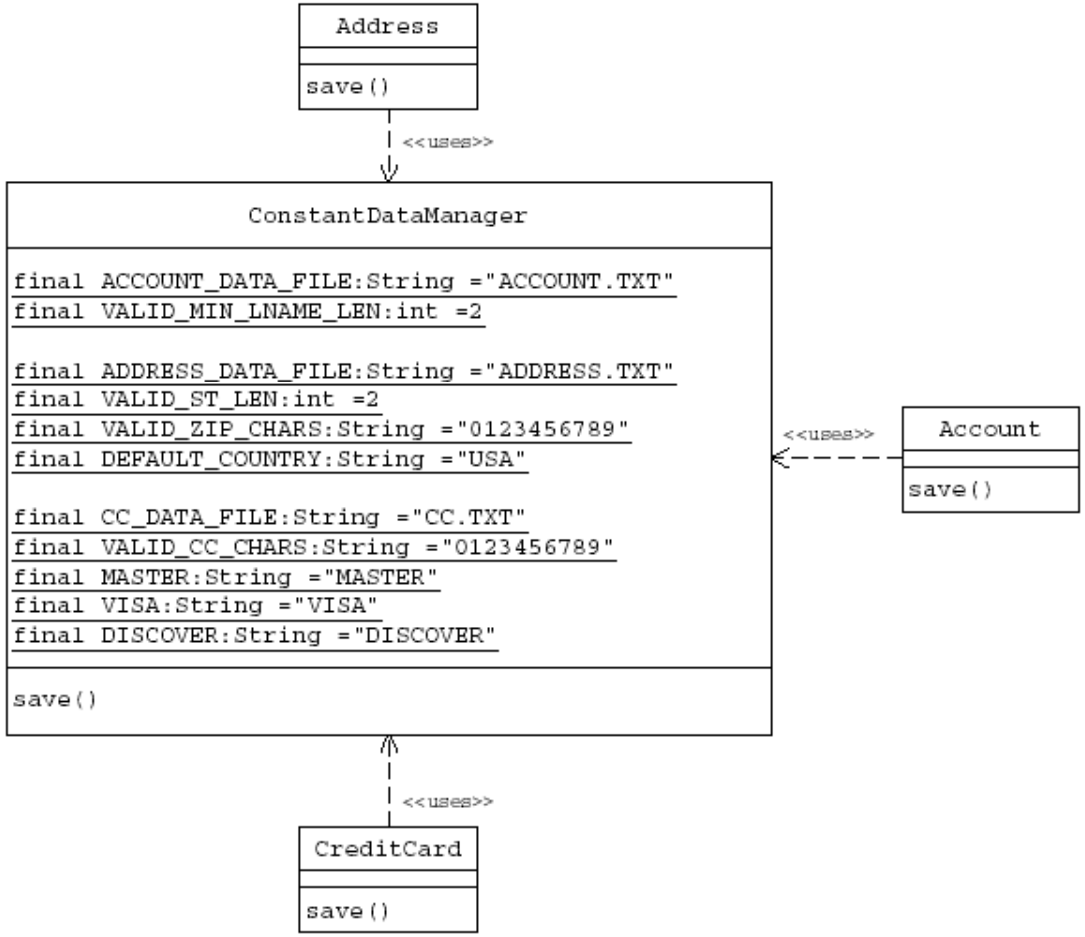
- Nomenclature
  - Variable non-booléenne
    - *getXXXX* : lecture de la valeur
    - *setXXXX* : modifie l'état
  - Variable booléenne
    - *isXXXX* :vérifie si vrai
    - *setXXXX* : modifie l'état



Gestionnaire de constantes

- Toutes les applications font usage de données
  - Valeurs constantes ou variables
- Ce patron facilite:
  - Le storage des constantes de l'application
  - L'accès par différentes classes de l'application
- Données cibles
  - Noms de fichiers, étiquettes de boutons, valeur minimim ou maximum, codes d'erreur, messages d'erreur…
- Centraliser dans une seule classe au lieu de répartir dans toute l'application
  - On n'a pas à chercher la classe à qui appartient la constante
  - Dépot centrale de données → Facilite la maintenance
  - Initialisation du système plus facile (lecture de paramètres dans un fichier)
- Classe ou interface?

Gestionnaire de constantes



Gestionnaire de constantes

```
public class ConstantDataManager {
        public static final String ACCOUNT_DATA_FILE = "ACCOUNT.TXT";
        public static final int VALID_MIN_LNAME_LEN = 2;
        public static final String ADDRESS_DATA_FILE = "ADDRESS.TXT";
        public static final int VALID_ST_LEN = 2;
        public static final String VALID_ZIP_CHARS = "0123456789";
        public static final String DEFAULT_COUNTRY = "USA";
        public static final String CC_DATA_FILE = "CC.TXT";
        public static final String VALID_CC_CHARS = "0123456789";
        public static final String MASTER = "MASTER";
        public static final String VISA = "VISA";
        public static final String DISCOVER = "DISCOVER";
}
```

constante

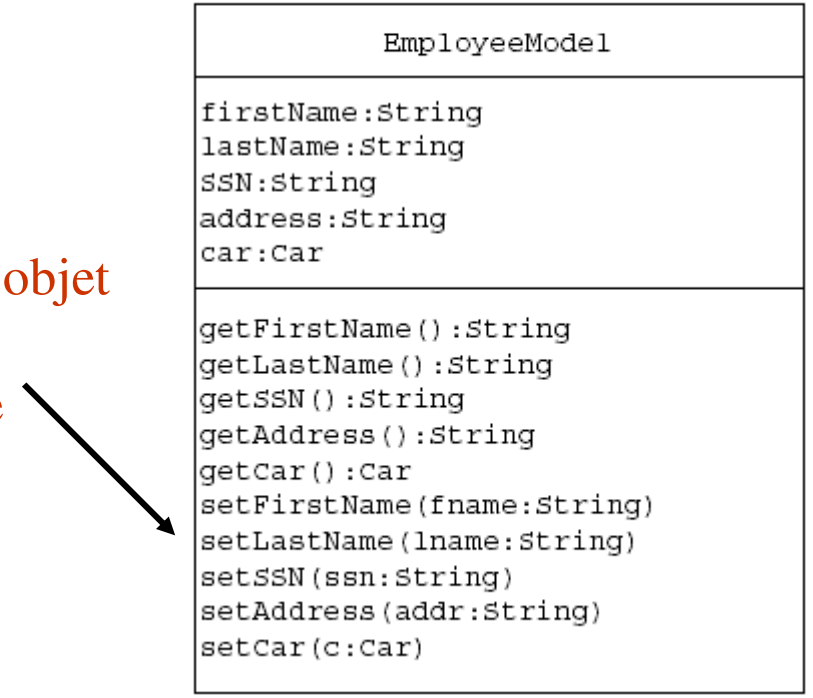
Accessible par la classe

Objet immuable

- Classe sans comportement spécifique
- Uniquement pour conserver des données
  - Exemple dossier d'un employé

setXXX

- Utile pour la création de l'objet
- Aucun contrôle sur les modification de l'instance

