DESIGN PATTERN

Les patterns GoF

La conception Orientée objet

- Un art difficile…
- Une conception réutilisable, extensible, adaptable, performante ➔ extrêmement difficile.
- Quelle est la différence entre un novice et un concepteur expérimenté
  - le novice hésite beaucoup entre différentes variantes
  - l'expert trouve tout de suite la bonne solution.
- Quel est le secret ?

  ➔ l'EXPERIENCE !

La conception Orientée objet

3

- L'expérience c'est
  - ne pas réinventer la roue,
  - réutiliser systématiquement des solutions qui ont fait leurs preuves,
- pour une conception modulaire, élégante, adaptable

  ➔ répétition de certains profils de classes ou collaboration d'objets
  - les design patterns
  - = modèles de conception
  - = patrons de conception

Définition

4

- Un *pattern* est une bonne pratique face à un problème courant.
- Un ***pattern*** est une capitalisation du savoir-faire et de l'expérience pour résoudre des problèmes récurrents intervenants dans les différents niveaux du processus:
  - analyse (*analysis pattern*),
  - architecture (*architectural pattern*)
  - conception (*design pattern*)
  - programmation (*idiomes* ou idiomatiques en français)
- C'est un moyen de partager la connaissance de la résolution d'un type de problème sous une forme « conceptuelle », mais ce n'est pas une solution implémentée.
- Aide à la construction de logiciels répondant à des propriétés précises, de logiciels complexes et hétérogènes
- Traductions : patrons de conception, schémas de conception, motifs de conception

Définition

5

- Un design pattern décrit une structure commune et répétitive de composants en interaction (la solution) qui résout un problème récurrent de conception dans un contexte particulier.
- Qu'est-ce qu'un bon patron de conception ?
  - il résout un problème
  - c'est un concept éprouvé
  - la solution n'est pas évidente
  - il décrit une collaboration entre objets

Objectifs

6

- Modularité
  - Facilité de gestion (technologie objet)
- Cohésion
  - Degré avec lequel les tâches réalisées par un seul module sont fonctionnellement reliées
  - Une forte cohésion est une bonne qualité
- Couplage
  - Degré d'interaction entre les modules dans le système
  - Un couplage faible est une bonne qualité
- Réutilisabilité
  - Bibliothèques, frameworks (cadres)

Exemple « mauvaise » cohésion

7

```
public class GameBoard {

public GamePiece[ ][ ] getState() { … }
 // Méthode copiant la grille dans un tableau temporaire, résultat de l'appel de la méthode.

public Player isWinner() { … }
 // vérifie l'état du jeu pour savoir s'il existe un gagnant, dont la référence est retournée.
 // Null est retourné si aucun gagnant.

public boolean isTie() { … }
 //retourne true si aucun déplacement ne peut être effectué, false sinon.

public void display () { … }
// affichage du contenu du jeu. Espaces blancs affichés pour chacune des
// références nulles.
}
```

GameBoard est responsable des règles du jeu et de l'affichage

Exemple: « bonne » cohésion

8

```
public class GameBoard {

public GamePiece[ ][ ] getState() { … }

public Player isWinner() { … }

public boolean isTie() { … }

}

public class BoardDisplay {

public void displayBoard (GameBoard gb) { … }
// affichage du contenu du jeu. Espaces blancs affichés pour chacune des
// références nulles.
}
```

Exemple : couplage

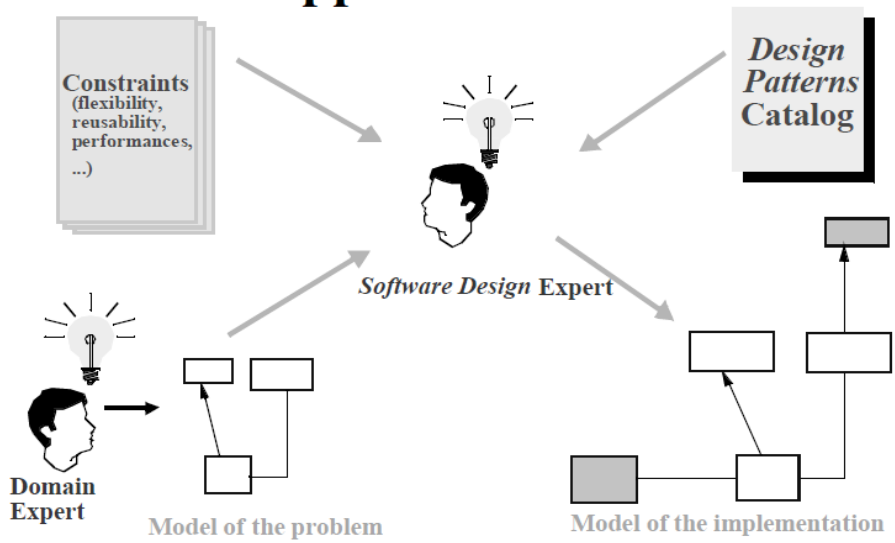

```
void initArray(int[] iGradeArray, int nStudents) {
   int i;
   for (i = 0; i < nStudents; i++) {
      iGradeArray[i] = 0;
   }
}
```

