IN201 : Conception et Programmation Orientées Objet

Examen : problème

Cet examen de 1h15 est composé de 5 questions. Le barême indiqué pour chaque question l'est à titre indicatif et peut être modifié.
Les seuls documents autorisés pour cet examen sont :
– les notes distribuées en cours
– vos notes manuscrites
Les annales des examens des années précédentes sont interdites. Les téléphones portables doivent être éteints et rangés. L'utilisation d'un ordinateur durant l'examen est interdite.

FIGURE 1 : « Suit up ! »
Neil Harris Patrick as Barney Stinson, *How I Met Your Mother*, ©CBS 2005-2012

Après quelques échecs essuyés lors de précédentes soirées étudiantes, vous décidez d'étudier de plus près quelques techniques d'approches éprouvées par des professionnels. Justement, vous avez vu dans la première partie de l'examen qu'un certain Barney avait quelques astuces à vous apprendre... Après visionnage d'un grand nombre d'épisodes de « *How I met your mother ?* », vous vous attaquez à la construction d'une application permettant de simuler le comportement de Barney que vous pourrez installer sur votre téléphone portable afin de vous guider lors de la prochaine soirée : le *PortableBarneyFlirtingSimulator* (PBFS). La classe principale du simulateur s'appelera donc `PBFS`. On supposera qu'elle possédera une méthode `giveMeATrickRecipe` qui aura l'allure suivante :

```
public Trick giveMeATrickRecipe() {
  Trick trick = new Trick();

  trick.prepareAccessories();
  trick.prepareCostume();
  trick.prepareHookLines();

  return trick;
}
```

On peut donc demander au simulateur de fournir une des astuces de Barney. Avant de la donner, le simulateur préparera les éventuels accessoires, costume et phrases d'approche nécessaires à la réalisation de l'astuce.

(1 pt) **1.** on suppose dans un premier temps que l'on ne dispose que de trois astuces : le *Lorenzo von Matterhorn* (LVM), le *Have you met Barney ?* (HYMB) et le *Scuba Diver* (SD). On va donc rendre `Trick` abstraite et créer une hiérarchie de classes représentée sur la figure suivante :

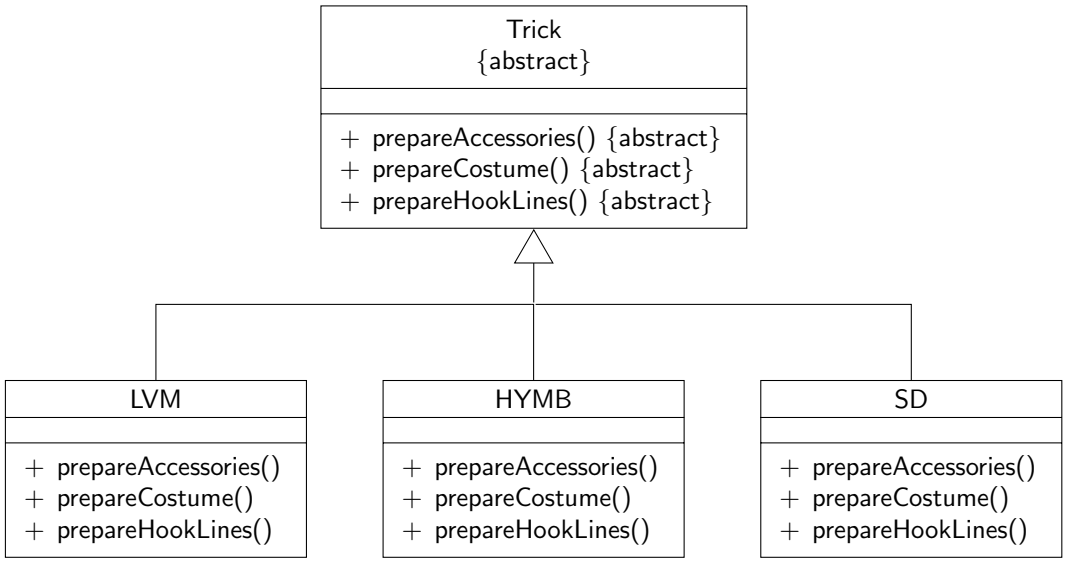


FIGURE 2 : Hiérarchie de classes représentant les astuces développées par Barney

On va modifier la méthode `giveMeATrickRecipe` pour utiliser les nouveaux types d'astuces : on va passer en paramètre de `giveMeATrickRecipe` une chaîne de caractères précisant la technique choisie (par exemple `"LVM"` ou `"HYMB"`). En fonction de cette chaîne de caractères, on construira l'astuce correspondante.
Modifier la méthode `giveMeATrickRecipe` en conséquence.