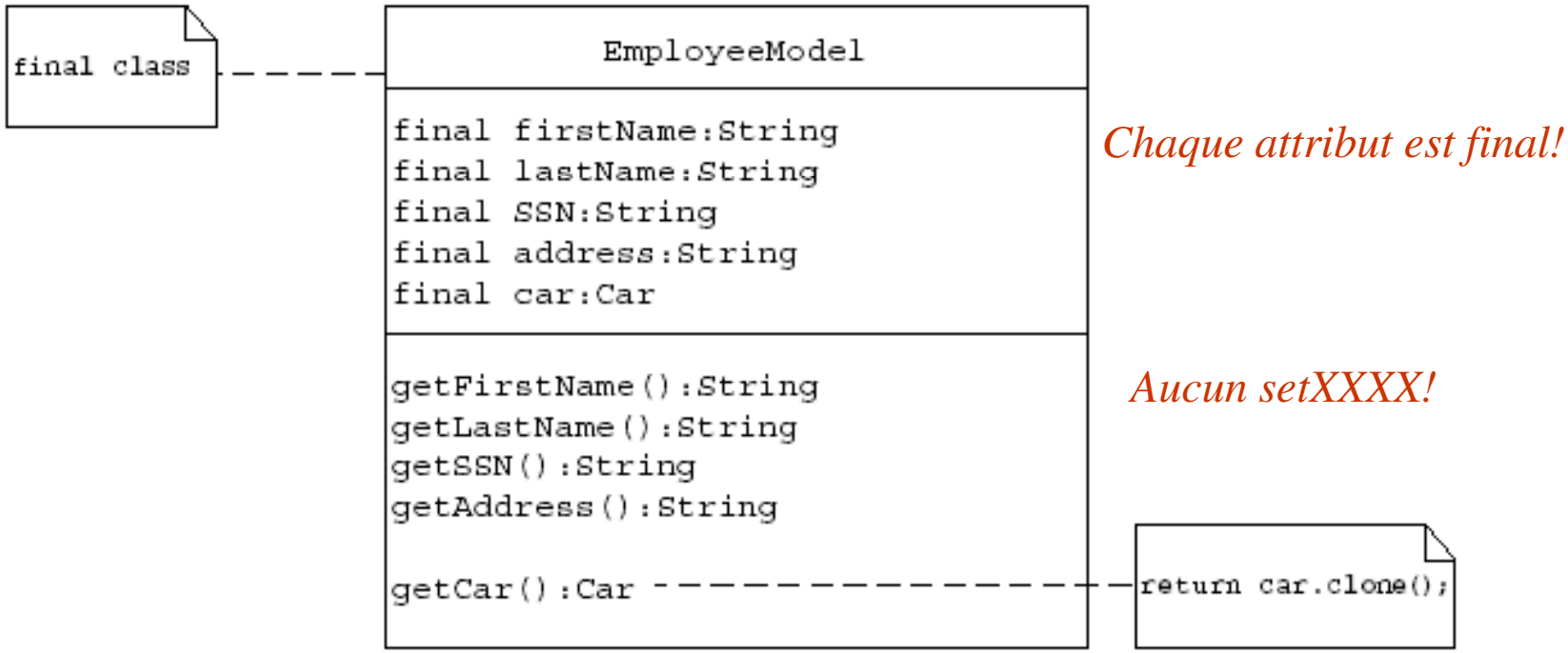


Objet immuable

- Éviter la manipulation
  - Contrôler l'accès concurrent par des fils (*threads*)
  - Limiter les problèmes de synchronisation
  - Pas de méthodes *setXXXX*
  - Données passées au constructeur de la classe
  - Déclarer la classe comme finale
    - Empêche de dériver une sous-classe avec un comportement
  - Si une variable contient une référence sur un objet
    - Faire une copie pour éviter les effets de bords

Objet immuable

Classe finale!



Chaque attribut est final!

Aucun setXXXX!

Copie des références!

Moniteur

- Comment accéder à une ressource partagée dans un environnement multithreaded
  - Assurer la consistance du comportement de la ressource
  - Difficile à gérer par les instances clients (solution globale)
- Exemple :
  - Accès à un fichier de journalisation (*log file*)
  - Éviter d'écraser le contenu
- Moniteur : permet de mettre un verrou
  - Un seul fil (thread) peut exécuter une méthode à un moment donné
- En Java
  - Synchronisation des méthodes

Moniteur

```
public class FileLogger {
   public synchronized void log(String msg) {
      DataOutputStream out = null;
      try {
         out = new DataOutputStream(
                        new FileOutputStream("log.txt",true));
         out.writeBytes(msg);
         out.close();
      } catch (FileNotFoundException ex) {
         //
      }  catch (IOException ex) {
         //
      }
   }
}
```

Patrons de génération

- Objectifs
  - Abstraire le processus d'instanciation
  - Rendre le système indépendant de la création des objets
- Thèmes récurrents
  - Encapsuler la connaissance de la classe concrète instanciée
  - Cacher le processus de création ou d'assemblage
- Tout ce que le système doit connaître des objets c'est leurs interfaces
  - *Interface*
  - *Abstract Parent Class*

Patrons de génération

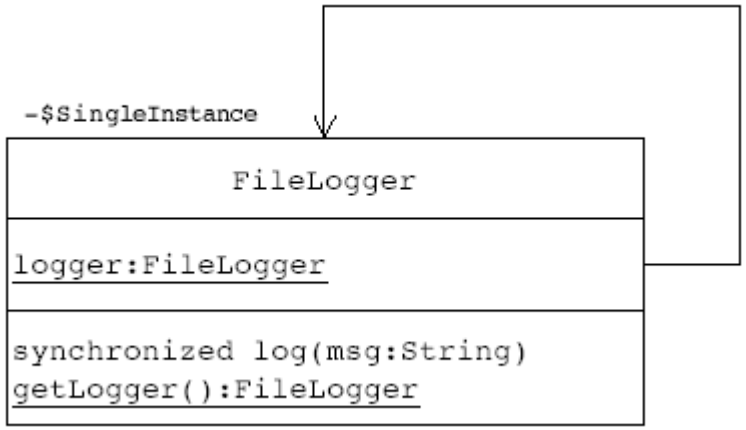
Patterns	**Description**
Méthode *factory*	Création d'une instance de plusieurs classes dérivées à partir d'une méthode
Singleton	Création de la seule instance qui puisse être générée d'une classe
Factory abstrait	Création d'une instance de plusieurs familles de classes
Prototype	Création par copie ou clônage d'une instance à partir d'un prototype
Builder	Création incrémentale d'une instance d'une classe complexe