Couplage entre client et initArray par le Paramètre "nStudents"

```
void initArray(int[ ] iGradeArray) {
   int i;
   for (i=0; i < iGradeArray.length; i++)  {
      iGradeArray[i] = 0;
   }
}
```

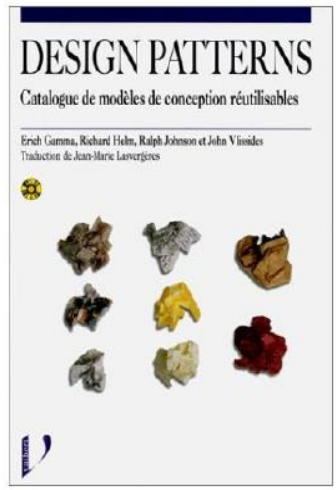
Couplage faible (et meilleure fiabilité) au travers de l'utilisation de l'attribut "length"

La conception dans le processus de développement avec UML



Ouvrage de référence

11



- GoF « Gang of Four »

Design patterns. Elements of reusable Object-Oriented Software [1994]

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Les design Patterns GoF

12

- Très célèbres
- conçus par 4 informaticiens : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, surnommés le **"Gang of Four" (GoF)**
- Ils proposent des solutions élégantes, pour résoudre différents problèmes récurrents rencontrés par les architectes logiciels.
- Le catalogue GoF patterns (23 patterns)

Catégories de Design Patterns

13

- **Création**
- Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
  - Singleton
  - Fabrication (Factory Method)
  - Fabrique abstraite (Abstract Factory)
  - Monteur (builder)
  - Prototype (Prototype)

Catégories de Design Patterns

14

- **Structure**
  - Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
  - Découplage de l'interface et de l'implémentation de classes et d'objets
    - Adaptateur (Adapter) ,
    - Pont (Bridge) ,
    - Objet composite (Composite),
    - Décorateur (Decorator) ,
    - Façade (Facade) ,
    - Poids-mouche ou poids-plume (Flyweight) ,
    - Proxy (Proxy)

Catégories de Design Patterns

15

- **Comportement**
  - Description de comportements d'interaction entre objets
  - Gestion des interactions dynamiques entre des classes et des objets
    - Chaîne de responsabilité (Chain of responsibility),
    - Commande (Command) ,
    - Interpréteur (Interpreter)
    - Itérateur (Iterator),
    - Médiateur (Mediator)
    - Memento (Memento)
    - Observateur (Observer)
    - État (State)
    - Stratégie (Strategy)
    - Patron de méthode (Template Method)
    - Visiteur (Visitor)

16

Design patterns de création

- Abstraction du processus de création
- Encapsulation de la logique de création
- On ne sait pas à l'avance ce qui sera créé ou comment cela sera créé

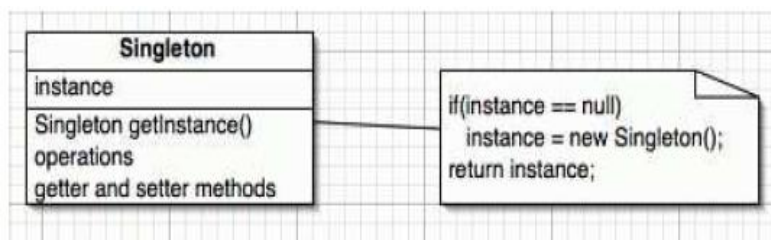
Singleton : présentation

17

- Intention
  - S'assurer qu'une classe a une seule instance, et fournir un point d'accès global à celle-ci.
- Exemple
  - Un seul window manager, un seul point d'accès à une base de données, un seul pilote de périphérique (instancier plus qu'une fois ➔surcharge inutile, comportement incohérent)
  - Comment créer une classe utilisée plusieurs fois dans une application mais instanciée une seule fois?
- Solution
  - Le Singleton va permettre de récupérer l'objet unique, sans se soucier de vérifier si il existe déjà ou pas
  - rendre le constructeur de la classe privée.
    - méthode statique qui retournera un objet correspondant au type de la classe

Singleton: Diagramme

18



- Construire un pseudo constructeur
  - Déclarer une méthode statique qui retourne un objet de type de la classe
  - On peut contrôler la valeur retournée

Singleton : implémentation

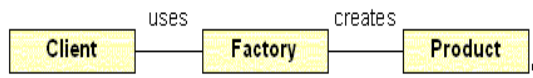
19

```
public class Singleton {
/** Création de l'instance au niveau de la variable statique. */
private static  Singleton INSTANCE = null;
/**La présence d'un constructeur privé supprime
* le constructeur public par défaut.  */
private Singleton() {}
/**Retourne l'instance du singleton. */
public static Singleton getInstance() {
If(INSTANCE ==null) {
INSTANCE =new Singleton()
}
return INSTANCE;
}
/** A partir d'ici, toutes les méthodes de l'instance (non statique)
*/
}
```

Pattern Factory: présentation

20

- Les langages oo fournissent des mécanismes d'instanciation et d'initialisation(new, newobj,..) : utiliser sans prévoir les conséquences → inflexibilités des systèmes due au couplage fort entre créateur et classe à créer
- Solution : modèles créateurs décrivant les mécanismes de création des objets qui favorisent la réutilisation
- il y a un certain nombre de variations du modèle de fabrique de classe
- la plupart des variantes utilisent le même jeu d'acteurs primaires, un **Client**, une **Fabrique** et un **Produit** (Client, Factory, Product).
- Le client utilise la fabrique pour créer une instance du produit.



