IN201 : Examen

SUPAERO 2A

Christophe Garion <garion@supaero.fr>

2 mars 2009

Cet examen est composé de trois parties indépendantes. Vous avez 2h30 pour le faire. Les documents autorisés sont les photocopies distribuées en cours et les notes manuscrites que vous avez prises en cours. Il sera tenu compte de la rédaction. L'exercice 3 est un exercice de modélisation avec UML. Chaque exercice sera noté sur 8 points, mais le barème final peut être soumis à de légères modifications.

Remarque importante : dans tous les exercices, il sera tenu compte de la syntaxe UML lors de l'écriture de diagrammes. Ne perdez pas des points bêtement.

1 Un MVC générique

Cet exercice est inspiré de [5].

Le MVC (Modèle-Vue-Contrôleur) est un patron de conception classique utilisé depuis les années 1970 avec le langage de programmation Smalltalk. Nous l'avons utilisé lorsque nous avons construit des interfaces graphiques avec Java. Nous allons chercher dans cet exercice à proposer un cadre générique pour le MVC sous forme de classes et d'interfaces.

1. rappeler brièvement quels sont les principes de fonctionnement du MVC.
2. on considère l'interface graphique proposée sur la figure 1.

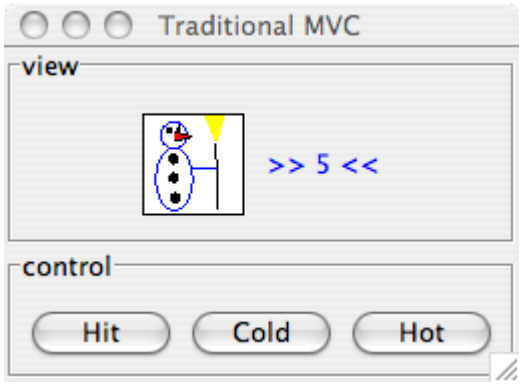

FIG. 1 – Vue de l'interface graphique

Sur cette interface, une instance de `JLabel` affiche à la fois un nombre correspondant au nombre de fois que l'on a appuyé sur le bouton « `Hit` » et une image correspond à une température fictive représentée par deux états, chaud et froid, contrôlés par deux boutons « `Cold` » et « `Hot` ». Cette instance de `JLabel` est contenue dans un `JPanel`.

La vue travaille donc avec deux modèles :

– un modèle `HitModel` qui représente le nombre de fois que l'on a appuyé sur le bouton ;
– un modèle `TemperatureModel` qui représente la température (chaud ou froid).

L'instance de `JLabel` qui affiche le nombre de fois que l'on appuyé sur le bouton « `Hit` » et l'icône correspondant à la température est en fait une instance d'une classe spécialisant `JLabel`, `LabelView`.

En utilisant un patron de conception abordé en cours, proposer sous la forme d'un diagramme de classe une architecture liant `LabelView` à `HitModel` et `TemperatureModel` et permettant de mettre à jour la vue lorsque l'un des modèles a changé. On supposera que plusieurs vues peuvent être intéressées par ces modèles.

3. on s'intéresse maintenant à la classe `TemperatureModel`. Cette classe représente un modèle de température qui peut être soit « chaud », soit « froid ». Depuis la version 5.0 de Java, on peut créer des types particuliers, appelés *énumérations*. Ces types représentent un ensemble fini de valeurs possibles. Par exemple, dans notre cas, une énumération `Temperature` représenterait l'ensemble de valeurs `{cold, hot}`. On supposera que l'on dispose d'une telle énumération. Le type `Temperature` s'utilise comme un type classique, mais on ne peut donner que `Temperature.HOT` ou `Temperature.COLD` comme valeur à une variable ou un paramètre typé par `Temperature` (cette contrainte est vérifiée à la compilation).

   Donner le code source de la classe `TemperatureModel` en respectant le diagramme proposé dans la question précédente et en utilisant l'énumération `Temperature`.

4. si l'on écrivait le code de `HitModel`, on se rendrait compte que l'on a beaucoup de code qui est en commun avec la classe `TemperatureModel` que l'on vient d'écrire. On va donc chercher à généraliser la notion de modèle grâce à une classe `Model`.

   Nous supposerons ici qu'un modèle est une classe encapsulant une propriété (une température, un nombre etc.) que l'on peut lire et modifier. Nous nous limitons à une seule propriété accessible[1]. On supposera également qu'un modèle doit respecter le patron de conception proposé dans la question 2.