Singleton

- Parfois, nous avons besoin d'une seule instance d'une classe
  - Par nécessité (ex: *System*)
  - une seule classe peut être suffisante
    - ex: *log file,* une seule connexion de base de données
- Singleton permet de:
  - S'assurer qu'il existe une et une seul instance d'une classe
  - L'accès consistante et globale par les autres classes
- Variable globale ?
  - Création d'autres instances
  - Plus coûteux à gérer par le client
  - Plus facile de le gérer au niveau de la classe

Singleton

- La classe maintient sa seule instance
- En java :
  - Rendre le constructeur privé
  - Gérer localement la création de l'instance
  - Avoir une méthode statique pour accéder à l'instance



Singleton

```
public class FileLogger implements Logger {

  private static FileLogger logger;

  //Prevent clients from using the constructor
  private FileLogger() {  }

  public static FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }

  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt",msg, true, true);
  }

}
```

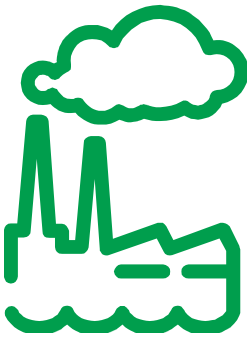
Une seule instance non accessible de l'extérieur

Constructeur non accessible de l'extérieur

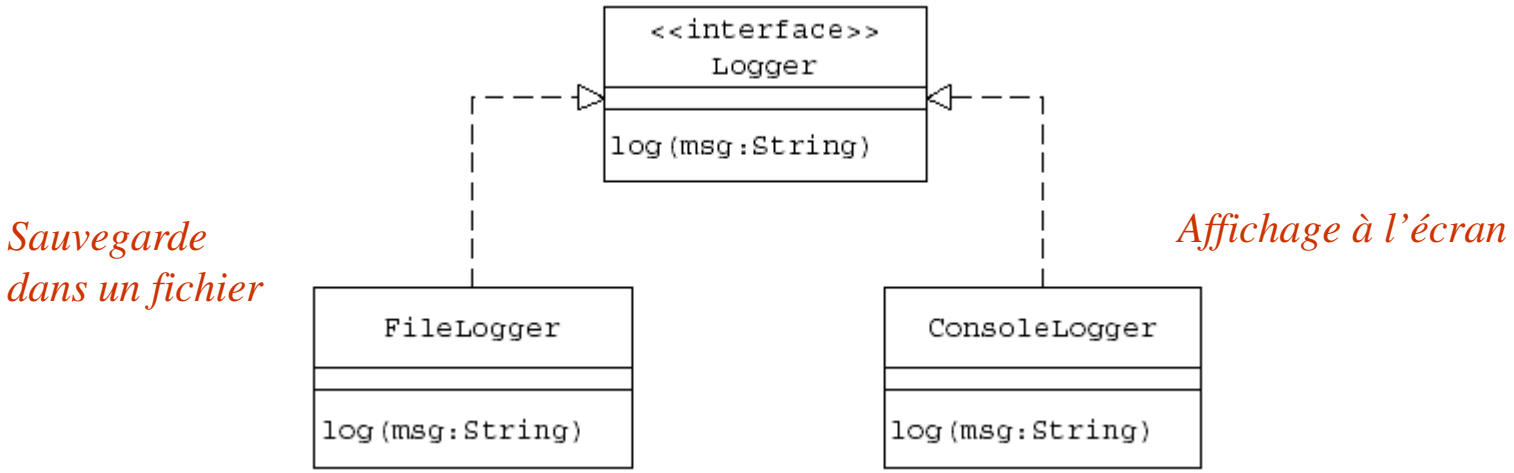
Un obtient l'instance par l'entremise de la classe