- La fabrique rend abstraite, pour le client, la création et l'initialisation du produit.
- si l'implémentation du produit change dans le temps, le client quant à lui reste inchangé.

Factory Method : présentation

21

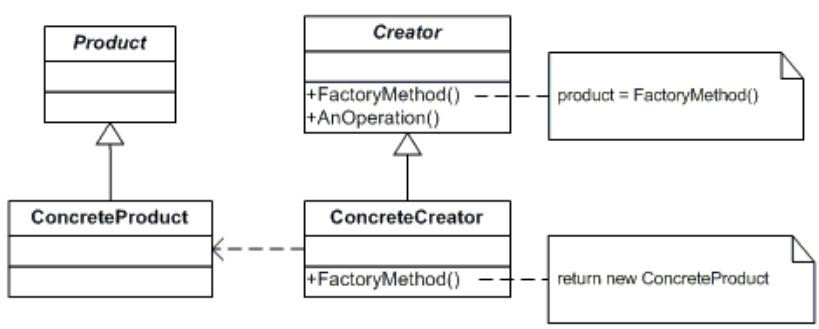
- Intention
- Définit une classe pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier
  - Permet la délégation d'instanciation
- Exemple
  - Instancier des classes, en connaissant uniquement les classes abstraites.
  - Framework pour des applications qui présentent des documents multiples à l'utilisateur

Factory Method : diagramme

22

Champs d'application

- une classe ne peut pas anticiper l'objet qu'elle doit construire (utiliser)
- une classe délègue la responsabilité de la création à ses sousclasses, tout en concentrant l'interface dans une classe unique



Factory Method: structure

23 Structure :

- Objet à créer :
  - Une interface (ou classe abstraite) Product
    - définit l'interface des objets créés par la fabrication
  - Une ou plusieurs classes concrètes : ConcreteProduct
    - implémente l'interface Product
- Createur
  - Une classe abstraite, Creator :
    - déclare la fabrication; celle-ci renvoie un objet de type Product. Le Creator peut également définir une implémentation par défaut de la fabrication, qui renvoie un objet ConcreteProduct par défaut. Il peut appeler la fabrication pour créer un objet Product.
    - public abstract Product FactoryMethod();
  - Une ou plusieurs classes concrètes ConcreteCreator, qui retournent des ConcreteProduct

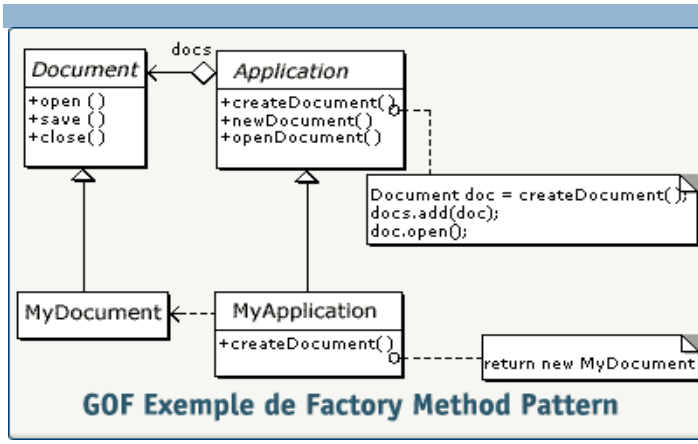
Factory Method: conséquences

24

- Conséquence :
  - La fabrication dispense d'avoir à incorporer à son code des classes spécifiques de l'application. Le code ne concerne que l'interface Product, il peut donc fonctionner avec toute classe ConcreteProduct définie par l'utilisateur
  - La création d'objets à l'intérieur d'une classe avec la méthode fabrication est toujours plus flexible que la création d'un objet directement