Solution :

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

```
public Trick giveMeATrickRecipe(String type) {
    Trick trick = null;

    if (type.equals("LVM")) {
        trick = new LVM();
    } else if (type.equals("HYMB")) {
        trick = new HYMB();
    } else if (type.equals("SD")) {
        trick = new SD();
    }

    trick.prepareAccessories();
    trick.prepareCostume();
    trick.prepareHookLines();

    return trick;
}
```

Points :

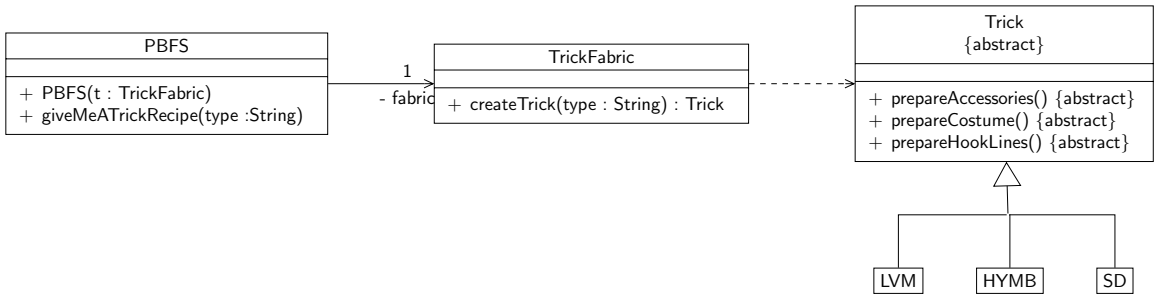
paramétrage de la méthode	0.25
déclaration et initialisation de `Trick`	0.25
utilisation de `equals`	0.25
structures **`if/else`**	0.25

2. Si l'on souhaite ajouter de nouvelles astuces, il va falloir modifier la méthode `giveMeATrickRecipe`. Or on souhaite fermer `giveMeATrickRecipe` à la modification [1].
Pour résoudre ce problème, on va déléguer la création des différentes astuces à une classe `TrickFabric` via une méthode `createTrick`.

(1 pt) (a) représenter sur un diagramme de classes les classes `PBFS`, `TrickFabric` et la hiérarchie des astuces.

Solution :

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



Points :