Moniteur

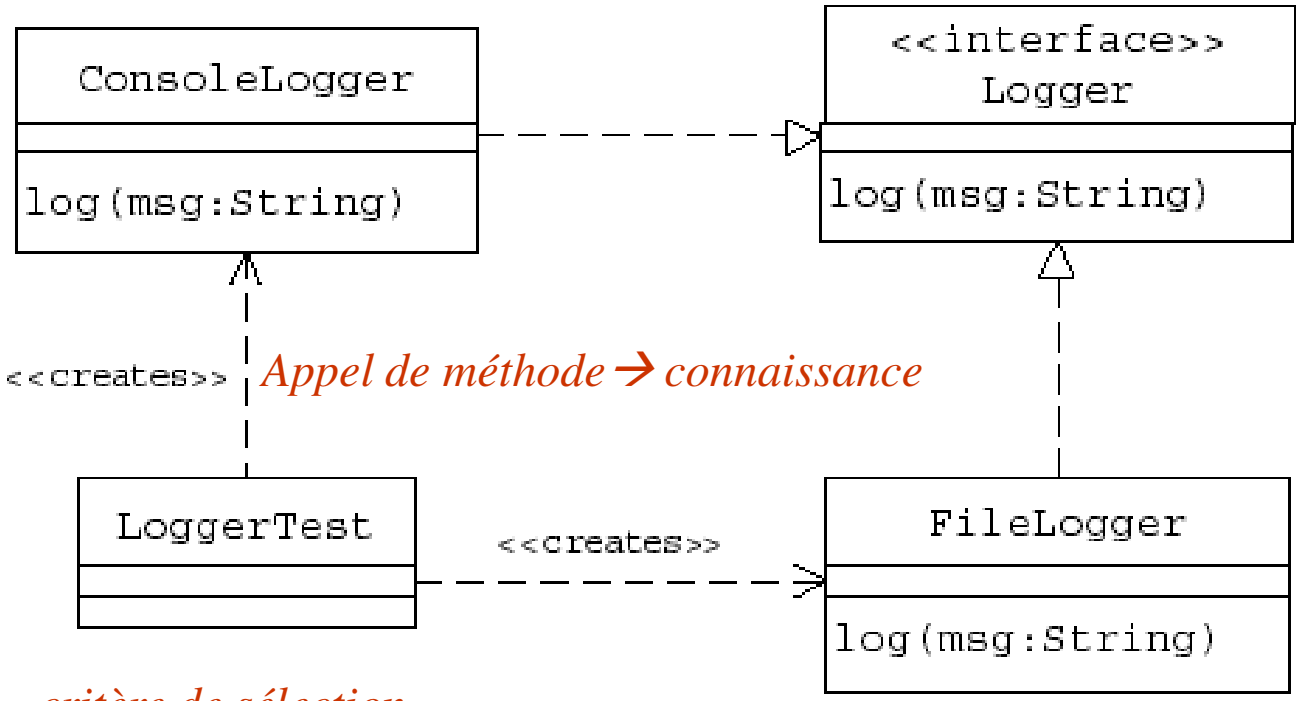
Méthode *factory*



- Exemple :
  - La journalisation (*log*) d'une application
  - Plusieurs objets veulent accéder au service
    - On crée une classe utilitaire pour éviter de répéter les mêmes détails au niveau de chaque client
  - Problème: on peut utiliser plusieurs médias (hiérarchie)



Méthode *factory*

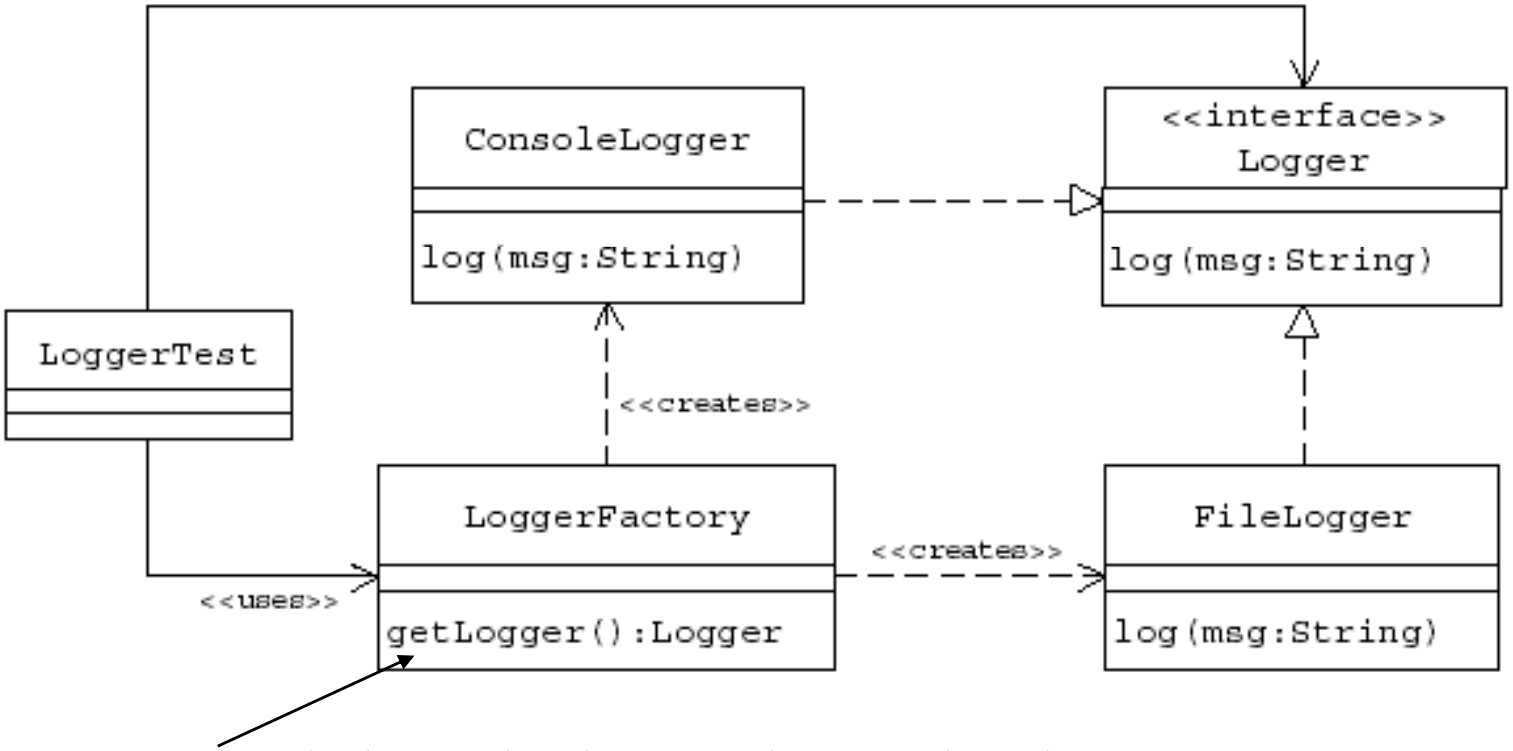


Choix = critère de sélection

Méthode *factory*

- Idée :
  - Encapsuler la sélection et création d'une classe de la hiérarchie dans une méthode
    - *Factory Method*
  - Comment déterminer le bon média
    - Un fichier de propriétés contient le média désiré *(FileLogging=OFF)*
  - Classe *Factory* doit
    - Lire le fichier de propriétés
    - Instancier la bonne classe (*FileLogger* ou *ConsoleLogger*)
    - Retourner une instance de la classe
  - Le client travaille avec l'abstraction du service
    - *Interface*
    - *Abstract Parent Class*

Méthode *factory*



Permet d'obtenir la classe et de gérer la sélection

Méthode *factory*

```
public class LoggerFactory {

    public boolean isFileLoggingEnabled() {
       Properties p = new Properties();
       try {
           p.load(ClassLoader.getSystemResourceAsStream("Logger.properties"));
           String fileLoggingValue =  p.getProperty("FileLogging");
           if (fileLoggingValue.equalsIgnoreCase("ON") == true)
             return true;
          else
             return false;
       } catch (IOException e) { return false; }
    }

    //Factory Method
    public Logger getLogger() {
          if (isFileLoggingEnabled()) {
             return new FileLogger();
          } else {
             return new ConsoleLogger();
          }
     } }
```