Exemple

25



Factory Method :implementation

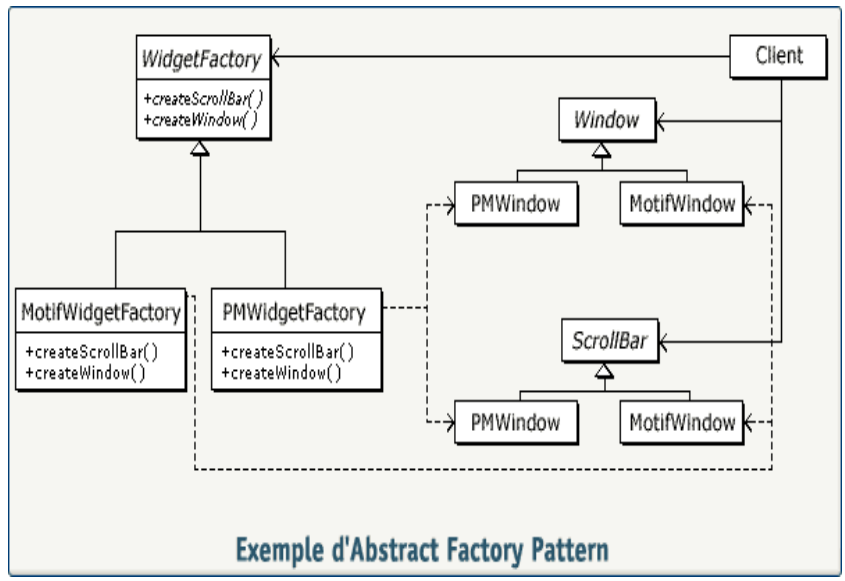
26

- Implémentation (2 possibilités)
  - La classe Creator est une classe abstraite et ne fournit pas d'implémentation
  - La classe Creator est une classe concrète qui fournit une implémentation par défaut pour la fabrication
- Fabrication paramétrée
  - Utilisation de paramètres pour déterminer la variété d'objet
  - Exemple : fabrique de documents, choix d'objet en fonction de dimensions, etc.

Abstract Factory : présentation

27

- *Intention*
  - Le pattern abstract factory fournit une interface pour la création de familles d'objets apparentés ou indépendants sans devoir spécifier leurs classes concrètes.
- *Exemple:*
  - *une boite à outils qui supporte de multiples apparences : motif et présentation Manager : pour qu'une application soit portable indépendamment de l'apparence utilisée les widgets ne doivent pas être codés en dur*



28

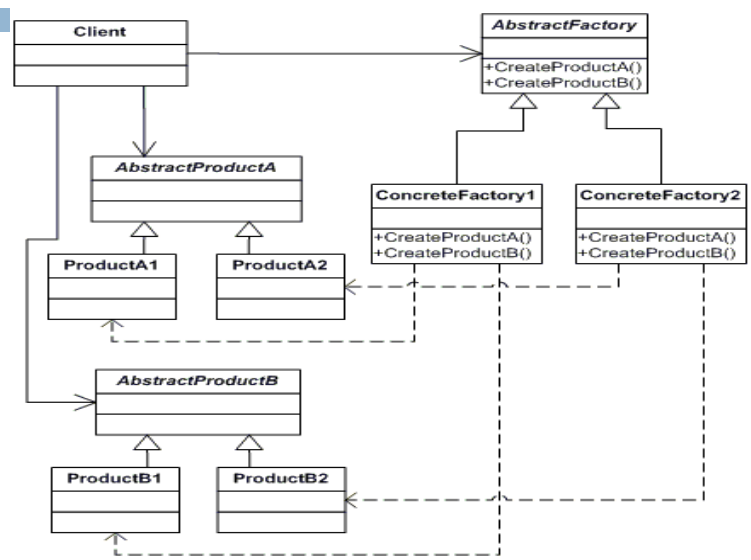
Abstract Factory: utilisation

29

- Champs d'application
  - Indépendance de comment les objets/produits sont créés, composés et représentés
  - Configuration d'un système par une instance d'une multitude de familles de produits
  - Conception d'une famille d'objets pour être utilisés ensemble et contrôle de cette contrainte
  - Bibliothèque fournie avec seulement leurs interfaces, pas leurs implémentations
    (bibliothèque graphique, look-and-feel)

Abstract Factory : diagramme

30



Abstract Factory : structure

31

- Structure
  - La fabrique
    - AbstractFactory déclare l'interface pour les opérations qui créent des objets abstraits
    - ConcreteFactory implémente les opérations qui créent les objets concrets
  - Les objets (plusieurs types)
    - AbstractProduct déclare une interface pour un type d'objet
    - ConcreteProduct définit un objet qui doit être créé par la fabrique concrète correspondante et implémente l'interface AbstractProduct
  - L'utilisateur
    - Client utilise seulement les interfaces déclarée par AbstractFactory et par les classes AbstractProduct

Abstract Factory: collaboration

32

- Collaborations
  - Normalement, une seule instance de fabrique concrète est créée à l'exécution. Cette fabrique crée les objets avec une implémentation spécifique. Pour créer différents sortes d'objets, les clients doivent utiliser différentes fabriques concrètes.
  - La fabrique abstraite défère la création des objets à ses sous-classes concrètes

Abstract Factory :conséquences

33

- Conséquences
  - Isolation des classes concrètes (seules les classes abstraites sont connues)
  - Échange facile des familles de produit
  - Encouragement de la cohérence entre les produits
  - Prise en compte difficile de nouvelles formes de produit car Abstract Factory détermine les produits qui peuvent être crées

Abstract Factory : implémentation

34

- Implémentation
  - Les fabriques sont souvent des singletons
  - Ce sont les sous-classes concrètes qui font la création, en utilisant le plus souvent une Factory Method

3 Création d'objets et Factory Method

On suppose que l'on souhaite faire une machine automatique cuisinant des tartes aux fruits[1]. Pour cela, on développe dans un premier temps une application Java permettant de simuler le comportement de la machine. La classe principale de l'application est `MachineTarte` et dispose d'une méthode `commanderTarte` qui aura l'allure suivante :

```
public Tarte commanderTarte() {
  Tarte tarte = new Tarte();

  tarte.preparer();
  tarte.cuire();
  tarte.emballer();

  return tarte;
}
```

Les méthodes `preparer`, `cuire` et `emballer` sont des méthodes simulant le travail effectif de la machine.

