IN201 : Corrigé de l'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.

   Nous avons étudié en cours le MVC. Son but est de pouvoir séparer les trois couches qui composent une application interagissant avec un utilisateur :
   - le *modèle* est la partie de l'application qui représente le domaine à modéliser et la logique métier, i.e. les API permettant d'accéder à la représentation du domaine et à la modifier.
   - la *vue* est la partie de l'application qui est la représentation visuelle de l'application.
   - le *contrôleur* gère les événements et change le modèle et éventuellement la vue en réponse à ces événements.

   La figure 1 présente les interactions existant entre les différentes parties d'un MVC.

2. on considère l'interface graphique proposée sur la figure 2.

   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.

   Il fallait bien sûr utiliser le patron de conception Observateur vu en cours. `LabelView` est un observateur des deux modèles `HitModel` et `TemperatureModel`. L'adaptation était triviale et est présentée sur la figure 3. J'ai choisi d'utiliser des instances d'`ArrayList` pour stocker les observateurs

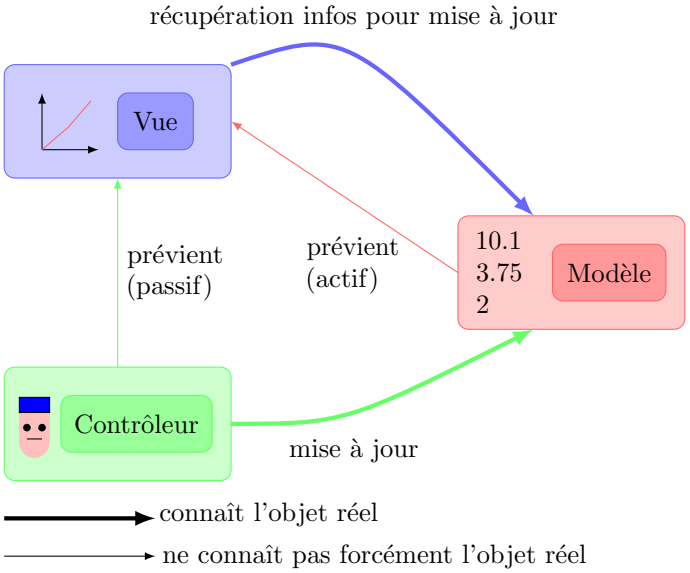

FIG. 1 – Interactions entre les différentes parties d'un MVC

de chaque modèle. J'ai également choisi directement d'utiliser des interfaces, mais on pouvait choisir une classe abstraite pour `Observable` du fait que les classes représentant les modèles n'héritent d'aucune classe (ce serait même une excellent solution). Enfin, j'ai choisi d'utiliser des termes en anglais pour faire plus sérieux.

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`.

   Le code source de `TemperatureModel` est présenté sur le listing 1. Pas de remarques particulières, mis à part le fait que j'ai introduit l'énumération `Temperature` directement dans `TemperatureModel`. On pourra remarquer que l'on peut utiliser la classe `java.util.Observable` et l'interface `java.util.Observer` qui existent déjà. Voir leur documentation javadoc pour plus de détails.

Listing 1 – La classe `TemperatureModel` avec observateurs

```
package fr.supaero.mvc.obs;

import java.util.ArrayList;

/**
 * TemperatureModel is a class modelling a temperature.
 *  It is also observable.
```

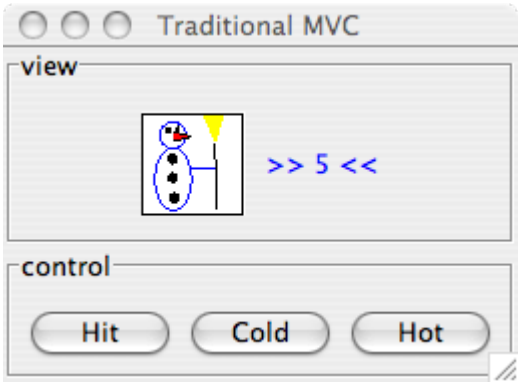

FIG. 2 – Vue de l'interface graphique

```
 *
 *
 * Created: Sun Jul  6 15:21:34 2008
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */

public class TemperatureModel implements Observable {

    public enum Temperature {COLD, HOT};

    private Temperature temp;
    private ArrayList<Observer> observers;

    /**
     * Creates a new <code>TemperatureModel</code> instance.
     *
     * @param temp_ the model's initial temperature (Temperature.COLD or
     *              Temperature.HOT only!)
     */
    public TemperatureModel(Temperature temp_) {
        this.temp = temp_;
        this.observers = new ArrayList<Observer> ();
    }

    /**
     * Gets the value of the temperature
     *
     * @return the value of the temperature
     */
    public final Temperature getTemperature() {
        return this.temp;
    }
```