   (a) on cherche à écrire une classe `Property` qui représente une propriété particulière. Une propriété a un type particulier qui peut être différent suivant le modèle étudié. Par exemple, la propriété de `HitModel` a un type entier puisque l'on compte un nombre de coups. `TemperatureModel` possède par contre une propriété qui est une température, donc soit `Temperature.COLD`, soit `Temperature.HOT`.

   Par contre, toutes les propriétés ont le même comportement, en particulier des implantations « identiques » pour les méthodes. Quel mécanisme va intervenir lors de la définition de `Property` ?

   (b) écrire la classe `Property` en Java.

   (c) écrire la classe `Model` en Java en utilisant la classe `Property`. Peut-on mettre une visibilité publique à l'attribut de la classe représentant la propriété ?

   (d) a-t-on encore besoin des classes `TemperatureModel` et `HitModel` ?

   (e) supposons maintenant que la propriété associée à `HitModel` soit un entier dont les valeurs possibles sont bornées à 10 (grâce à la méthode `Math.min(i,j)` qui renvoie le minimum entre `i` et `j`). On ne dispose que de la classe `Property<Integer>` qui caractérise une propriété représentant un nombre entier. Est-ce que la solution précédente fonctionne ? Quelle solution simple proposez-vous ? Cette solution peut-elle toujours être disponible (on pensera au cas où vous n'avez pas écrit personnellement `Model` par exemple) ? Dans le cas contraire, comment définir la propriété associée à `HitModel` et redéfinir son constructeur ?

5. on revient maintenant sur la classe `LabelView`. Les instances de cette classe vont être associées à un modèle dont la propriété sera une température et un modèle dont la propriété sera un entier.

   (a) lorsqu'une des propriétés va changer, l'instance de `LabelView` devra mettre à jour soit son texte (le nombre entier), soit son icône et sa couleur (pour représenter la température). Peut-on actuellement différencier ces deux mises-à-jour ?

   (b) proposer une solution simple pour pallier ce problème (explication textuelle).

[1] On peut bien sûr utiliser une collection comme propriété et ainsi avoir plusieurs « valeurs » dans une propriété.

2 Abstract Factory Method (ou le retour de la vengeance des tartes)

Dans cet exercice, vous serez amenés à écrire du code. Si vous devez écrire le code d'une méthode dont vous ne connaissez pas exactement le comportement, vous pourrez utiliser des appels à `System.out.println` pour afficher du texte correspondant à ce que devrait faire la méthode.

Dans l'examen de l'année dernière, nous nous étions intéressés à la modélisation d'une machine permettant de cuisiner des tartes aux fruits. Cette machine était modélisée par une classe `MachineTarte`. Nous voulions avoir deux types de machines, une qui cuisinait des tartes classiques (`MachineTarteSimple`) et une autre des tartes sans gluten (`MachineTarteSsGluten`). Une classe abstraite `Tarte` était spécialisée en différentes tartes, déclinées suivant les fruits que l'on mettait dessus et le fait qu'elles soient sans gluten ou pas.

Pour garantir le fait que la machine « sans gluten » ne construise que des tartes sans gluten, nous nous étions appuyés sur un patron de conception, *Factory Method* [2], qui nous proposait d'utiliser une méthode dans `MachineTarte` pour créer la tarte que l'on commandait. La solution ainsi construite est proposée sur la figure 2.

Dans ce diagramme, on dit que la méthode `creerTarte` est une *factory method* : elle permet de créer des tartes sans lier le type *réel* de tartes créé (avec ou sans gluten) à la classe `MachineTarte`. Lorsque l'on veut créer une tarte, on n'appelle donc pas directement `new TartePoiresSimple()` par exemple, mais on utilise une instance de `MachineTarteSimple` et on appelle `creerTarte("poires")` dessus. On est alors sûr que la tarte retournée sera de type « simple »[2]. On rappelle sur la figure 3 le patron de conception *factory*.

Le patron de conception *factory* nous permet d'appliquer le principe d'*inversion de dépendance* : on doit essayer de dépendre d'abstractions (classes abstraites ou interfaces) et non pas de classes concrètes. Dans notre cas :

- la classe `MachineTarte` ne dépend plus directement des types de tartes concrets considérés (`tartePoireSimple`, `tartePommesSsGluten` etc.), mais d'une classe abstraite `Tarte` ;
- de la même façon, les classes concrètes représentant les tartes dépendent de cette même abstraction, puisqu'elles en héritent.