1. on suppose que l'on a une machine évoluée qui peut faire plusieurs types de tartes aux fruits. On va donc rendre `Tarte` abstraite et créer une hiérarchie de classes représentée sur la figure 9.

   On va donc modifier la méthode `commanderTarte` pour utiliser les nouveaux types de tartes : on va passer en paramètre de `commanderTarte` une chaîne de caractères précisant le type de tarte choisi

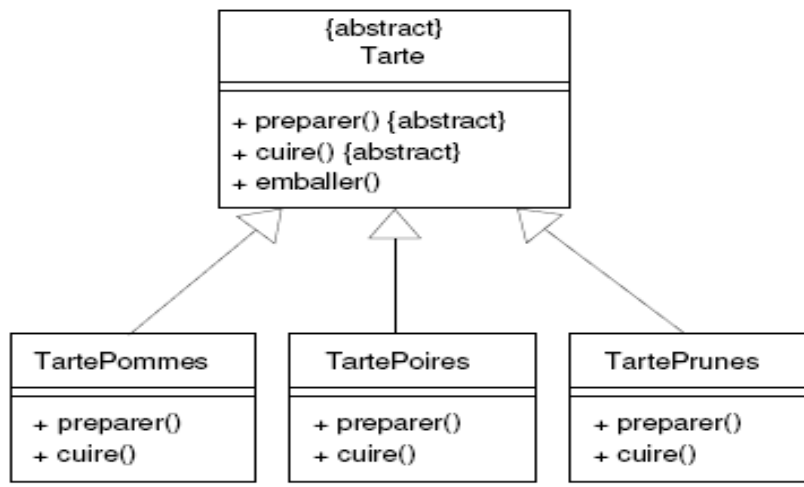


FIG. 9 – Hiérarchie de classes représentant les tartes

La méthode commanderTarte est présentée ci-après. Il fallait juste ajouter un paramètre de typ String à la méthode et utiliser la méthode equals pour comparer cette chaîne de caractères au chaînes connues. La gestion des éventuelles erreurs n'est pas optimale : si le type de tarte n'est pa reconnu, on aura une exception de type NullPointerException qui sera levée.

```
public Tarte commanderTarte(String type) {
    Tarte tarte = null;

    if (type.equals("pommes")) {
        tarte = new TartePommes();
    } else if (type.equals("poires")) {
        tarte = new TartePoires();
    } else if (type.equals("prunes")) {
        tarte = new TartePrunes();
    }

    tarte.preparer();
    tarte.cuire();
    tarte.emballer();

    return tarte;
```

si l'on souhaite ajouter de nouvelles tartes, il va falloir modifier la méthode commanderTarte. Or on souhaite fermer commanderTarte à la modification[2].

Pour résoudre ce problème, on va déléguer la création des tartes à une classe FabriqueTarte via une méthode creerTarte.

(a) représenter sur un diagramme de classes les classes MachineTarte, FabriqueTarte et la hiérarchie des tartes.

Le diagramme de classes est représenté sur la figure 10. Rien de bien particulier ici, les associations entre les classes étant assez simples.

(b) représenter sur un diagramme de séquence les interactions entre les classes lors de la commande d'une tarte aux pommes.