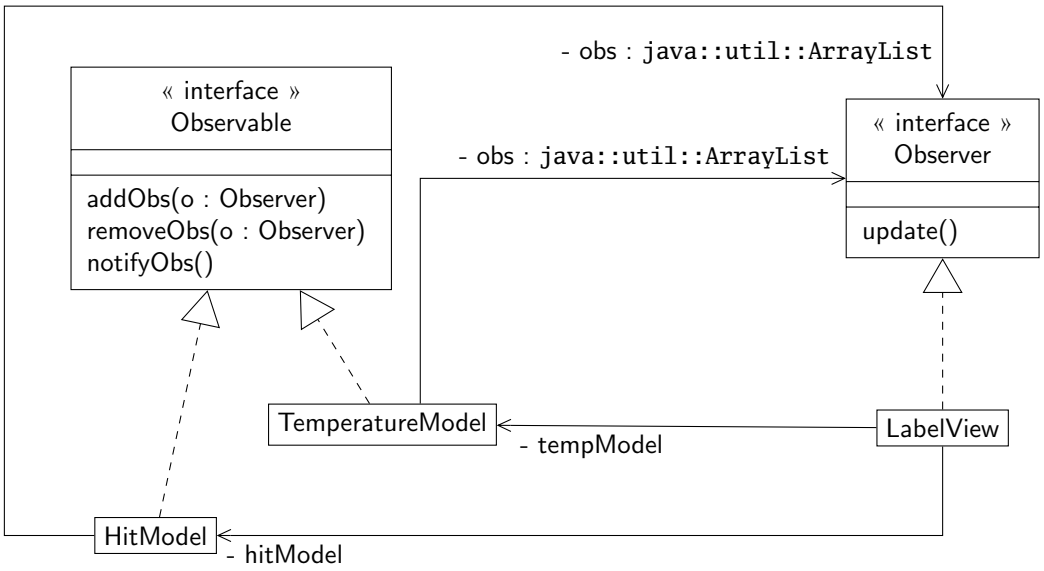

FIG. 3 – Le patron de conception Observateur adapté au problème

```
    /**
     * Sets the temperature
     *
     * @param argTemp value of the temperature (Temperature.COLD or
     *                Temperature.HOT only!)
     */
    public final void setTemperature(final Temperature argTemp) {
        this.temp = argTemp;
        this.notifyObs();
    }

    // Implementation of fr.supaero.mvc.obs.Observable

    public void addObs(Observer o) {
        this.observers.add(o);
    }

    public void removeObs(Observer o) {
        this.observers.remove(o);
    }

    public void notifyObs() {
        for (Observer o : this.observers) {
            o.update();
        }
    }
}
```

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` ?

   On cherche ici à avoir une « famille » de classes qui ont le même comportement, mais qui ne travaillent pas sur les mêmes types de données. Il s'agit donc ici d'utiliser le mécanisme de *généricité* de Java.

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

   Le source de la classe `Property` est donné sur le listing 2. Rien de bien particulier ici, il fallait juste bien utiliser le type générique. J'ai choisi d'appeler l'accesseur `get` et le modifieur `set`, car le contexte permettait de raccourcir les noms des méthodes et d'omettre `PropertyValue`.

Listing 2 – La classe `Property`

```
package fr.supaero.mvc.generics;

/**
 * Property is a class representing a property to be
 * associated with a model.
 *
 *
 * Created: Sun Jul  6 15:43:03 2008
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Property<T> {

    private T propertyValue;

    /**
     * Creates a new <code>Property</code> instance.
     *
     * @param propertyValue_ the initial value of the property
     */
    public Property(T propertyValue_) {
        this.propertyValue = propertyValue_;
    }
```

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

```
    /**
     * Get the value of the property
     *
     * @return the current value of the property
     */
    public T get() {
        return propertyValue;
    }

    /**
     * Set the value of the property
     *
     * @param propertyValue_ the new value of the property
     */
    public void set(T propertyValue_) {
        this.propertyValue = propertyValue_;
    }
}
```

(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 ?

La classe `Model` est présentée sur le listing 3. Rien de bien particulier encore, il fallait juste bien penser utiliser un paramètre de type générique puisque l'on avait une propriété comme attribut.

La question sur la visibilité est bien évidemment reliée au principe d'encapsulation qui énonce que l'on doit protéger l'état d'un objet et donc ne pas permettre un accès direct à l'état de cet objet. Dans notre cas, on peut amener deux solutions pour satisfaire le principe d'encapsulation :

- soit on applique strictement le principe vu en cours qui permet de respecter « directement » le principe d'encapsulation et on met l'attribut *privé* : on ne peut pas modifier depuis l'extérieur de la classe la valeur de cet attribut.
- la valeur réelle de la propriété étant protégée dans la classe `Property` (car l'attribut `propertyValue` y est privé), on peut se permettre de mettre une visibilité publique à l'attribut que j'ai appelé `property` dans la classe `Model`. Reste un problème : si `property` est publique, on peut affecter un autre objet de type `Property` à cet attribut. Pour pallier ce problème, il suffit de rendre l'attribut **`final`** ce qui permet de ne l'affecter qu'une seule fois.

Dans notre cas, il faut également tenir compte du fait que l'on doit pouvoir appeler la méthode `notifyObs` lorsque l'on change la valeur de la propriété. Il faut donc conserver l'attribut `property` privé pour pouvoir avoir une méthode `setPropertyValue` dans `Model` dans laquelle on fait appel à `notifyObs`[2].

Listing 3 – La classe `Model`

```
package fr.supaero.mvc.generics;

import fr.supaero.mvc.obs.Observable;
import fr.supaero.mvc.obs.Observer;
import java.util.ArrayList;
```

[2]On aurait pu utiliser la seconde solution en utilisant des classes internes, mais cela dépasse le cadre de l'examen. Vous pourrez trouver plus de détails sur [5].

```java
/**
 * <code>Model</code> is a generic class representing a model
 * with a single property.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class Model<T> implements Observable {

    private Property<T> property;
    private ArrayList<Observer> observers;

    public Model(Property<T> initialProperty) {
        this.property = initialProperty;
        this.observers = new ArrayList<Observer>();
    }

    public final T getPropertyValue() {
        return this.property.get();
    }

    public final void setPropertyValue(T value) {
        this.property.set(value);
        this.notifyObs();
    }

    // Implementation of fr.supaero.mvc.obs.Observable

    public void addObs(Observer o) {
        this.observers.add(o);
    }

    public void removeObs(Observer o) {
        this.observers.remove(o);
    }

    public void notifyObs() {
        for (Observer o : this.observers) {
            o.update();
        }
    }
}
```

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

On se rend que l'on n'a plus besoin des deux classes `TemperatureModel` et `HitModel`. Il suffit d'utiliser une instance de `Model<Integer>` ou de `Model<Temperature>` directement.

(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 ?

Si l'on veut contraindre les valeurs possibles de l'entier propriété de `HitModel`, on ne peut plus simplement utiliser une instance de `Model<Integer>`, car la propriété `Property<Integer>` ne permet pas l'utilisation de contrainte. Il faut donc réécrire cette fois-ci une classe `HitModel`.

