IN201 : Corrigé de l'examen

SUPAERO 2A

Christophe Garion <garion@supaero.fr> 3 mars 2008

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.

Remarque importante : dans tous les exercices, il sera tenu compte de la syntaxe UML lors de l'écriture de diagrammes.

1 Modélisation du système immunitaire humain

Cet exercice a été élaboré en utilisant les références [3, 1]. Il ne prétend pas être une référence en médecine ou en biologie cellulaire. Certaines approximations ont été faites pour simplifier le sujet.

On cherche à fournir à un centre de recherche en biologie un logiciel de simulation de réaction du système immunitaire humain à différents virus et bactéries. Dans un premier temps, on cherche donc à proposer un modèle objet des notions « métier » nécessaires aux biologistes pour modéliser le système immunitaire.

Le système immunitaire est une collection de mécanismes permettant à un organisme de se protéger de l'infection en détectant et en tuant des agents pathogènes. Il faut donc qu'il puisse distinguer les molécules du corps des molécules étrangères. Les agents pathogènes peuvent être des bactéries (comme l'anthrax), des virus (comme l'herpès), des protozoaires (comme la malaria), des champignons (comme le candidiasis), des parasites (comme le ver solitaire) ou des protéines (comme les prions). Les différents pathogènes ont pour effet d'attaquer ou de modifier les cellules du corps.

Le système immunitaire d'un être humain est constitué d'un ensemble de protéines, de cellules, d'organes et de tissus. Il est décomposé en plusieurs couches : une couche physique, le système inné et le système adaptatif. La couche physique empêche certains pathogènes, comme les virus et les bactéries d'entrer dans le corps. Si un pathogène réussit à rentrer, le système immunitaire inné déclenche une première réponse non spécifique. Si celle-ci n'est pas efficace, le système immunitaire adaptatif permet d'adapter la réponse du corps au pathogène et de se souvenir de cette réponse appropriée via la mémoire immunologique.

La couche physique comporte par exemple la peau, les poumons, les larmes produites par les yeux, la flore intestinale ou des barrières chimiques, par exemple contenues dans la salive (enzymes) ou produites par la peau (peptides).

Le système immunitaire inné entre en action lorsqu'un organisme étranger entre dans le corps. Il propose une réponse générique au pathogène sous plusieurs formes : inflammation, système du complément (ensemble d'une vingtaine de protéines permettant de détruire la membrane des cellules étrangères), leucocytes (globules blancs). Ces derniers ont plusieurs formes, la plus connue étant les phagocytes, tous capables de phagocyter un pathogène. On trouve parmi eux les macrophages, les neutrophiles et les cellules dendritiques.

Le système immunitaire adaptatif fournit lui une réponse spécialisée par type de pathogène. Ses principales fonctions sont : la reconnaissance des antigènes n'appartenant pas au corps, la génération d'une réponse adaptée et le développement d'une mémoire immunologique. Il utilise un type particulier de leucocytes, les lymphocytes. Les lymphocytes peuvent reconnaître des cibles pathogènes particulières. Ils se divisent en deux catégories : les lymphocytes T, et les lymphocytes B, qui peuvent créer des plasmocytes fabriquant des anticorps.

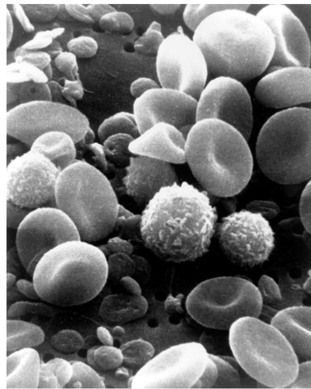


FIG. 1 – Une photo prise au MEB d'un échantillon de sang circulant humain. On distingue de nombreux globules rouges sous forme de disques applatis biconcaves, ainsi que de nombreux types de globules blancs : des lymphocytes, un monocyte, un neutrophile, et de nombreuses plaquettes (source NCA, 1982).

1. proposer un diagramme de classes d'analyse représentant le domaine. 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.

   Vous trouverez sur la figure 2 une proposition de corrigé. Ce n'est évidemment pas la seule solution possible.