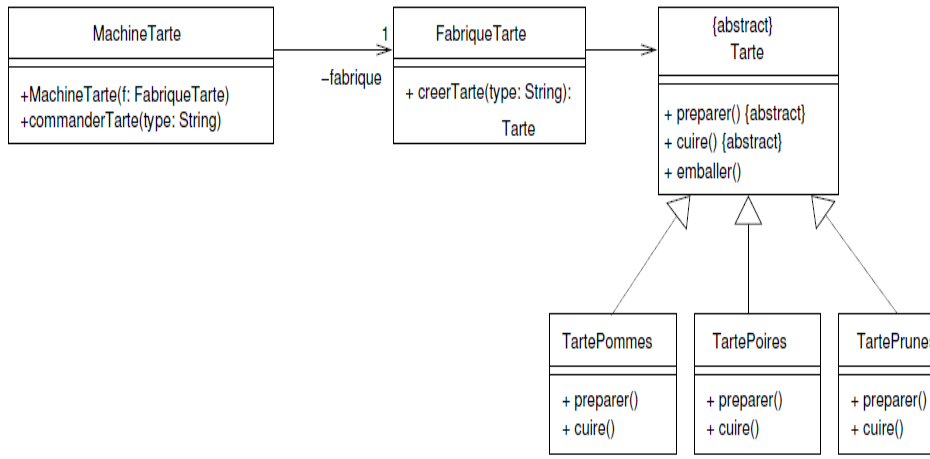


FIG. 10 – Diagramme de classes représentant `FabriqueTarte`

39

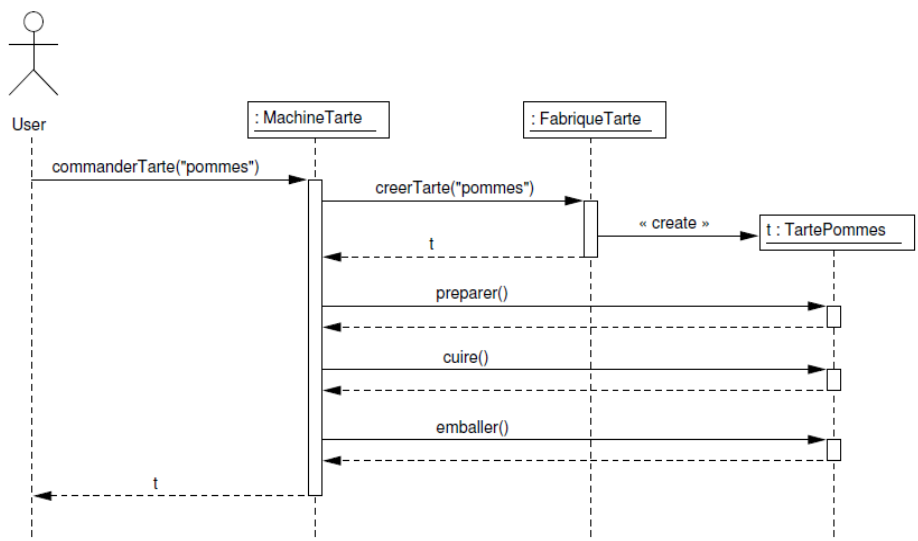


FIG. 11 – Diagramme de séquence représentant l'interaction entre `MachineTarte`, `FabriqueTarte` `TartePommes`

```java
public class FabriqueTarte {

    /**
     * <code>creerTarte</code> permet de creer une tarte. Attention, les seuls
     * types connus actuellement sont "pommes", "poires" et "prunes".
     *
     * @param type une instance de <code>String</code> representant le type
     *             de tarte a faire
     * @return la <code>Tarte</code> demandee. Peut etre <code>null</code> si
     *         le type de tarte n'existe pas
     */
    public Tarte creerTarte(String type) {
        Tarte tarte = null;

        if (type.equals("pommes")) {
            tarte = new TartePommes();
        } else if (type.equals("poires")) {
            tarte = new TartePoires();
        } else if (type.equals("prunes")) {
            tarte = new TartePrunes();
        }

        return tarte;
    }
}
```

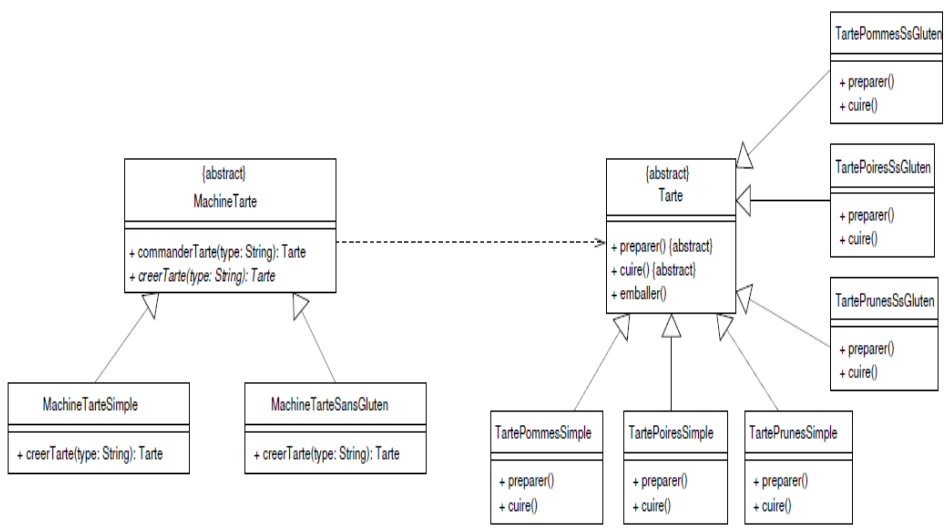
```java
public class MachineTarteSecond {

    private FabriqueTarte fabrique;

    /**
     * Creer une nouvelle instance de <code>MachineTarteSecond</code>.
     *
     * @param fabrique l'instance de <code>FabriqueTarte</code> que l'on veut
     *                 utiliser pour construire des tartes
     */
    public MachineTarteSecond(FabriqueTarte fabrique) {
        this.fabrique = fabrique;
    }

    /**
     * <code>commanderTarte</code> permet a un utilisateur de commander
     * une tarte d'un certain type.
     *
     * @param type une instance de <code>String</code> representant le type
     *             de tarte. Pour l'instant, seuls "pommes", "poires" et
     *             "prunes" sont connus.
     * @return la <code>Tarte</code> prete a etre degustee
     */
    public Tarte commanderTarte(String type) {
        Tarte tarte = this.fabrique.creerTarte(type);

        tarte.preparer();
        tarte.cuire();
        tarte.emballer();
```