Si l'on écrit la classe `HitModel`, on se rend compte qu'elle ne contient plus grand chose : elle hérite de `Model<Integer>` et va simplement contenir un constructeur qui va permettre d'initialiser la valeur de la propriété.

La solution la plus simple pour contraindre la valeur de l'entier est donc de redéfinir dans `HitModel` la méthode `setPropertyValue` de `Model` pour calculer le minimum du paramètre et de 10 et de passer le résultat en paramètre de la méthode `setPropertyValue` de `Model` via **`super`**.

Il se peut très bien que l'on ne puisse pas redéfinir la méthode `setPropertyValue` : il suffit que celle-ci soit déclarée **`final`** dans `Model`. Le fait de ne pas disposer du code source n'est bien évidemment pas un obstacle pour redéfinir une méthode qui n'est pas **`final`**.

Vous remarquerez que dans le code de `Model` que j'ai développé, j'ai mis la méthode `setPropertyValue` **`final`** : cela permet en effet d'éviter une redéfinition de la méthode dans laquelle on oublierait d'appeler `notifyObs`.

Dans ce cas, une solution simple est la suivante : lorsque l'on fait appel au constructeur de `Model` dans le constructeur de `HitModel`, on lui passe en paramètre une instance de `Property<Integer>`. C'est à ce moment qu'on peut utiliser le mécanisme de *classes anonymes* pour définir « à la volée » une nouvelle classe dans laquelle on a redéfini la méthode `set`. Celle-ci se contente d'appeler la méthode `set` de `Property<Integer>` en calculant le minimum de l'entier passé en paramètre et de 10.

Le code source est donné sur le listing 4

Listing 4 – La classe `HitModel` avec propriété contrainte

```
package fr.supaero.mvc.generics;

/**
 * <code>HitModel</code> is a model class with an integer property. This integer
 * is constrained and cannot be greater than 10.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class HitModel extends Model<Integer> {

    public HitModel() {
        super(new Property<Integer> (0) {
                public void set(Integer i) {
                    super.set(Math.min(10, i));
                }
            });
    }
```

```
}
```

On peut remarquer que l'on pourrait également lever une exception dans la redéfinition de `set` lorsque la valeur introduite est supérieure à 10^3. Dans ce cas, cette exception doit être hors contrôle, car la méthode redéfinie ne peut pas déclarer d'exception si la méthode `set` originale n'en déclare pas.

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 ?

   Non, il n'est pas possible de différencier les deux mises-à-jour car les deux modèles vont faire appel à la même méthode de `LabelView`, soit `update`.

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

   Pour pallier ce problème très simplement, il suffit de faire deux observateurs différents : un observateur pour l'instance de `HitModel` qui devra mettre à jour le texte de l'instance de `LabelView` et un observateur pour `TemperatureModel` qui devra mettre à jour l'icône et la couleur de l'instance de `LabelView`.

   On peut également passer en paramètre de la méthode `update` l'objet qui est responsable de la notification (instance de `HitModel` ou de `TemperatureModel`). C'est cette solution qui a été choisie par exemple dans la classe `java.util.Observer` et l'interface `java.util.Observable`.

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 4.

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 »[4]. On rappelle sur la figure 5 le patron de conception *factory*.

[3] Avec la solution précédente, l'utilisateur peut ne pas se rendre compte qu'il introduit une valeur incorrecte. Le fait

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 5 et 6 et vous permettront de mieux comprendre le principe de ce patron de conception.

Listing 5 – 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
     *             "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;
```

d'utiliser une valeur par défaut en cas de problème est d'ailleurs une solution qui est loin d'être la meilleure !

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

```
    }
}
```

Listing 6 – 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 ?

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

2. pour pallier ce problème, on va utiliser un patron de conception particulier, *abstract factory method*. Ce patron est présenté sur la figure 6. 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`.

   Le diagramme est proposé sur la figure 7. Rien de bien particulier : les ingrédients étaient ce qui était identifié comme des produits dans le patron. Le client est en fait ici la classe `Tarte` qui a besoin des ingrédients pour se faire.

3. écrire la classe `IngredientsFactory` en Java.

   Il n'y avait pas de problème particulier. Il fallait bien mettre les méthodes abstraites et ne pas oublier de mettre également la classe abstraite. Le source de la classe est donné sur le listing 7.

Listing 7 – La classe `IngredientsFactory`

```
package fr.supaero.factory;

/**
 * <code>IngredientsFactory</code> est une factory abstraite pour des
 * ingredients pour les tartes.
 *
 *
 * Created: Tue Jul  8 11:00:09 2008
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class IngredientsFactory {

    /**
     * <code>creerPate</code> permet de creer de la pate pour
     * une tarte.
```

```
     *
     * @return une instance de <code>Pate</code>
     */
    public abstract Pate creerPate();

    /**
     * <code>creerLiant</code> permet de creer du liant pour
     * une tarte.
     *
     * @return une instance de <code>Liant</code>
     */
    public abstract Liant creerLiant();

    /**
     * <code>creerChocolat</code> permet de creer du chocolat pour
     * une tarte.
     *
     * @return une instance de <code>Chocolat</code>
     */
    public abstract Chocolat creerChocolat();
}
```

4. écrire la classe `IngredientsFactorySimple` en Java.

   Là encore, il n'y avait pas de problème particulier. Il fallait bien évidemment redéfinir les méthodes abstraites. On pouvait choisir de spécialiser les types de retour des méthodes, ce qui est autorisé par le principe de redéfinition. Le source de la classe est donné sur le listing 8.

Listing 8 – La classe `IngredientsFactorySimple`

```
package fr.supaero.factory;

/**
 * <code>IngredientsFactorySimple</code> premet d'obtenir des ingredients
 * pour une tarte simple.
 *
 *
 *
 * Created: Tue Jul  8 11:39:21 2008
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class IngredientsFactorySimple extends IngredientsFactory {

    /**
     * <code>creerPate</code> permet de creer de la pate sablee
     * pour une tarte.
     *
     * @return une instance de <code>PateSablee</code>
     */
```