Abstract Factory

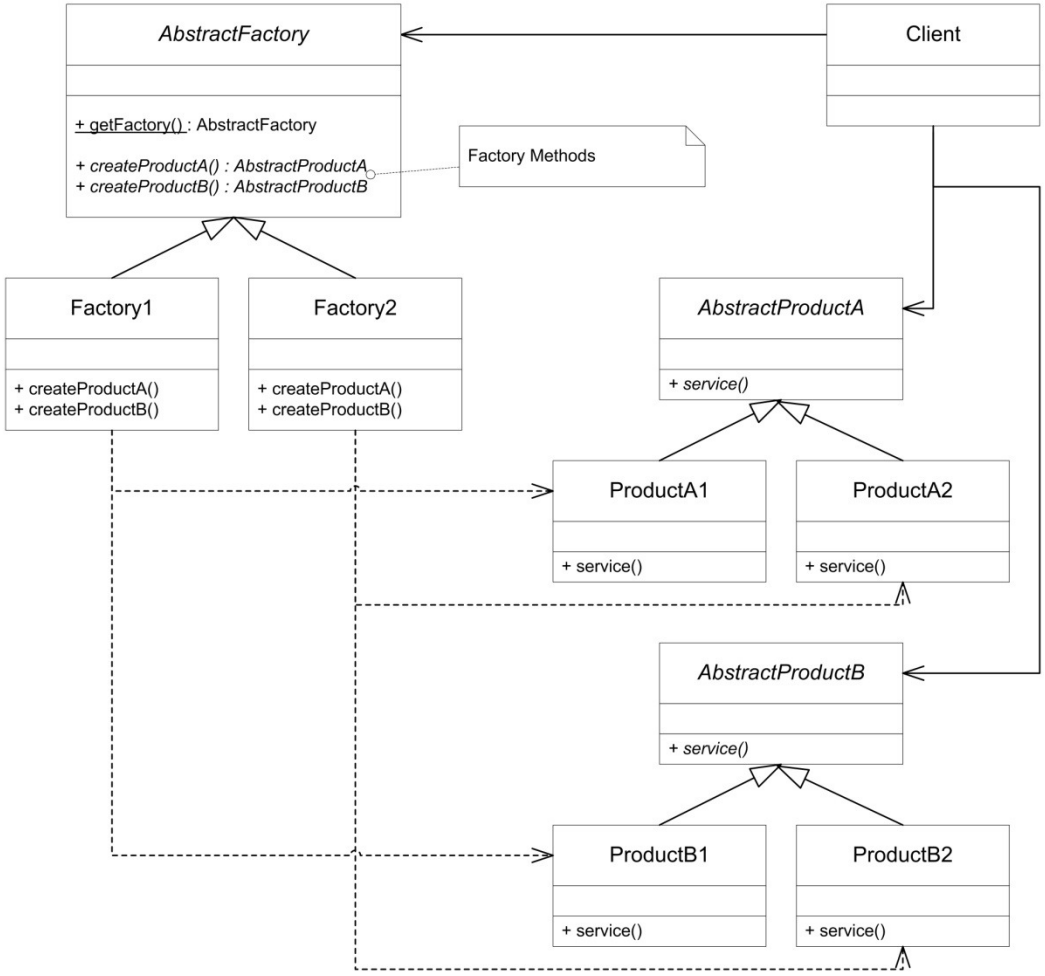
- Permet la création d'une instance de plusieurs familles de classes

- Problème
  - Plusieurs classes sont dépendantes ou sont en relations pour fournir un service
  - Le service peut être implanté de différentes façons par différentes familles de classes
  - Un objet requiert les services d'une famille de classes
  - L'objet ne connaît pas la famille de classes et les classes à instancier
    - La logique de sélection de la famille de classes

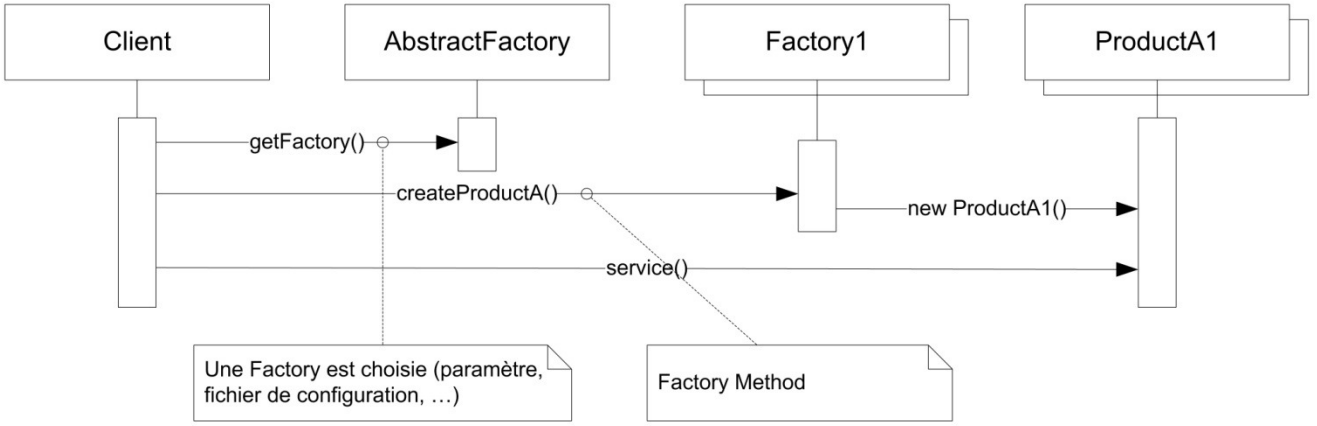
Abstract Factory

- Solution
  - Encapsuler les types génériques des participants au service
    - Abstractions des participants
    - Définir de méthodes de création des participant au service
      - *Factory Methods*
  - Encapsuler la fonction de sélection d'une famille de classes particulière et l'instanciation des classes qui composent cette famille
  - Le client fait appel à une *Factory* concrète pour obtenir les objets participants au service
  - Le client travaille avec une abstraction des participants
    - *Interface*
    - *Abstract Parent Class*

Abstract Factory



Abstract Factory



Abstract Factory

- Conséquences
  - Cache la logique de sélection de la famille de produits
  - Cache la logique de sélection du produit
  - Uniformise l'instanciation des produits
    - Instanciation dans les *Factory Methods*
  - Le client ne connaît ni les familles de classes ni la hiérarchie de classes des familles
    - Abstraction
  - Permet l'échange de familles de classes
  - Favorise la consistance des produits dans une famille

Abstract Factory - Exemple

- Système de gestion d'adresses et de numéros de téléphones
  - Format Français
    - Adresse : Rue ↵ Code postal Ville ↵ Pays
    - Téléphone : 33.0X.XX.XX.XX.XX
  - Format Américain
    - Adresse : Rue ↵ Ville, Région Code postal ↵ Pays
    - Téléphone : (1) XXX-XXX-XXXX