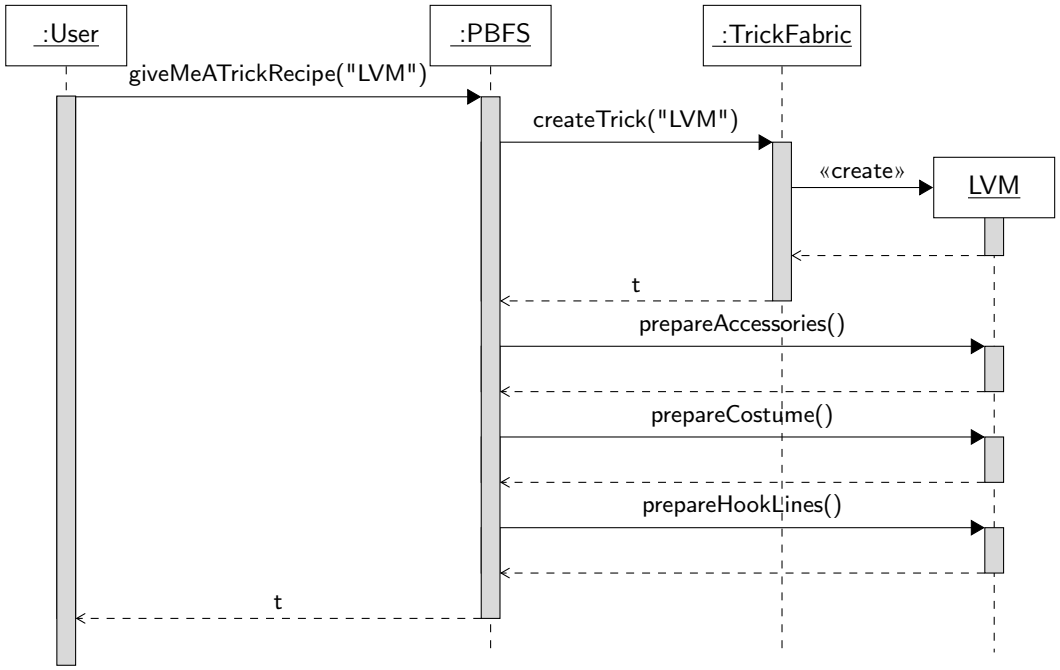
classe `PBFS`	0.25
classe `TrickFabric`	0.25
lien `PBFS/TrickFabric`	0.25
lien `TrickFabric/Trick`	0.25

(1 pt) (b) représenter sur un diagramme de séquence les interactions entre les objets lors de la demande de l'astuce « LVM ».

1. C'est en effet un bon principe de conception objet : lorsqu'une méthode est correcte, on n'« autorise » pas les modifications directes de la méthode, mais on permet de la redéfinir dans une sous-classe.

Solution :

Le diagramme de séquence est représenté sur la figure ci-après.



Points :

appel à `TrickFabric` depuis `PBFS` et retour	0,5
création de la technique	0,25
appels aux différentes méthodes de `Trick`	0,25

(1 pt) (c) donner le code Java des classes `TrickFabric` et `PBFS`.

Solution :

Le code est donné dans ce qui suit. Rien de bien particulier sur ces classes, il fallait juste ne pas oublier l'attribut de type `TrickFabric` dans `PBFS`.

```
/**
 * <code>TrickFabric</code> represente des objets dont le role est de
 * construire des instances de techniques.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TrickFabric {

    /**
     * <code>createTrick</code> permet de creer une technique. Attention, les
     * seuls types connus actuellement sont "LVM", "HYMB" et "SD".
     *
     * @param type une instance de <code>String</code> representant le type
     *             de technique a faire
     * @return la <code>Trick</code> demandee. Peut etre <code>null</code> si
     *         le type de technique n'existe pas
     */
```

```
    public Trick createTrick(String type) {
        Trick trick = null;

        if (type.equals("LVM")) {
            trick = new LVM();
        } else if (type.equals("HYMB")) {
            trick = new HYMB();
        } else if (type.equals("SD")) {
            trick = new SD();
        }

        return trick;
    }
}
```

```
/**
 * <code>PBFSSecond</code> est un simulateur de Barney utilisant
 * une fabrique de techniques.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class PBFSSecond {

    private TrickFabric fabric;

    /**
     * Creer une nouvelle instance de <code>PBFS</code>.
     *
     * @param fabric l'instance de <code>TrickFabric</code> que l'on veut
     *               utiliser pour construire des techniques
     */
    public PBFSSecond(TrickFabric fabric) {
        this.fabric = fabric;
    }

    /**
     * <code>giveMeATrickRecipe</code> permet a un utilisateur de demander
     * une technique d'un certain type.
     *
     * @param type une instance de <code>String</code> representant le type
     *             de technique. Pour l'instant, seuls "LVM", "HYMB" et
     *             "SD" sont connues.
     * @return la technique demandee
     */
    public Trick giveMeATrickRecipe(String type) {
        Trick trick = this.fabric.createTrick(type);

        trick.prepareAccessories();
        trick.prepareCostume();
        trick.prepareHookLines();
```

```
        return trick;
    }
}
```