Les sources des classes `MachineTarte` et `MachineTarteSimple` sont donnés respectivement sur les listings 1 et 2 et vous permettront de mieux comprendre le principe de ce patron de conception.

Listing 1 – La classe `MachineTarte`

```
/**
 * <code>MachineTarte</code> represente une machine fabriquant des tartes
 * aux fruits. La classe est abstraite, il faut l'etendre en implantant
 * la methode creerTarte pour pouvoir l'utiliser.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class MachineTarte {

    /**
     * <code>creerTarte</code> permet de creer une tarte.
     *
     * @param type une instance de <code>String</code> representant le type
     *             de tarte. Pour l'instant, seuls "pommes", "poires" et
```

[2] Évidemment, l'implantation de `creerTarte` garantit que l'on obtient bien le bon type de tarte.

```
     *             "prunes" sont connus.
     * @return la <code>Tarte</code> prete a etre travaillee
     */
    public abstract Tarte creerTarte(String type);

    /**
     * <code>commanderTarte</code> permet de commander une tarte particuliere.
     *
     * @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 = creerTarte(type);

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

        return tarte;
    }
}
```

Listing 2 – La classe `MachineTarteSimple`

```
/**
 * <code>MachineTarteSimple</code> est une machine permettant de
 * faire des tartes simples.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class MachineTarteSimple extends MachineTarte {

    // Implementation of MachineTarte

    /**
     * <code>creerTarte</code> permet de creer une tarte simple.
     *
     * @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 travaillee
     */
    public Tarte creerTarte(String type) {
        Tarte tarte = null;

        if (type.equals("pommes")) {
            tarte = new TartePommesSimple();
```

```
        } else if (type.equals("poires")) {
            tarte = new TartePoiresSimple();
        } else if (type.equals("prunes")) {
            tarte = new TartePrunesSimple();
        }

        return tarte;
    }
}
```

On s'intéresse maintenant plus précisément à la classe `Tarte`. La classe `Tarte` possède une méthode abstraite, `preparer`, qui représente les actions nécessaires à la préparation d'une tarte. Pour cela, on a besoin de différents ingrédients :

- de la pâte
- un liant
- le fruit utilisé qui est donné par le type de tarte

Pour que les tartes soient plus savoureuses, on rajoute du chocolat dans la tarte aux poires (et seulement celles là).

On s'est rendu compte que les tartes avec et sans gluten avait le même procédé de fabrication. Par contre, les ingrédients étaient différents dans ces deux familles de tartes :

- les tartes sans gluten utilisaient une pâte brisée, alors que les tartes avec gluten utilisaient une pâte sablée
- les tartes sans gluten utilisaient de la crème simple comme liant, alors que les tartes avec gluten utilisaient de la crème pâtissière
- les tartes sans gluten utilisent un chocolat garanti sans trace de gluten

On suppose que l'on dispose d'une classe abstraite pour chaque type d'ingrédients et de classes la spécialisant. Par exemple, on aura une classe abstraite `Pate` et deux sous-classes `PateBrisee` et `PateSablee`.

On cherche donc ici à garantir que l'on ne fabriquera des tartes qu'en utilisant des *ensembles* d'ingrédients compatibles (pas question par exemple d'utiliser de la pâte brisée avec de la crème pâtissière).

1. supposons que l'on utilise le patron de conception *factory* pour pouvoir nous abstraire des représentations concrètes des ingrédients. On va donc avoir une *factory* par type d'ingrédients nécessaire à la réalisation de la tarte. Est-ce que cette solution nous garantit la cohérence des ingrédients ?
2. pour pallier ce problème, on va utiliser un patron de conception particulier, *abstract factory method*. Ce patron est présenté sur la figure 4. Ce patron de conception fournit une interface permettant de construire un ensemble d'objets concrets interdépendants.

   Proposer un diagramme de classes adaptant le patron *abstract factory method* à notre problème. On utilisera une classe `IngredientsFactory` et deux classes `IngredientsFactorySimple` et `IngredientsSansGlutenFactory`.
3. écrire la classe `IngredientsFactory` en Java.
4. écrire la classe `IngredientsFactorySimple` en Java.
5. en utilisant `IngredientsFactory`, écrire la classe `Tarte` en Java. La méthode `preparer` reste-t-elle abstraite ?
6. écrire la classe `TartePoireSimple` en Java. Quel mécanisme permet de garantir que de la pâte sablée et de la crème pâtissière seront utilisées lors de l'appel à `preparer` sur une instance de `TartePoireSimple` ?