Abstract Factory - Exemple

```
/**
 * Telephone asbtrait  (Abstract Product)
 */
public abstract class AbstractTelephone {
  private String numero;

  public String getNumero() {
    return this.numero;
  }

  public abstract String getCodeRegional();

  public void setNumero(String valeur){
    try {
      Long.parseLong(valeur);
      this.numero = valeur;
    } catch (NumberFormatException ex) {
      ex.printStackTrace();
    }
  }

  public String toString() {
    return getNumero();
  }

}
```

Abstract Factory - Exemple

```
/**
 * Téléphone français (Concrete Product)
 */
public class FRTelephone extends AbstractTelephone {

  private static final String CODE_REGIONAL = "33";
  private static final int N_DIGITS = 9;

  public String getCodeRegional() {
    return CODE_REGIONAL;
  }

  public void setNumero(String valeur){
    if (valeur.length() == N_DIGITS){
      super.setNumero(valeur);
    }
  }

  public String toString() {
    String num = getNumero();
    return getCodeRegional() + "."
            + "0" + num.charAt(0) + "."
            + num.substring(1, 3) + "."
            + num.substring(3, 5) + "."
            + num.substring(5, 7) + "."
            + num.substring(7, 9);
  }

}
```

Abstract Factory - Exemple

```
/**
 * Abstract Factory
 */
public abstract class AbstractFactory {

  // getFactory()
  public static AbstractFactory createFactory(String type) {
    AbstractFactory factory;
    if (type.equalsIgnoreCase("fr")) {
      factory = new FRFactory();
    } else {
      factory = new USFactory();
    }
    return factory;
  }

  // Factory Methods
  public abstract AbstractAdresse createAdresse();
  public abstract AbstractTelephone createTelephone();

}
```

Abstract Factory - Exemple

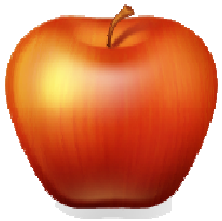
```
/**
 * Factory d'adresses et téléphones français (Concrete Factory)
 */
public class FRFactory extends AbstractFactory {

  // Factory Method
  public AbstractAdresse createAdresse() {
    return new FRAdresse();
  }

  // Factory Method
  public AbstractTelephone createTelephone() {
    return new FRTelephone();
  }

}
```

Abstract Factory - Exemple

```
public class Main {

  public static void main(String[] args) {
    String pays = "fr";

    AbstractFactory factory;
    factory = AbstractFactory.createFactory(pays);

    AbstractAdresse adresse = factory.createAdresse();
    adresse.setRue("De la Montagne");
    adresse.setVille("Paris");
    adresse.setRegion("");
    adresse.setCodePostal("75002");

    AbstractTelephone telephone = factory.createTelephone();
    telephone.setNumero("422345432");

    System.out.print(adresse.toString());
    System.out.print(telephone.toString());
  }

}
```

Prototype

- Problème :
  - Un système nécessite la création d'un ensemble d'objets semblables
    - Les objets ne différent que par quelques éléments de leur état
  - Les objets peuvent être coûteux à créer et initialiser

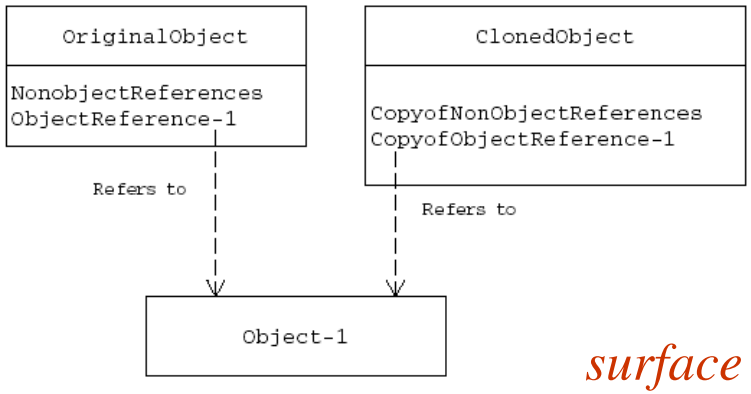


Prototype

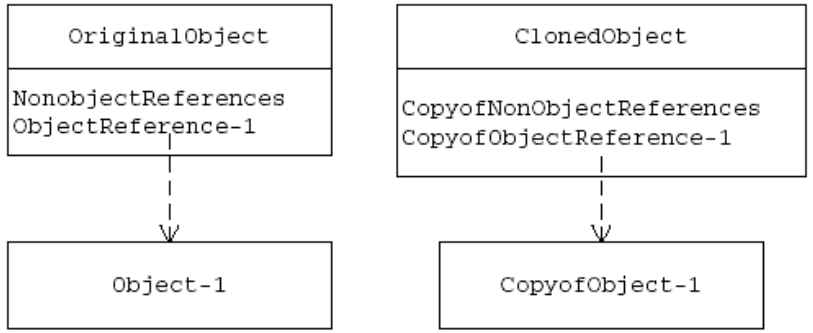
- Idée :
  - Créer un ou des objets de référence (prototype)
  - Créer les objets semblables par copie du prototype
  - Modifier l'état de la copie pour obtenir l'objet voulu
  - Copie ou clonage?
    - En java, repose sur la fonctionnalité *clone()*
    - Difficulté liée aux objets imbriqués
    - *Shallow copy* vs. *Deep copy*

Prototype

- Copie de surface (*shallow*)
  - L'objet original de haut niveau est dupliqué
  - Les objets imbriqués ne sont pas dupliqués
    - On ne copie que les références qui pointent vers le même objet de bas niveau
- Copie profonde (*deep*)
  - L'objet de haut niveau et les objets imbriqués sont tous dupliqués