```
    public PateSablee creerPate() {
        return new PateSablee();
    }

    /**
     * <code>creerLiant</code> permet de creer de la creme patissiere
     * pour une tarte.
     *
     * @return une instance de <code>CremePatissiere</code>
     */
    public CremePatissiere creerLiant() {
        return new CremePatissiere();
    }

    /**
     * <code>creerChocolat</code> permet de creer du chocolat simple.
     *
     * @return une instance de  <code>ChocolatSimple</code>
     */
    public ChocolatSimple creerChocolat() {
        return new ChocolatSimple();
    }
}
```

5. en utilisant `IngredientsFactory`, écrire la classe `Tarte` en Java. La méthode `preparer` reste-t-elle abstraite ?

   La méthode `preparer` n'est plus abstraite : on sait en effet l'écrire complètement, *même si on utilise des références typées par des classes abstraites* (`Liant` et `Pate`). On remarquera qu'on n'utilise pas de chocolat, car seules les tartes aux poires l'utilisent. Pour que ces dernières aient accès à la *factory* afin d'obtenir du chocolat, j'ai choisi de mettre l'attribut `factory` protégé (on aurait également pu mettre l'attribut privé et créer un accesseur protégé à cet attribut). Le code source de la classe est donné sur le listing 9.

Listing 9 – La classe `Tarte`

```
package fr.supaero.factory;

/**
 * <code>Tarte</code> est une classe representant une tarte.
 *
 *
 * Created: Tue Jul  8 12:04:04 2008
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public abstract class Tarte {

    protected IngredientsFactory factory;
```

```
    /**
     * Creer une instance de <code>Tarte</code> en utilisant une factory
     * d'ingredients particuliere.
     *
     * @param factory_ une instance de <code>IngredientsFactory</code> qui
     *                 va servir a produire les ingredients necessaires pour
     *                 faire la tarte
     */
    public Tarte(IngredientsFactory factory_) {
        this.factory = factory_;
    }

    public void preparer() {
        Pate p = this.factory.creerPate();
        System.out.println("J'utilise cette pate : " + p);

        Liant l = this.factory.creerLiant();
        System.out.println("J'utilise ce liant : " + l);

        System.out.println("Je melange le tout.");
    }

    public abstract void cuire();

    public void emballer() {
        System.out.println("J'emballe la tarte...");
    }
}
```

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` ?

   Le code source de la classe est donné sur le listing 9. Il fallait redéfinir la méthode `preparer` car on devait utiliser du chocolat. Il ne fallait pas oublier l'appel au constructeur de `Tarte` pour initialiser la *factory* d'ingrédients. Enfin, la méthode `cuire` devait être définie car elle était abstraite dans `Tarte`. Si l'on regarde la méthode `preparer` de la classe `Tarte`, le liant et la pâte utilisés sont `abstraits` (typés par `Liant` et `Pate`). Or, lorsque l'on va appeler `preparer` sur `TartePoireSimple`, on utilisera de la pâte sablée et de la crème pâtissière car on utilise la `factory` correspondant aux tartes simples. C'est le principe de *liaison tardive* qui nous garantit que ce sont les objets réels qui seront utilisés lors de l'appel à `preparer`.

Listing 10 – La classe `TartePoireSimple`

```
package fr.supaero.factory;

/**
 * <code>TartePoireSimple</code> represente une tarte aux poires simple.
 *
 *
 * Created: Wed Jul  9 15:34:47 2008
```

```
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class TartePoireSimple extends Tarte {

    /**
     * Creer une instance de <code>TartePoireSimple</code> en utilisant une
     * factory d'ingredients simple.
     *
     * @param factory_ une instance de <code>IngredientsFactorySimple</code>
     *                 qui va servir a produire les ingredients necessaires
     *                 pour faire la tarte
     */
    public TartePoireSimple(IngredientsFactorySimple factory_) {
        super(factory_);
    }

    public final void preparer() {
        super.preparer();

        Chocolat c = this.factory.creerChocolat();
        System.out.println("J'utilise ce chocolat : " + c);
    }