3 Algorithmes génétiques pour l'optimisation

Cet exercice est inspiré de [1]. Le lecteur curieux pourra également consulter [4, 3] pour plus de renseignements.

On cherche ici à écrire une application objet permettant d'optimiser des fonctions via des algorithmes génétiques. Les algorithmes génétiques sont des algorithmes de recherche ou d'optimisation utilisant le processus de sélection naturelle. Ils permettent de calculer la solution la plus optimale possible à un problème suivant un critère d'évaluation donné. Pour se faire, on va faire évoluer des populations de solutions à la manière de la théorie de l'évolution : en sélectionnant les solutions les meilleures (suivant le critère donné), en les mélangeant pour obtenir de nouvelles solutions et un ajoutant un degré d'aléa grâce à la mutation de certaines solutions.

Le *solver* permettant de résoudre un problème avec des algorithmes génétiques utilise plusieurs modules que nous détaillerons dans ce qui suit[3].

Le premier module concerne la modélisation du problème. La première chose à faire lorsque l'on utilise des algorithmes génétiques est de pouvoir représenter une solution à ce problème sous forme d'un chromosome constitué de gènes. Ces gènes représentent une donnée du problème et peuvent donc être typés : gène représentant un nombre, un entier, un réel, une chaîne de caractères, un booléen par exemple.

Les différents chromosomes sont ensuite regroupés dans une population qui est utilisée par le *solver*.

Pour que les algorithmes génétiques fonctionnent, on a besoin d'une fonction de *fitness* permettant de calculer sous forme d'une valeur entière quelle est la valeur d'une solution à partir de son chromosome. On peut bien sûr proposer plusieurs fonctions de *fitness*.

En utilisant cette fonction de *fitness*, on peut ensuite sélectionner quelles sont les solutions que l'on va retenir pour construire une prochaine population de chromosomes. Pour cela, différents mécanismes de sélection, appelés sélecteurs, sont disponibles : choix des meilleures valeurs absolues pour chaque chromosome, utilisation d'un système de tournoi entre chromosomes etc.

Enfin, on peut effectuer des opérations sur les chromosomes. Les opérations de base sont :
- la *reproduction* qui permet de copier une solution potentielle ;
- le *crossover* qui permet de mélanger les gènes de deux solution potentielles ;
- la *mutation* qui permet d'altérer aléatoirement la valeur d'un gène d'un chromosome.

On pourra bien sûr ajouter facilement de nouvelles opérations sur les chromosomes, comme des opérateurs de mutation gaussiens etc. Il faudra en tenir compte dans la conception du *solver*.

1. proposer un diagramme de classes d'analyse représentant le domaine étudié. On fera apparaître les classes, les relations entre classes, les noms de rôles et les multiplicités des associations, et on justifiera l'utilisation de classes abstraites et d'interfaces. On pourra utiliser quelques attributs et représenter quelques méthodes si nécessaire. Enfin, on cherchera à avoir une solution la plus générique possible, i.e. permettant l'ajout de nouvelles fonctionnalités facilement (fonctions de *fitness*, sélecteurs, opérations sur les chromosomes).
2. supposons que nous ayons une solution représentable par un seul gène de type entier et une classe particulière, `Generator`, qui permet d'effectuer un certain nombre d'opérations. Lorsque l'on cherche à faire muter un chromosome d'une population donnée, les étapes sont les suivantes :
   (a) on appelle la méthode `mutate` d'une instance connue de `Generator` permettant de savoir si l'on va faire muter le chromosome en question (nous nous placerons dans le cas où cette méthode renvoie **`true`**) ;
   (b) le générateur vérifie que l'on peut faire muter le chromosome en appelant une méthode *adhoc* de la classe `Chromosome` ;
   (c) si on peut le faire muter, le générateur créé un clone de ce chromosome ;

[3] Le terme module est utilisé ici de façon très générale, il ne faut pas le prendre au sens informatique.

(d) le générateur récupère une référence vers le gène de ce chromosome ;

(e) le générateur fait muter le gène.

Représenter le scénario précédent par un diagramme de séquence.