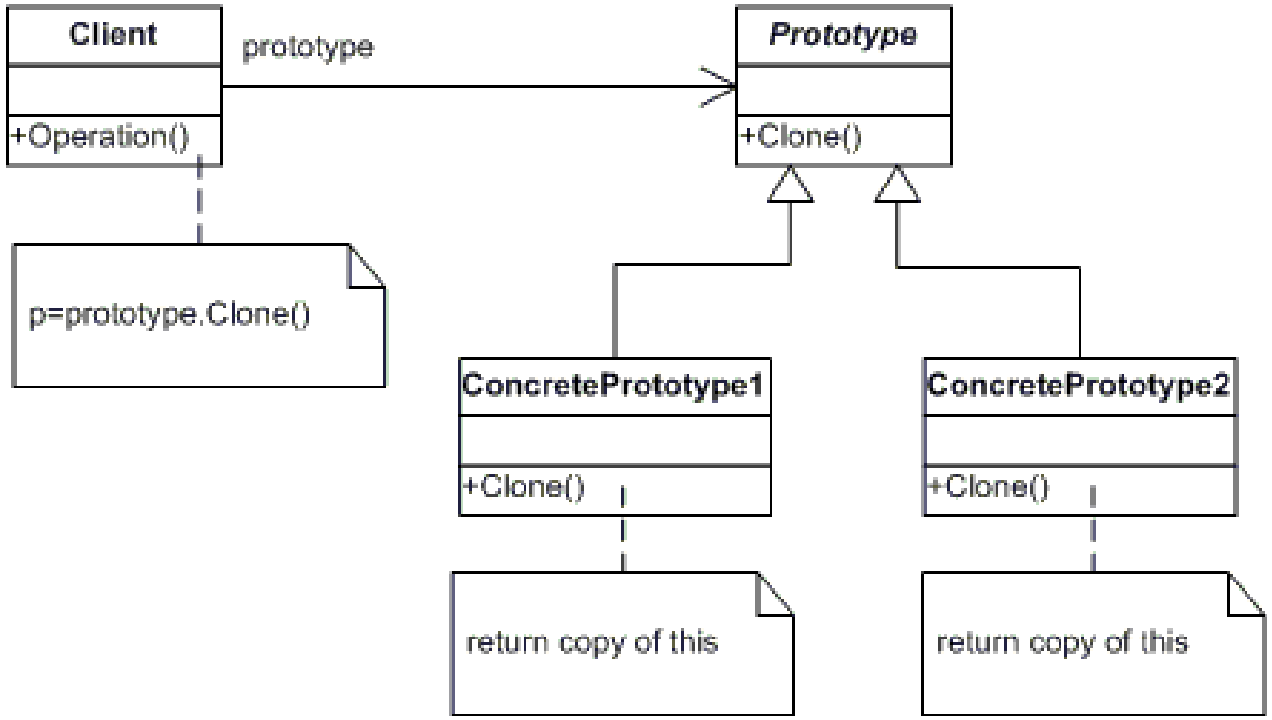


surface



profonde

Prototype


Client
prototype
+Operation()
p=prototype.Clone()
Prototype
+Clone()
ConcretePrototype1
+Clone()
return copy of this
ConcretePrototype2
+Clone()
return copy of this

Prototype

- Exemple de surface: Création d'un compte de banque

Les permissions sont les même pour un type de compte de banque

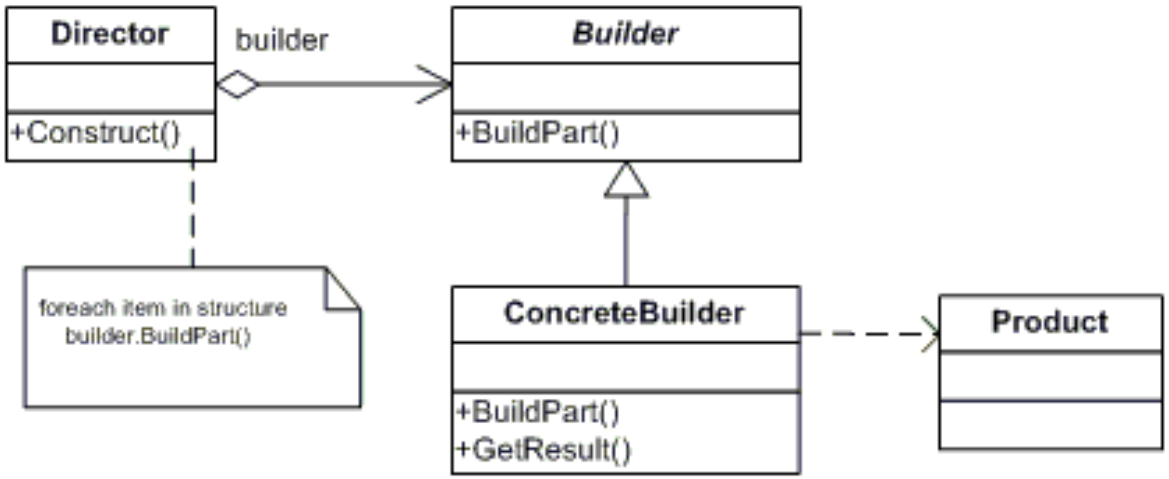
```
public class UserAccount implements Cloneable {
    private String userName;
    private String password;
    private String fname;
    private Vector permissions = new Vector();
    …
    …
    public Object clone() {       //Shallow Copy
        try {
                return super.clone();
         } catch (CloneNotSupportedException e) {
        return null;
    }
…
}
```

Builder

- Problème
  - La construction d'un objet nécessite un ensemble d'étapes complexes qui peuvent être implantées de différentes façons
  - Un processus de construction peut mener à différentes représentations

Builder

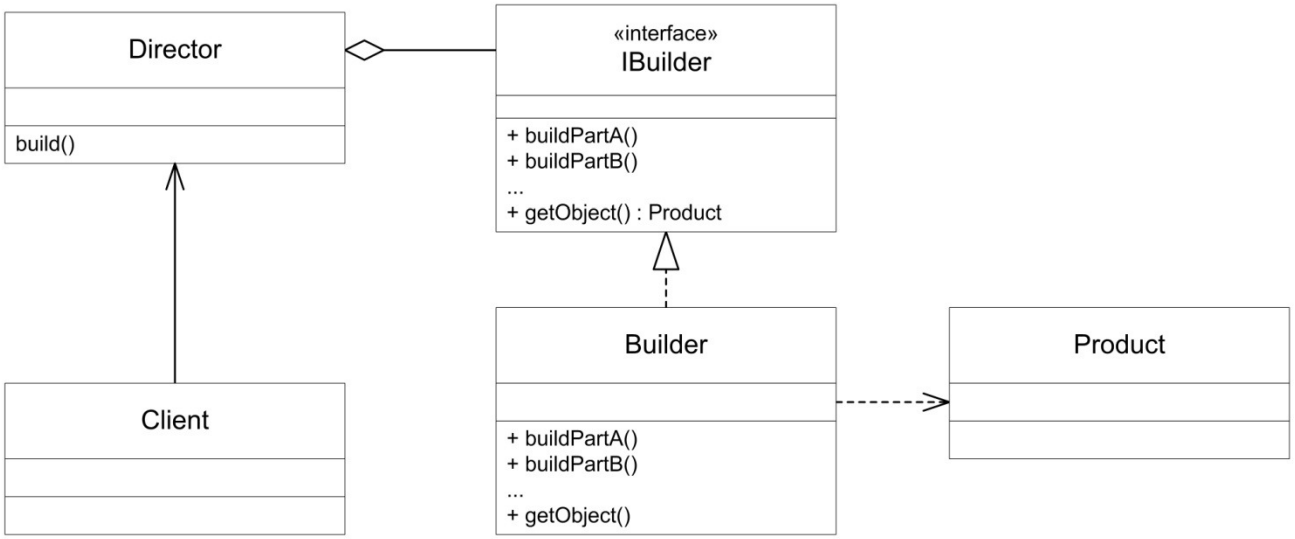
- Séparer la construction d'un objet complexe de sa représentation
- Le même processus de construction peut créer différentes représentations
- Permet la construction incrémentale d'une instance