Points :

classe `PBFS`	0.5
classe `TrickFabric`	0.5

(1 pt) **3.** On suppose maintenant que Barney possède deux grands types d'astuces : des astuces ne nécessitant pas d'aide extérieure (comme par exemple LVM) et des astuces nécessitant l'intervention d'un *wingman* (comme par exemple les astuces HYMB et SD). L'utilisateur de l'application PBFS choisira au départ la « famille » de techniques qu'il veut (avec ou sans *wingman*), puis le nom exact de la technique (LVM, HYMB, etc.). Dans la simulation, on veut donc avoir deux types de `PBFS` : un pour les techniques classiques et l'autre pour les techniques avec *wingman*.
On pourrait donc spécialiser `TrickFabric` en `TrickFabricWingman` et `TrickFabricWithoutWingman`.
Que pensez-vous de cette solution ? En particulier, comment créer un simulateur pour un type d'astuces particulier ? Contrôle-t-on l'utilisation qui sera faite des fabriques d'astuces ?

Solution :

Avec cette solution, on va créer une simulateur pour les techniques sans *wingman* par exemple en passant en paramètre du constructeur de `PBFS` une instance de `TrickFabricWithoutWingman`. On n'a donc aucun contrôle sur le type réel de fabrique que l'on utilise dans un simulateur. De plus, les fabriques peuvent être utilisées en dehors des simulateurs sans aucun problème.

Points :

explication	0.5

4. Pour pallier les problèmes soulevés précédemment, on décide de placer la méthode permettant de créer les astuces non pas dans une classe extérieure, mais dans les classes `PBFS`, `PBFSWingman` et `PBFSWithoutWingman`.
On décide donc d'utiliser le *design pattern* Factory Method représenté sur la figure 3.

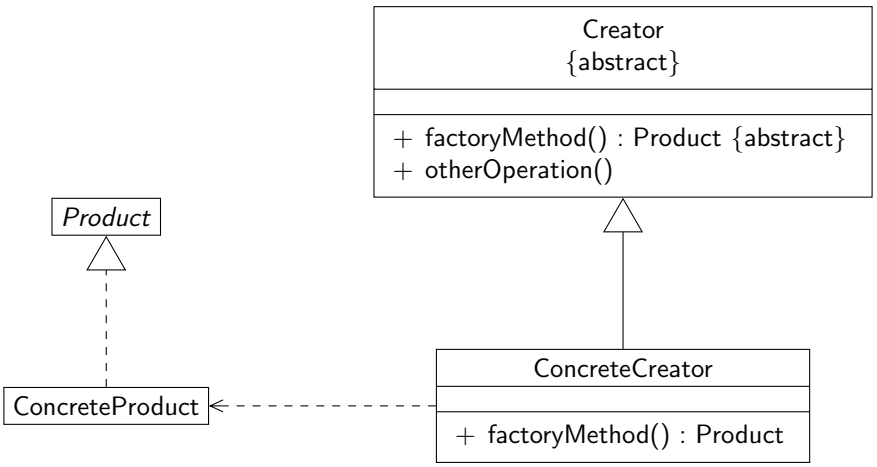


FIGURE 3 : Le *Design Pattern* Factory Method

Dans ce *pattern*, la classe `Creator` permet de créer des objets typés par `Product` grâce à la méthode `factoryMethod`. On spécialise ensuite cette classe dans des classes concrètes permettant de créer un ou plusieurs produits concrets en spécialisant la méthode `factoryMethod`.

(½ pt) (a) pourquoi utilise-t-on une interface (ou une classe abstraite) pour `Product` ?

Solution :

On utilise un type abstrait pour que les classes utilisant `Product` puissent se reposer sur une abstraction

et non pas une implantation particulière. L'objectif du *pattern* est de laisser les sous-classes décider du type réel de retour de la méthode *factory*.

Points :

explications	0.50

(½ pt) (b) peut-on mettre `ConcreteProduct` comme type de retour de `factoryMethod` dans `ConcreteCreator` ?