on suppose maintenant que l'on veut pouvoir créer deux grands types de tartes : des tartes normales et des tartes sans gluten (pour les personnes allergiques). La hiérarchie de classes représentant les tartes va donc s'en trouver modifiée. L'utilisateur choisira au départ la « famille » de tarte qu'il veut (sans gluten ou normale), puis le type de la tarte (aux pommes, aux poires etc.). Dans notre simulation, nous voulons donc avoir deux types de `MachineTarte` : un pour les tartes classiques et l'autre pour les tartes sans gluten.

Nous voulons également rester cohérents : la méthode emballer par exemple devra rester la même pour les deux machines (on utilise un emballage spécial que l'on ne veut pas changer).

On pourrait donc spécialiser `FabriqueTarte` en `FabriqueTarteNormale` et `FabriqueTarteSansGluten`.

43



***

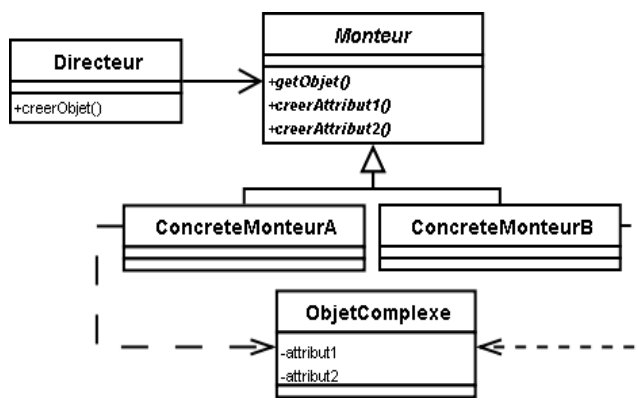
Builder : présentation

45

- Intention
  - Séparer la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction adapte la représentation selon les objets
- Exemple
  - Dans un carnet d'adresse, traiter de la même façon les personnes et les groupes
  - Manipuler ensemble des objets « proches » (générateur d'interface le cas des différentes fenêtres d'une IHM. Elles comportent des éléments similaires (titre, boutons), mais chacune avec des particularités (libellés, comportements)