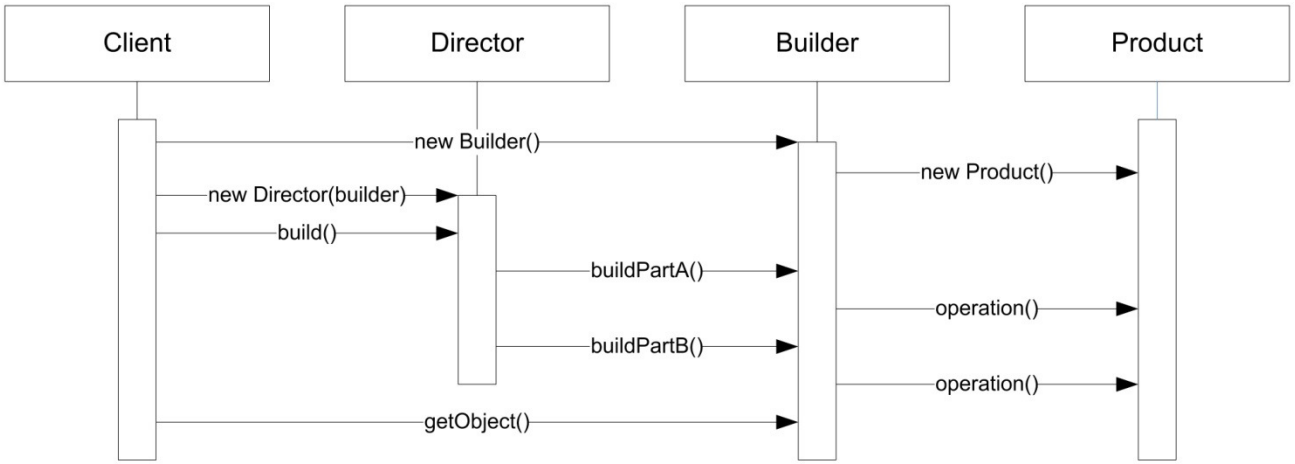
Builder

- Solution
  - Abstraire le processus général de construction
    - Les étapes de construction
  - Sortir la logique de construction dans une classe (*Builder*)
    - Un *Builder* implante les étapes qui mènent à une représentation
  - Mettre en place un coordonnateur des étapes de la construction
    - Le *Director* utilise le *Builder* pour construire un objet

Builder


Director
build()
«interface»
IBuilder
+ buildPartA()
+ buildPartB()
...
+ getObject() : Product
Builder
+ buildPartA()
+ buildPartB()
...
+ getObject()
Product
Client

Builder


Client
Director
Builder
Product
new Builder()
new Director(builder)
build()
buildPartA()
buildPartB()
getObject()
new Product()
operation()
operation()

Builder

- Conséquences
  - Améliore la modularité
  - Rend le processus de construction indépendant des composantes qui forment l'objet produit
  - Facilite l'ajout ou la modification d'une implantation d'un processus de construction menant à une représentation particulière d'un objet

Builder - Exemple

- Système de commande pour un restaurant de type restauration rapide (fast-food)
  - Construction de trios
    - Plat principal
    - Accompagnement
    - Breuvage

Builder - Exemple

```
/**
 * Trio (Product)
 */
public class Trio {
  private String platPrincipal;
  private String accompagnement;
  private String boisson;

  public String getPlatPrincipal() { return this.platPrincipal; }

  public String getAccompagnement() { return this.accompagnement; }

  public String getBoisson() { return this.boisson; }

  public void setPlatPrincipal(String plat) {
    this.platPrincipal = plat;
  }

  public void setAccompagnement(String acc) {
    this.accompagnement = acc;
  }

  public void setBoisson(String brevage) {
    this.boisson = brevage;
  }
}
```

Builder - Exemple

```
/**
 * Constructeur de trio (IBuilder)
 */
public interface ITrioBuilder {
  // buildPart
  public void ajouterPlatPrincipal();
  public void ajouterAccompagnement();
  public void ajouterBoisson();

  // getObject
  public Trio getTrio();
}
```

Builder - Exemple

```
/**
 * Constructeur de trio concret (Builder)
 */
public class TrioSaladeBuilder implements ITrioBuilder {
  private Trio trio;

  public TrioSaladeBuilder() {
    this.trio = new Trio();
  }

  public void ajouterPlatPrincipal() {
    this.trio.setPlatPrincipal("Salade");
  }

  public void ajouterAccompagnement() {
    this.trio.setAccompagnement("Yogourt");
  }

  public void ajouterBoisson() {
    this.trio.setBoisson("Jus de fruits");
  }

  public Trio getTrio() {
    return this.trio;
  }

}
```

Builder - Exemple

```
/**
 * Caissier (Director)
 */
public class Caissier {
  private ITrioBuilder builder;

  public Caissier(ITrioBuilder b) {
    this.builder = b;
  }

  // build
  public void preparerCommande() {
    this.builder.ajouterPlatPrincipal();
    this.builder.ajouterAccompagnement();
    this.builder.ajouterBoisson();
  }

  public Trio getTrio() {
    return this.builder.getTrio();
  }

}
```

Builder - Exemple

```
public class Main {

  public static void main(String[] args) {
    ITrioBuilder trioSalade = new TrioSaladeBuilder();
    ITrioBuilder trioCheese = new TrioCheeseBurgerBuilder();

    System.out.println("------------------------------");
    Caissier caissier = new Caissier(trioSalade);
    caissier.preparerCommande();
    Trio trio = caissier.getTrio();
    System.out.println(trio.getPlatPrincipal());
    System.out.println(trio.getAccompagnement());
    System.out.println(trio.getBoisson());

    System.out.println("------------------------------");
    caissier = new Caissier(trioCheese);
    caissier.preparerCommande();
    trio = caissier.getTrio();
    System.out.println(trio.getPlatPrincipal());
    System.out.println(trio.getAccompagnement());
    System.out.println(trio.getBoisson());
  }

}
```