Solution :

C'est un problème classique de redéfinition de méthode. On sait qu'en Java le type de retour d'une méthode ne fait pas partie de sa signature. On ne peut donc pas avoir deux méthodes avec la même signature mais des types de retour différents.
`factoryMethod` est abstraite dans `Creator`. Il faut donc la redéfinir dans `ConcreteCreator`. On va donc écrire dans `ConcreteCreator` une méthode avec la même signature que `factoryMethod`. On devrait normalement avoir le même type de retour, i.e. `Product`, sinon une erreur serait détectée à la compilation. Mais comme `ConcreteProduct` hérite de `Product`, d'après le principe de substitution, on peut utiliser `ConcreteProduct` à la place de `Product`. Donc on peut bien utiliser `ConcreteProduct` comme type de retour de `factoryMethod` dans `ConcreteCreator`.

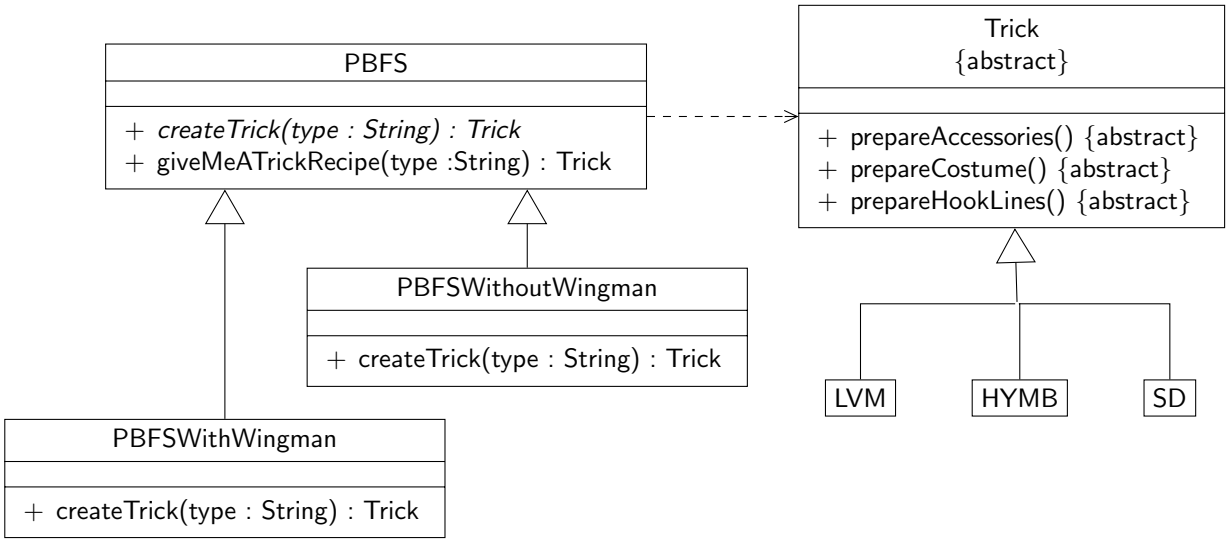
Points :

explications	1.00

(1 pt) (c) proposer une adaptation de ce *pattern* à notre problème via un diagramme de classes.

Solution :

Une adaptation simple est proposée sur la figure suivante. Rien de bien particulier ici.



Points :

`PBFS` abstraite et méthode abstraite	0.25
`PBFSWithWingman` et `PBFSWithoutWingman`	0.25
redéfinition de la méthode abstraite dans les deux classes	0.25
lien vers la hiérarchie des techniques	0.25

(1 pt) (d) écrire les classes `PBFS` et `PBFSWithoutWingman`.

Solution :