3. lorsque l'on veut résoudre un problème via des algorithmes génétiques, le *solver* fonctionne suivant l'algorithme suivant :
   - il initialise une population donnée.
   - il évalue chaque individu de la population en utilisant la fonction de *fitness*.
   - il sélectionne au vu de ces résultats les meilleurs individus.
   - il utilise les opérateurs de mutation et de croisement avec ces individus pour obtenir de nouveaux individus.
   - il remplace les plus mauvais individus de la population de départ pour les nouveaux individus ainsi produits.
   - il recommence à évaluer la population.
   - il s'arrête soit lorsqu'il a trouvé une solution, ou lorsque l'utilisateur lui demande d'arrêter, ou encore lorsqu'un certain nombre d'itérations de processus ont été faites.

   Représenter le fonctionnement du *solver* par un diagramme d'états-transitions.

Références

[1] JGap - Java Genetic Algorithms package. `http://jgap.sourceforge.net/`.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1994.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.

[4] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, , and A. Chircop. ECJ : a Java-based evolutionary computation research system. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[5] A. Vermeij. A generic MVC model in Java. `http://www.onjava.com/pub/a/onjava/2004/07/07/genericmvc.html`.

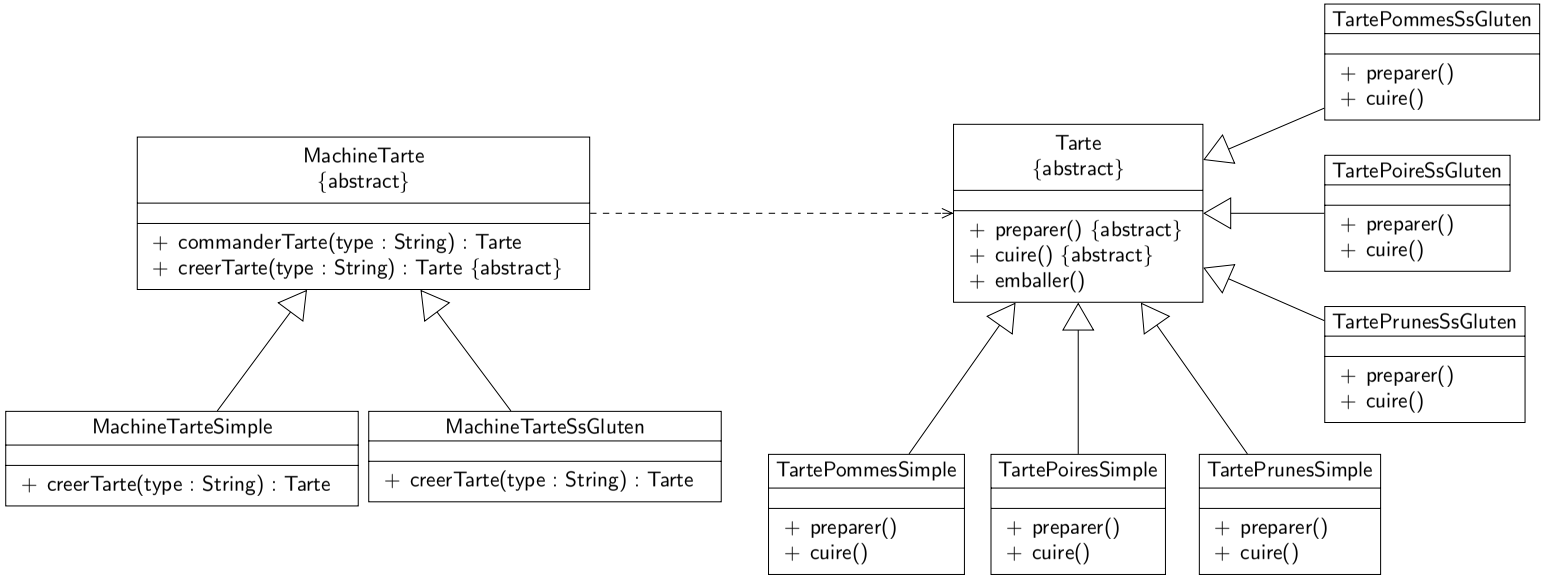

FIG. 2 – *Design pattern factory method* adapté au problème des tartes

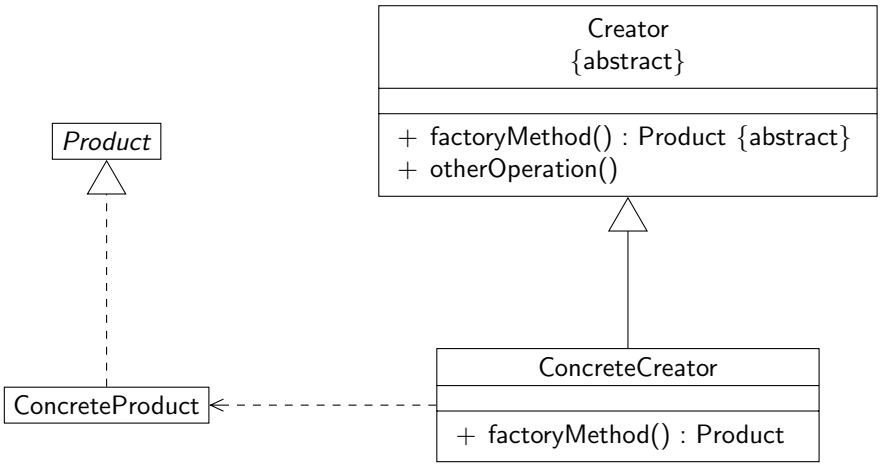

FIG. 3 – Le patron de conception *factory*

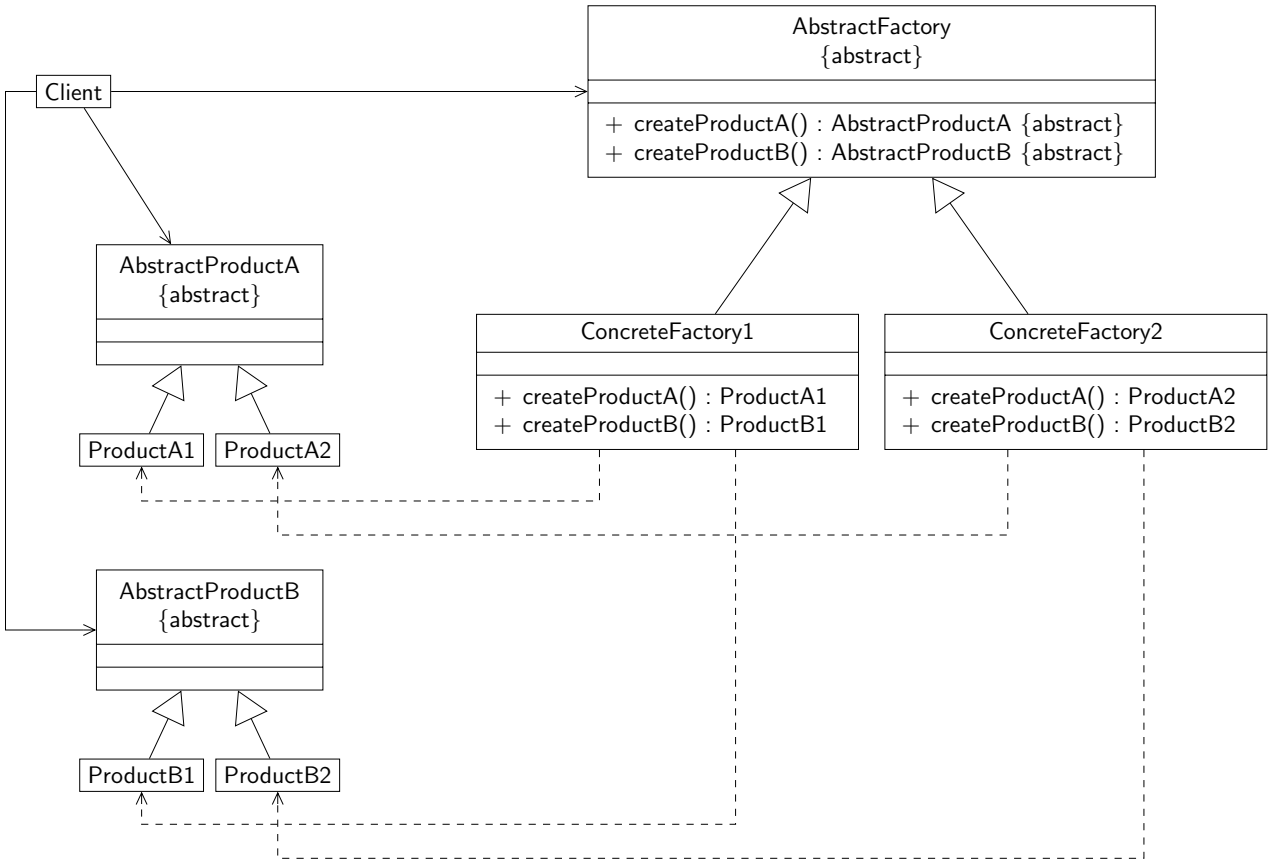

FIG. 4 – Le patron de conception *abstract factory method*