Builder: diagramme

46



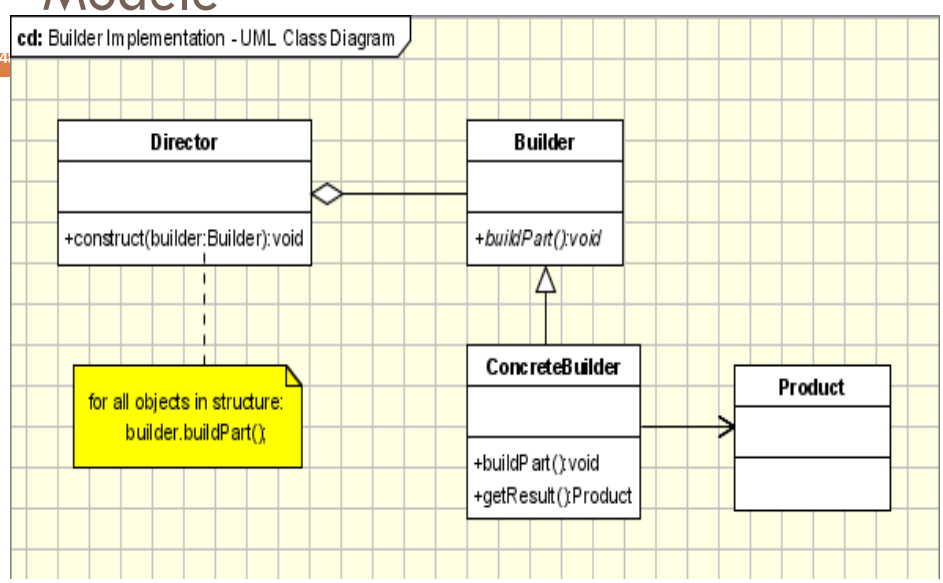
Builder: structure

47

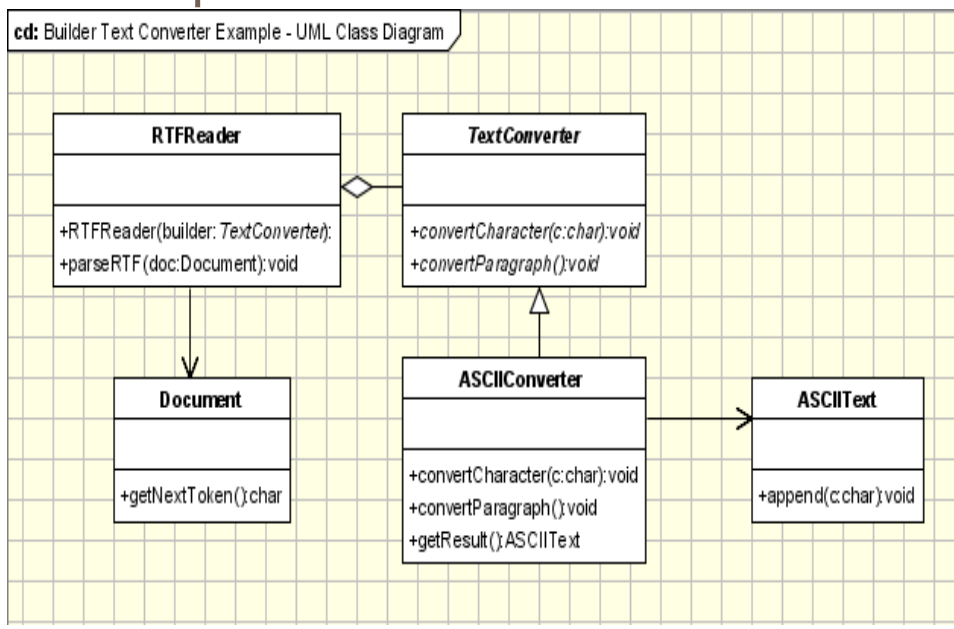
- Structure
  - ObjetComplexe : est la classe d'objet complexe à instancier.
  - Monteur : définit l'interface des méthodes permettant de créer les différentes parties de l'objet complexe.
  - ConcreteMonteurA et ConcreteMonteurB : implémentent les méthodes permettant de créer les différentes parties. Les classes conservent l'objet durant sa construction et fournissent un moyen de récupérer le résultat de la construction.
  - Directeur : construit un objet en appelant les méthodes d'un Monteur.
  - La partie cliente instancie un Monteur. Elle le fournit au Directeur. Elle appelle la méthode de contruction du Directeur.
- Conséquences
  - La représentations internes d'un objet peut varier.
  - La modularité : les builders sont indépendants

Modèle

48



Exemple



Application

50

- Le Client a besoin de convertir un doc du RTF au format ASCII
- Il appelle une methode *createASCIIText* qui a commme paramètre le document à convertir
- Cette méthode appelle la classe (concrete builder) *ASCIIConverter, sous classe de* (Builder)*TextConverter*,
- La classe (Director) RTFReader, parcourt le document et appelle la méthode du builder selon le token rencontré.
- La classe (product) ASCIIText, est construite étape par étape, en ajoutant les caractères convertis

Implémentation Java

51

```
//Abstract Builder
class abstract class TextConverter{
    abstract void convertCharacter(char c);
    abstract void convertParagraph();
}
// Product
class ASCIIText{
    public void append(char c){ //Implement the code
    here }
}
```

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

    /*converts a character to target representation and appends to
    the resulting*/
    object void convertCharacter(char c){
            char asciiChar = new Character(c).charValue();
                    //gets the ascii character
            asciiTextObj.append(asciiChar);
    }
    void convertParagraph(){}
    ASCIIText getResult(){
            return asciiTextObj;
    }
}
```

52

```
//This class abstracts the document object
class Document{
	static int value;
	char token;
	public char getNextToken(){
	//Get the next token
		return token;
	}
}
```

53

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

54

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

55

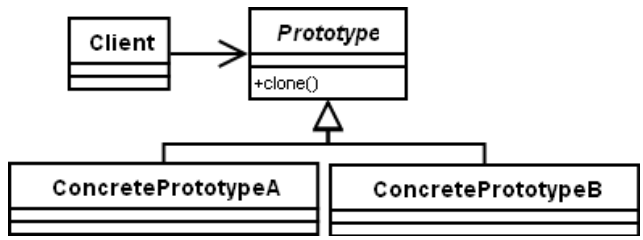
Prototype

56

- Intention
  - Dupliquer (cloner) un objet trop coûteux à créer
  - Créer des objets différents uniquement par le comportement. Le prototype sert de modèle.
- Exemple
  - Le système doit créer de nouvelles instances, mais il ignore de quelle classe. Il dispose cependant d'instances de la classe désirée.
  - Cela peut être le cas d'un logiciel de DAO comportant un copier-coller. L'utilisateur sélectionne un élément graphique (cercle, rectangle, ...), mais la classe traitant la demande de copier-coller ne connaît pas la classe exacte de l'élément à copier.
  - La solution est de disposer d'une duplication des instances (élément à copier : cercle, rectangle). La duplication peut être également intéressante pour les performances (la duplication est plus rapide que l'instanciation).

Prototype: diagramme

57



Prototype

58

- Structure
  - Prototype : définit l'interface de duplication de soi-même.
  - ConcretePrototypeA et ConcretePrototypeB : sont des sous-classes concrètes de Prototype. Elles implémentent l'inteface de duplication.
  - La partie cliente appelle la méthode clone() de la classe Prototype. Cette méthode retourne un double de l'instance.
- Conséquences
  - Création d'instances « différentes » sans créer de classe