Rien de bien particulier non plus dans cette question. Vous trouverez les sources des deux classes ci-après.

```java
/**
 * <code>PBFS</code> represente un simulateur de Barney. La classe est abstraite,
 * il faut l'etendre en implantant la methode createTrick pour pouvoir l'utiliser.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class PBFS {

    /**
     * <code>createTrick</code> permet de creer une technique.
     *
     * @param type une instance de <code>String</code> representant le type
     *             de technique. Pour l'instant, seuls "LVM", "HYMB" et
     *             "SD" sont connus.
     * @return la <code>Trick</code> en question
     */
    public abstract Trick createTrick(String type);

    /**
     * <code>giveMeATrickRecipe</code> permet de demander une technique particuliere.
     *
     * @param type une instance de <code>String</code> representant le type
     *             de tarte. Pour l'instant, seuls "LVM", "HYMB" et
     *             "SD" sont connus.
     * @return la <code>Trick</code> prete a etre utilisee
     */
    public Trick giveMeATrickRecipe(String type) {
        Trick Trick = createTrick(type);

        Trick.prepareAccessories();
        Trick.prepareCostume();
        Trick.prepareHookLines();

        return Trick;
    }
}
```

```java
/**
 * <code>PBFSWithoutWingman</code> est un simulateur permettant de
 * demander des techniques sans wingman.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class PBFSWithoutWingman extends PBFS {

    // Implementation of PBFS

    public Trick createTrick(String type){
        Trick trick = null;
```

```
        if (type.equals("LVM")) {
            trick = new LVM();
        }

        return trick;
    }
}
```

Points :

classe `FPBS`	0.50
classe `FPBSWithoutWingman`	0.50

5. On s'intéresse maintenant à la classe `Trick`. La classe `Trick` possède une méthode abstraite, `prepareCostume`, qui représente les actions nécessaires à la préparation du costume de Barney pour réaliser l'astuce. Pour cela, on a besoin de différents habits :

- un pantalon
- un haut
- une paire de chaussures

Suivant la technique à réaliser, les habits à utiliser sont différents. Par exemple :

- pour la plupart des techniques, un pantalon et une veste de costume ainsi que des chaussures de ville suffisent
- pour le *Scuba Diver*, il faut un pantalon et un haut de plongée et une paire de palmes

On suppose que l'on dispose d'une classe abstraite pour chaque type d'habits et de classes la spécialisant. Par exemple, on aura une classe abstraite `Pants` et deux sous-classes `SuitPants` et `DivingPants`.
On cherche donc ici à garantir que le simulateur ne fournira des astuces qu'en utilisant des *ensembles* d'habits compatibles (pas question par exemple d'utiliser un pantalon de combinaison de plongée avec une veste de costume !).

(½ pt) (a) supposons que l'on utilise le patron de conception *factory* pour pouvoir nous abstraire des représentations concrètes des habits. On va donc avoir une *factory* par type d'habits nécessaires à la réalisation de la technique. Est-ce que cette solution nous garantit la cohérence des habits de Barney ?

Solution :

Cette solution ne garantit absolument pas la cohérence des habits. Même si on utilise des *factories* pour chacun des habits dans `Trick`, rien n'empêche quelqu'un qui écrit le code d'une technique particulière d'utiliser des habits « incompatibles ».

Points :

explications	0.50

(1½ pt) (b) 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.