    public final void cuire() {
        System.out.println("Je cuis pendant 30 minutes...");
    }
}
```

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[5].

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.

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

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).

   Une solution est proposée sur la figure 9. Quelques remarques :

   - la relation de composition entre `Solver` les autres classes nous indique que `Solver` est la classe « principale » de l'application.
   - j'ai choisi également d'utiliser une relation de composition entre `Population` et `Chromosome` et `Chromosome` et `Gene`, car on utilisera plutôt un mécanisme de clonage pour copier un gène ou un chromosome plutôt qu'un partage de référence.
   - `Gene` est une interface, car je ne voyais pas de méthode *concrète* à mettre dans une éventuelle classe abstraite. L'utilisation d'une classe abstraite était toutefois possible. Un certain nombre de classes concrètes réalisent ensuite cette interface, chacune d'entre elles représentant un type de gène particulier. On remarquera que j'ai choisi d'introduire un niveau de hiérarchie intermédiaire avec `GeneNombre` qui est une classe abstraite.
   - on demandait à ce que l'on puisse ajouter ou modifier facilement des fonctions de *fitness*, des sélecteurs et des opérateurs sur les chromosomes. Comme ces opérations se font normalement sur des chromosomes, on aurait tendance à proposer une méthode dans `Chromosome` pour calculer la fonction de *fitness*, plusieurs méthodes dans `Chromosome` pour les opérations sur les chromosomes et une autre dans `Chromosome` pour sélectionner ou non le chromosome en question. Malheureusement, cette approche a ses limites : lorsque l'on va vouloir spécialiser la classe `Chromosome` pour redéfinir une nouvelle méthode de `fitness` ou un nouvel opérateur, on va se retrouver avec un nombre très important de classes. De plus, du code sera dupliqué dans ces classes : si par exemple on a deux sous-classes de `Chromosome` qui redéfinissent chacune une méthode de *fitness*, si l'on veut ajouter un seul opérateur sur les chromosomes, il va falloir spécialiser ces deux classes en ajoutant la même méthode.