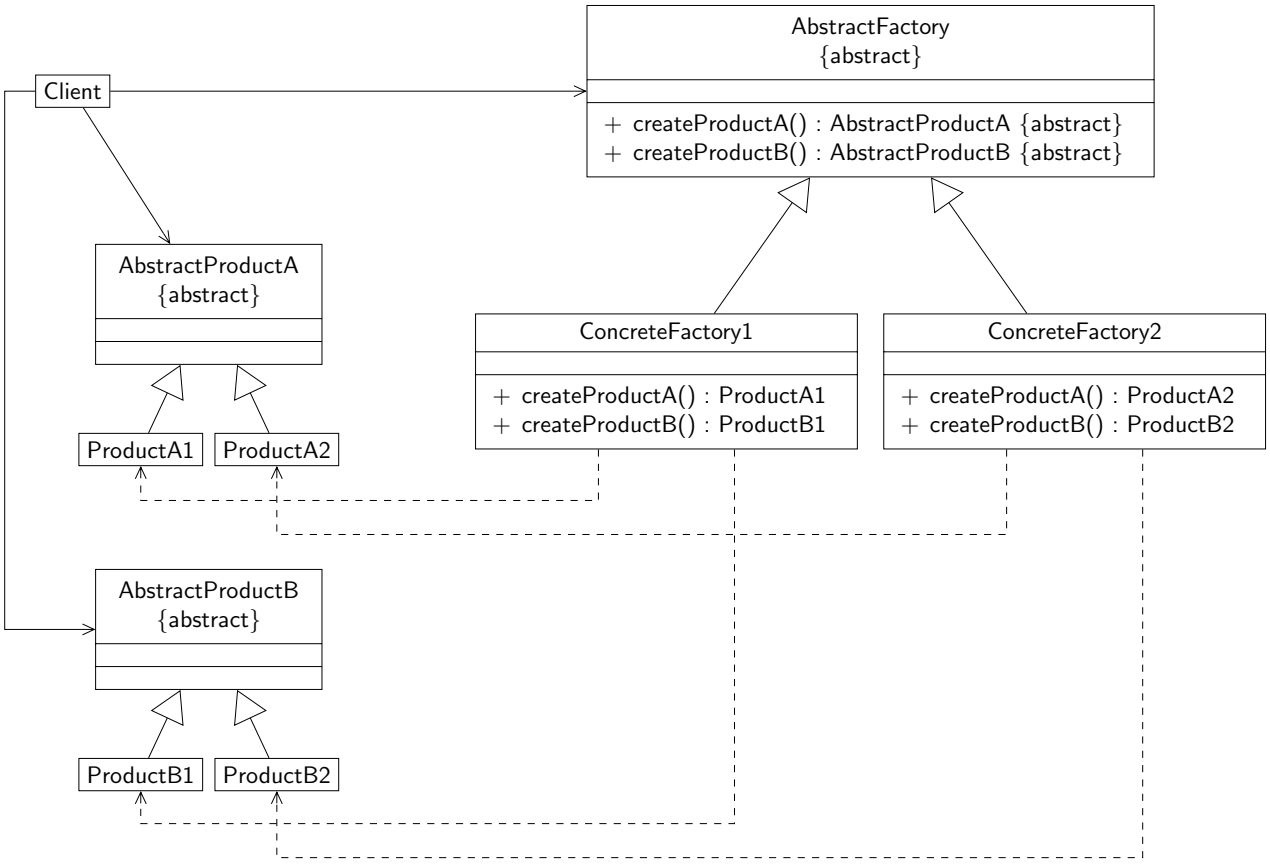
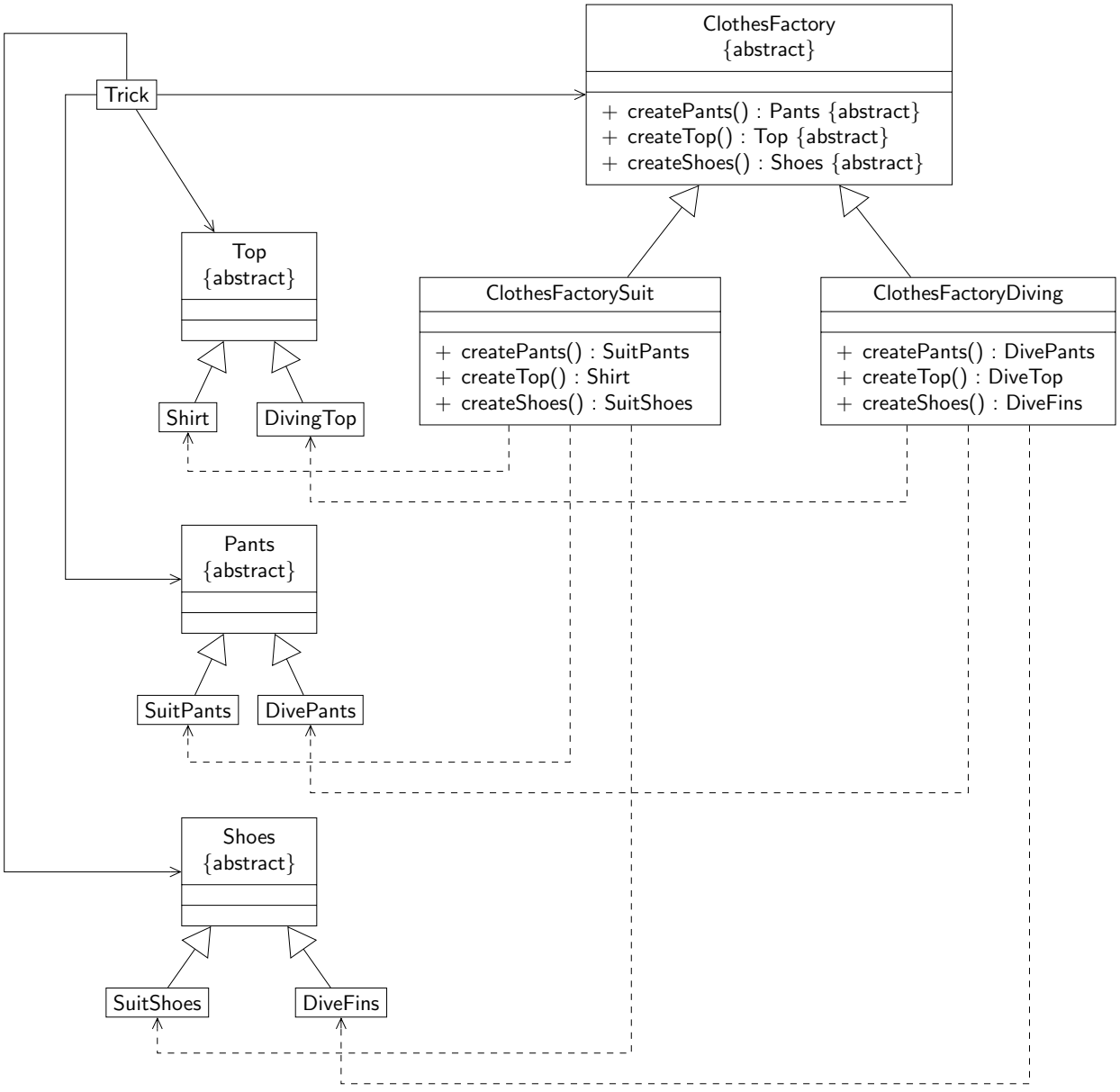