     Pour pallier ce problème, on peut utiliser le patron de conception *Stratégie*. Ce patron, présenté sur la figure 8, utilise la composition plutôt que la spécialisation pour proposer différents algorithmes. Le principe est de représenter une méthode par une classe et de spécialiser cette classe lorsque l'on veut définir un nouveau comportement pour cette méthode.
     Ce patron de conception est utilisé ici pour définir les notions de fonction de *fitness*, de sélecteur, et d'opérateur sur les chromosomes. On remarquera que j'ai choisi d'utiliser une classe abstraite

pour `FonctionFitness`, car on peut écrire la méthode `getPopulationEval` : il suffit d'utiliser la méthode `evaluate` sur chacun des 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 ;

   (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.

   Rien de bien difficile ici, le diagramme est donné sur la figure 10. Je suppose qu'il existe un constructeur de `Chromosome` qui peut prendre un chromosome en paramètre pour le copier.

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.

   Un diagramme de machine d'états est proposé sur la figure 11. J'ai supposé que l'on disposait d'un booléen `cond` qui permettait de spécifier si l'arrêt du *solver* ou la découverte d'une solution étaient prioritaires par rapport au rebouclage (pour éviter l'indéterminisme). Il y a bien sûr des cas de indéterminisme possibles, mais ils amènent tous au même état, j'ai donc choisi de les ignorer.

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.

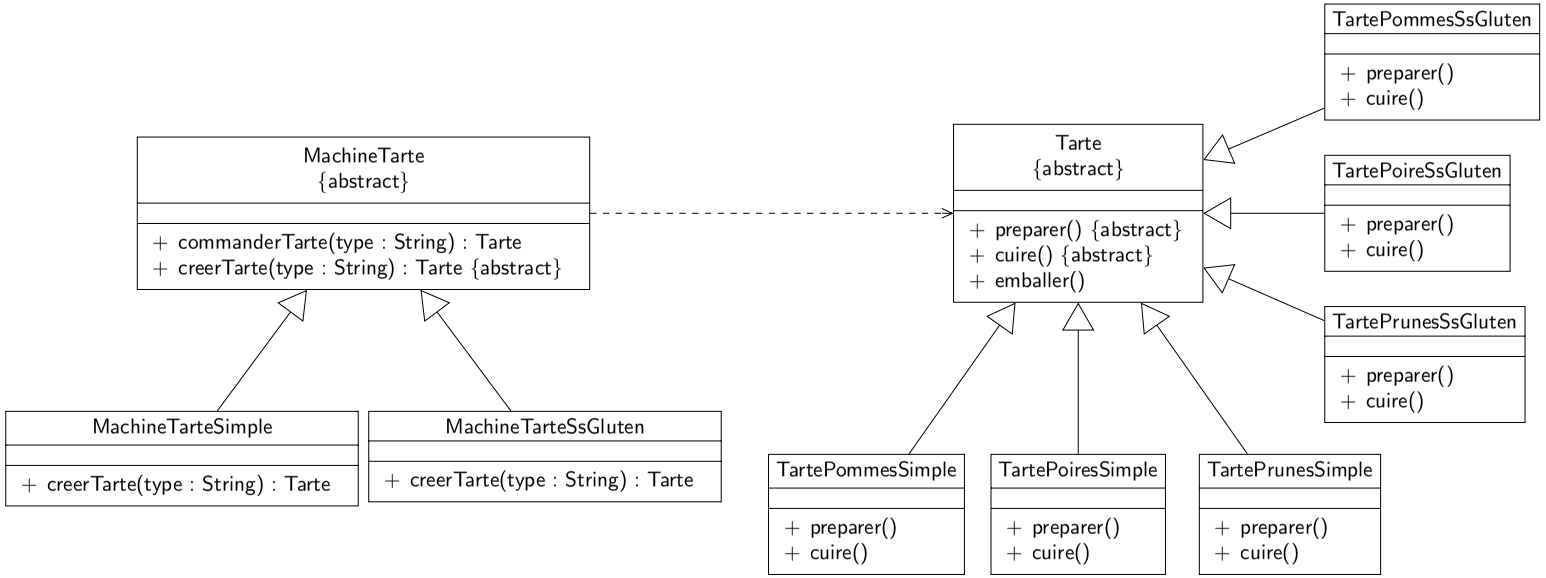

FIG. 4 – *Design pattern factory method* adapté au problème des tartes

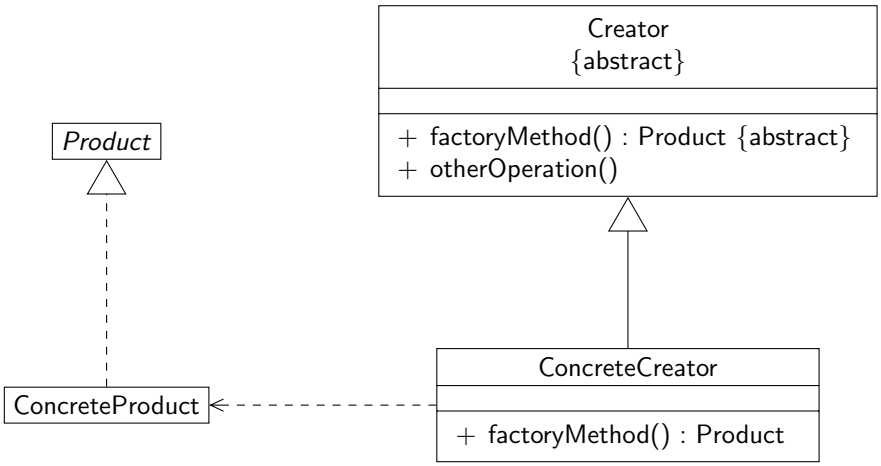

FIG. 5 – Le patron de conception *factory*

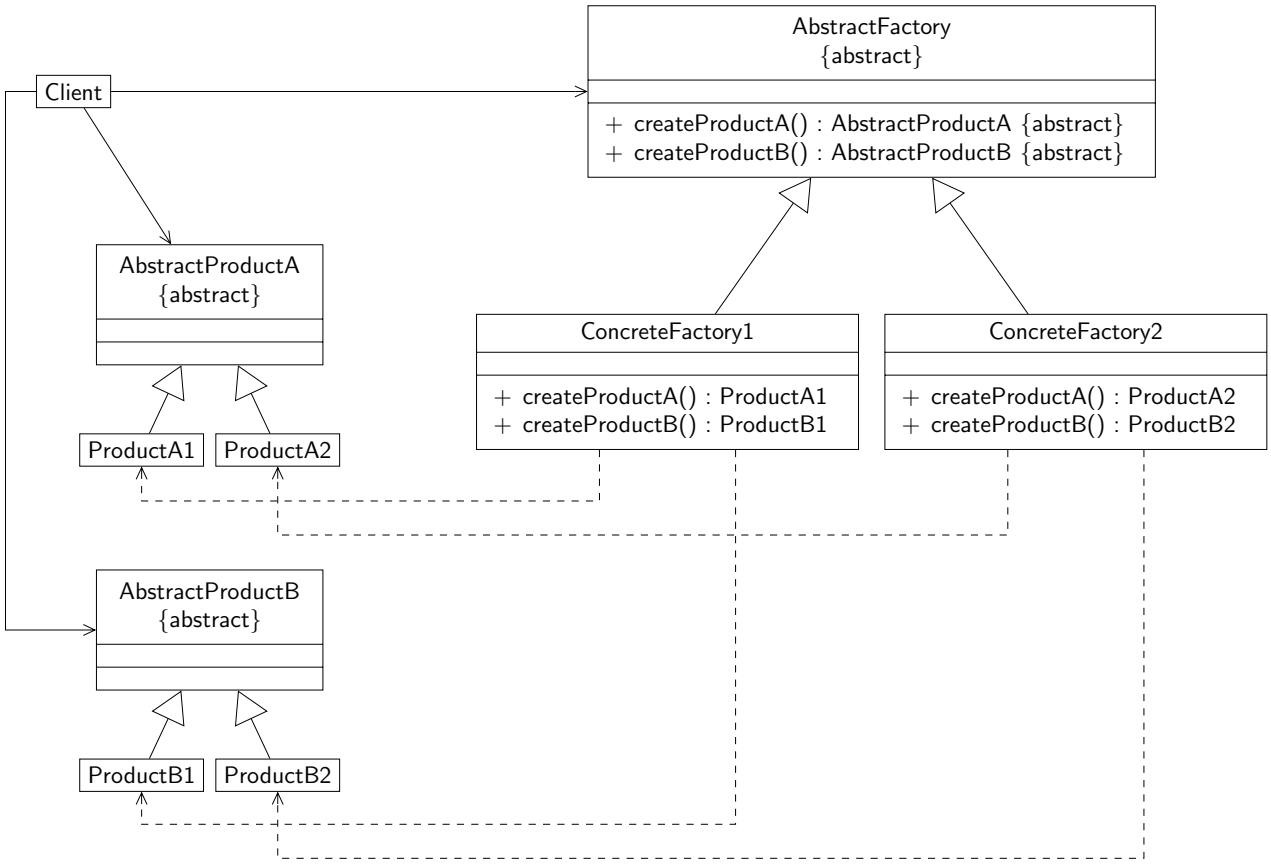

FIG. 6 – Le patron de conception *abstract factory method*

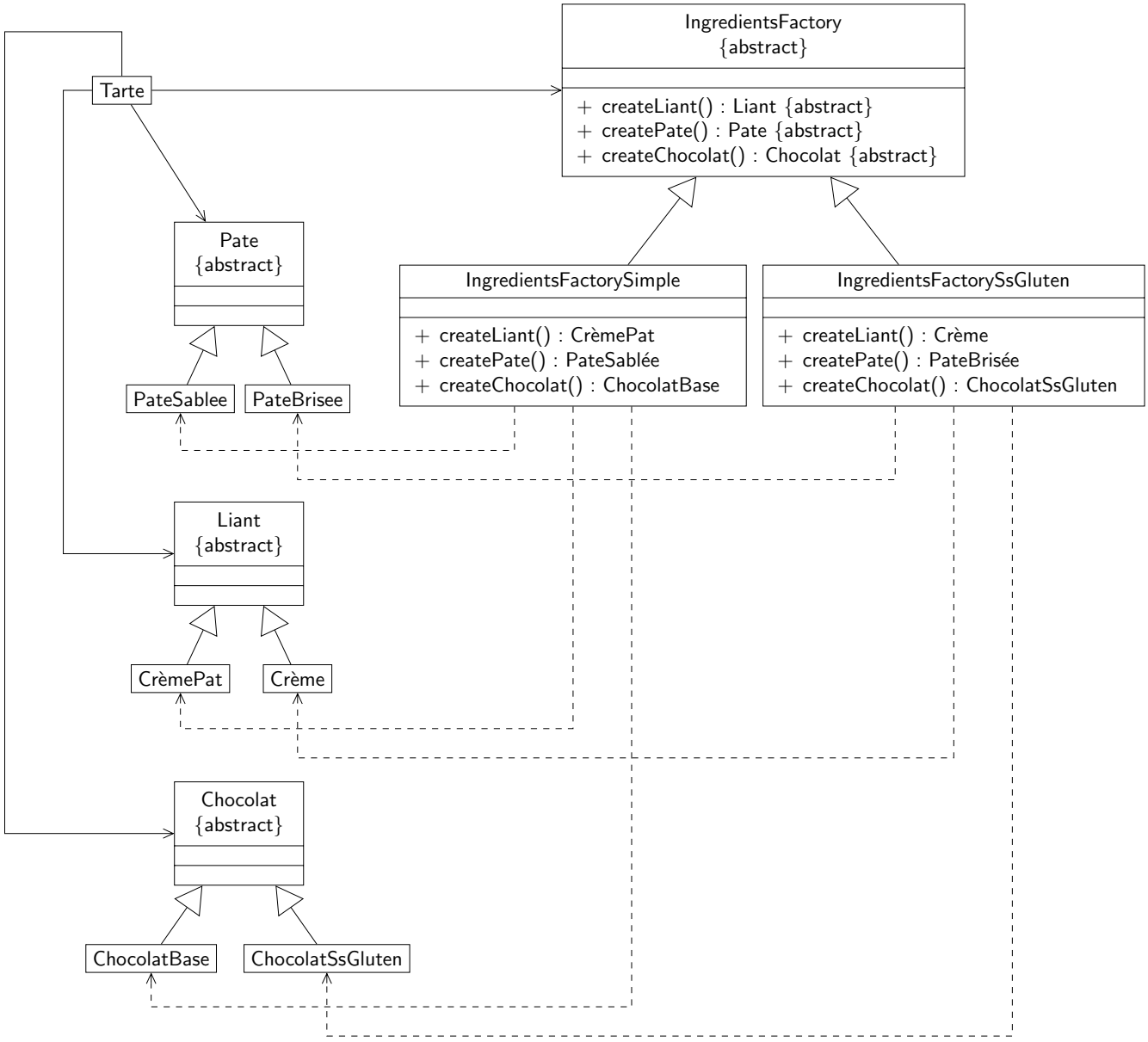

FIG. 7 – Diagramme de classe adaptant le *pattern abstract factory* au problème des tartes

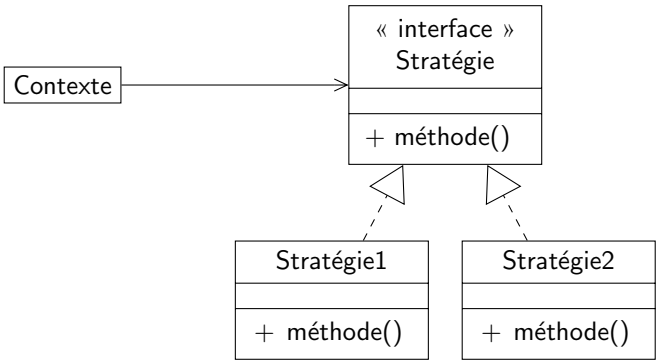

FIG. 8 – Le patron de conception Stratégie

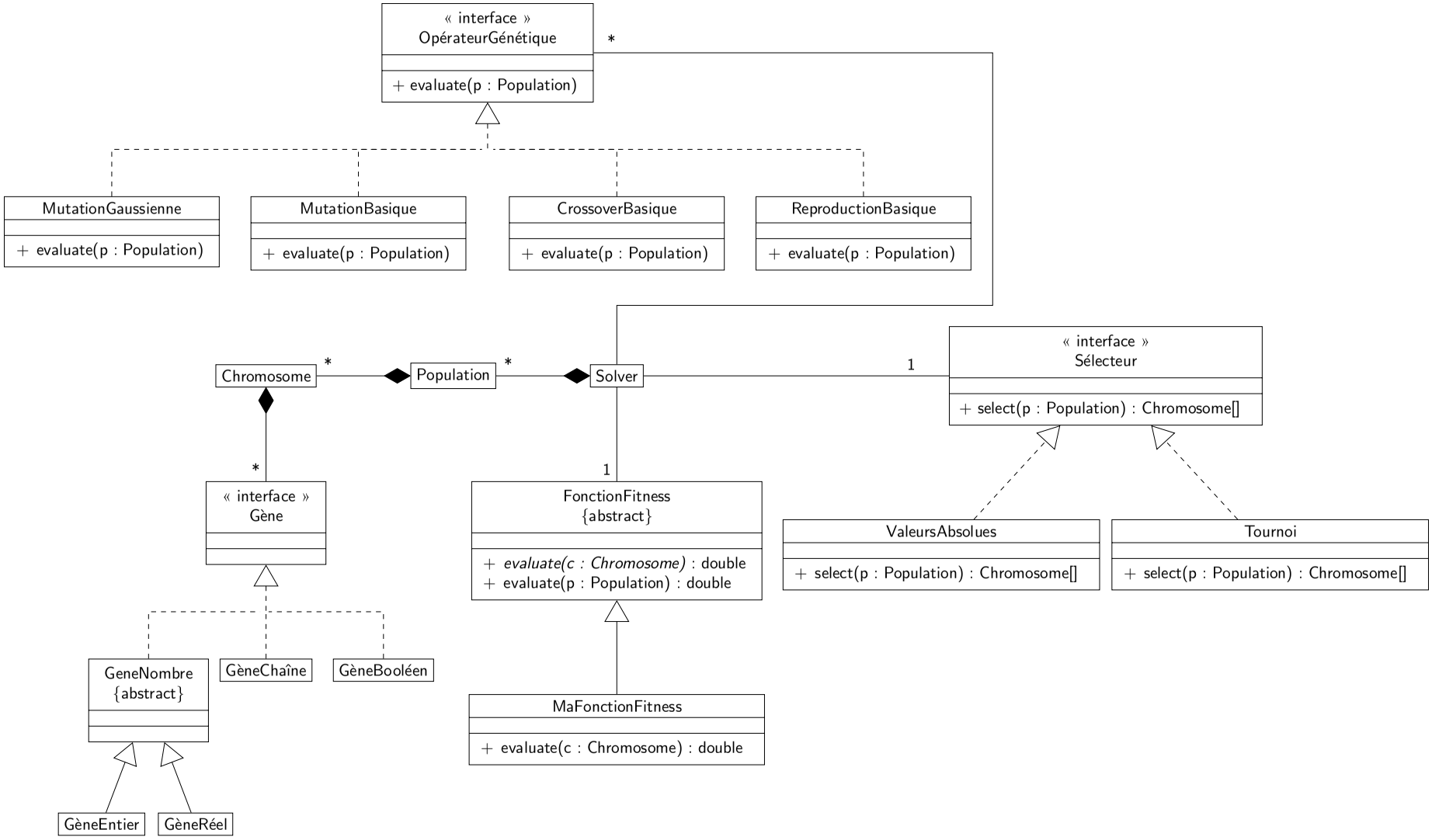

FIG. 9 – Diagramme d'analyse du *solver*

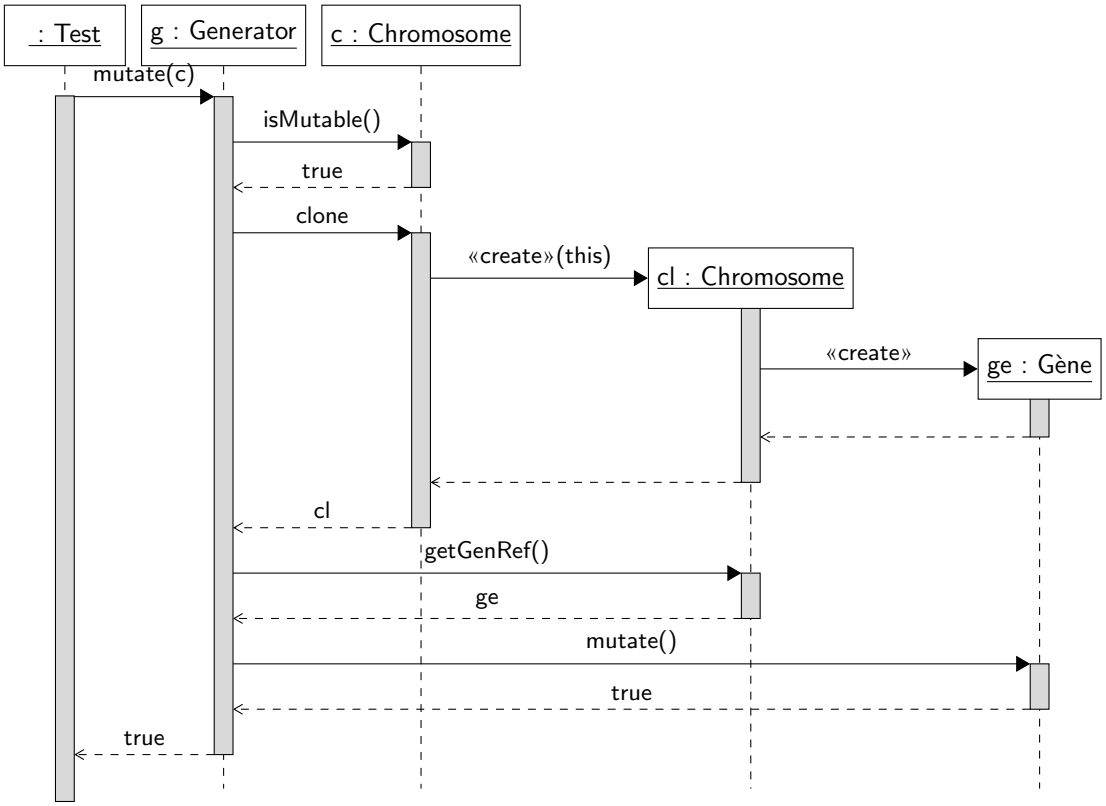

FIG. 10 – Diagramme de séquence représentant la mutation d'un gène

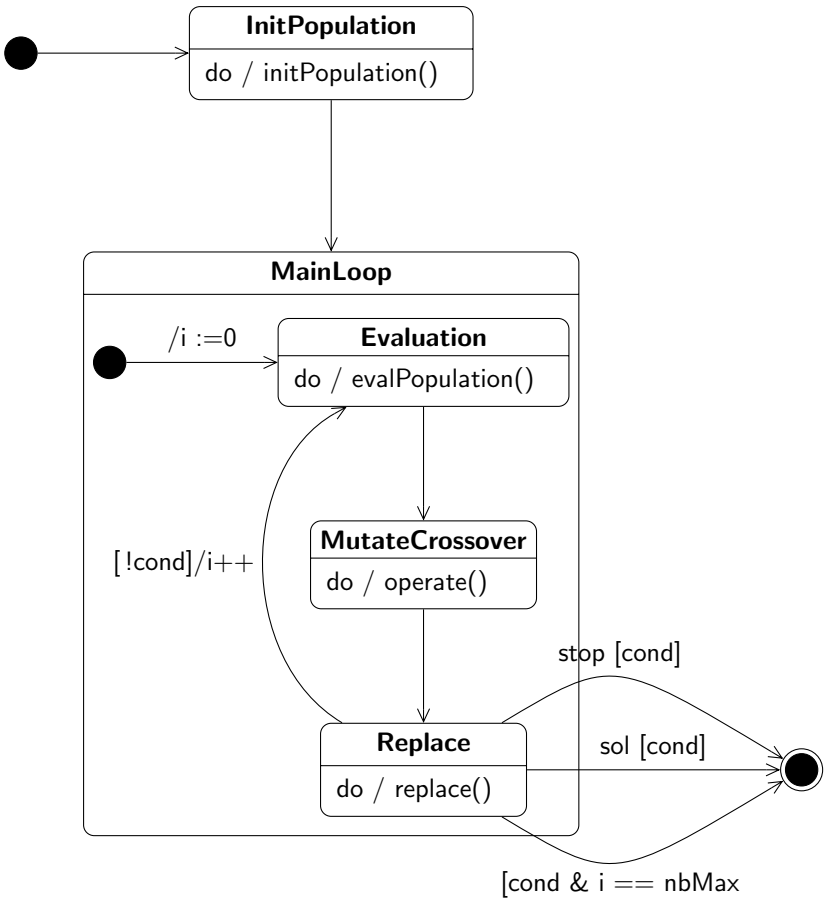

FIG. 11 – Diagramme de machine d'états représentant le fonctionnement du *solver*