FIGURE 4 : Le patron de conception *abstract factory method*

Proposer un diagramme de classes adaptant le patron *abstract factory method* à notre problème. On utilisera une classe `ClothesFactory` et deux classes `ClothesFactorySuit` et `ClothesFactoryDiving`.

Solution :

Le diagramme est proposé sur la figure ci-dessous. Rien de bien particulier : les habits étaient ce qui était identifié comme des produits dans le patron. Le client est en fait ici la classe `Trick` qui a besoin des habits pour pouvoir se préparer.



Points :

`Trick` client	0.50
hiérarchie des habits	0.50
`ClothesFabricSuit` et `ClothesFabricDiving` et liens	0.50

Solution : Récapitulatif points

Question 1		1.00
	paramétrage de la méthode	0.25
	déclaration et initialisation de `Trick`	0.25
	utilisation de `equals`	0.25
	structures **`if`**/**`else`**	0.25
Question 2.a		1.00
	classe `PBFS`	0,25
	classe `TrickFabric`	0,25
	lien `PBFS`/`TrickFabric`	0,25
	lien `TrickFabric`/`Trick`	0,25
Question 2.b		1.00
	appel à `TrickFabric` depuis `PBFS` et retour	0,5
	création de la technique	0,25
	appels aux différentes méthodes de `Trick`	0,25
Question 2.c		1.00
	classe `PBFS`	0.5
	classe `TrickFabric`	0.5
Question 3		0.50
Question 4.a		0.50
Question 4.b		1.00
Question 4.c		1.00
	`PBFS` abstraite et méthode abstraite	0.25
	`PBFSWithWingman` et `PBFSWithoutWingman`	0.25
	redéfinition de la méthode abstraite dans les deux classes	0.25
	lien vers la hiérarchie des techniques	0.25
Question 4.d		1.00
	classe `FPBS`	0.50
	classe `FPBSWithoutWingman`	0.50
Question 5.a		0.50
Question 5.b		1.50
	`Trick` client	0.50
	hiérarchie des habits	0.50
	`ClothesFabricSuit` et `ClothesFabricDiving` et liens	0.50