   Quelques remarques sur ce diagramme de classes :

   - j'ai utilisé une interface, `AgentPathogene`, pour représenter les agents pathogènes. Cette interface propose deux services : un pour attaquer une cellule et un pour détruire une cellule. On aurait pu se demander s'il n'aurait pas mieux valu placer ces méthodes dans une classe `Cellule`. Je pense qu'elles sont mieux ici, car la façon d'attaquer ou de détruire une cellule va dépendre de l'agent pathogène en question.
   - on trouve sous `AgentPathogene` un ensemble de classes abstraites représentant les grands types de pathogènes : virus, bactéries etc. J'ai choisi des classes abstraites, car je pense que l'on peut factoriser soit du comportement, soit des caractéristiques pour chacune de ces familles. Pour chaque famille, une classe concrète propose un exemple.
   - la hiérarchie « `AgentPathogene` » n'est pas reliée directement aux autres classes, mais sera utilisée dans des paramètres ou des retours de méthodes.
   - le système immunitaire est lui représenté par une classe `SystemeImmunitaire`. Je n'ai pas représenté les liens donnés dans l'énoncé vers tissus, protéines, cellules etc. La caractéristique principale du système immunitaire est d'être composé de trois grands mécanismes. J'ai donc choisi une relation de composition vers les classes `CouchePhysique`, `SystemeInne` et `SystemeAdaptatif`.
   - les éléments de la couche physique étaient assez simples à modéliser. J'ai choisi une classe abstraite pour `BarriereChimique`, que j'ai sous-classée ensuite avec `Enzyme` et `Peptide`.
   - pour le système inné, rien de bien particulier, mis à part le fait que l'on voyait apparaître la hiérarchie des leucocytes. `Leucocyte` est une classe abstraite (on suppose que l'on peut regrouper des caractéristiques à tous les leucocytes), sous-classée ensuite avec `Phagocyte`. J'ai fait apparaître explicitement une méthode abstraite dans `Phagocyte`, `phagocyter`, qui représente l'action principale de ces cellules.
   - j'ai placé une contrainte sur la liaison entre `SystemInne` et `Leucocyte`, car je suppose que les

lymphocytes, qui sont aussi des leucocytes, n'appartiennent pas au système inné, mais adaptatif.
– pour le système adaptatif, j'ai ajouté quelques méthodes tirées de l'énoncé.

2. représenter par un diagramme de séquence le scénario suivant :
   (a) le virus de la grippe entre dans un corps humain ;
   (b) le corps réagit dans un premier temps en produisant une inflammation ;
   (c) un lymphocyte B a le bon récepteur antigène pour le virus. Il inactive le virus ;
   (d) le lymphocyte phagocyte alors le virus ;
   (e) le lymphocyte produit alors un complexe peptidique ;
   (f) ce complexe est reconnu par un lymphocyte T qui stimule le lymphocyte B par l'envoi de cytokines ;
   (g) le lymphocyte B se réplique alors en plasmocytes qui secrètent des anticorps permettant de neutraliser le virus.

   On supposera que l'on dispose d'une instance de la classe `SystemeImmunitaire`. On réfléchira en particulier aux appels de méthodes : dans quel sens se font-ils ? Avec quels arguments ?

   Une proposition de corrigé est présentée sur le diagramme 3. Rien de bien particulier sur ce diagramme. On peut juste remarquer que j'ai considéré que lorsque le virus était phagocyté, il était ensuite détruit. De la même façon, lorsque le lymphocyte B se réplique en plasmocytes, l'instance de `LymphocyteB` correspondante est détruite.
   J'ai choisi de considérer qu'il y avait une méthode **`getCaracteristiques`** dans **`ComplexePeptidique`** dont le retour permettait au lymphocyte T de déclencher la stimulation du lymphocite B. On aurait également pu considérer que ce soit l'instance de **`ComplexePeptidique`** qui appelle une méthode **`reconnaitre`** de **`LymphocyteT`**, car l'appel à l'une ou l'autre des méthodes se fait lors du contact entre les deux entités.

3. on s'intéresse maintenant au cycle de vie d'un lymphocyte B. Un lymphocyte « naît » dans la moelle osseuse. Il est alors en mode « naïf » et passe dans le système lymphatique. Lorsqu'il rencontre un de ses antigènes de prédilection, il engloutit l'antigène et le digère. Il affiche ensuite des fragments de l'antigène. La combinaison de l'antigène attire alors un lymphocyte auxiliaire T mature. Ce dernier sécrète des cytokines qui enclenchent un processus de division du lymphocyte B. Celle-ci produit alors des plasmocytes qui peuvent produire des anticorps. Les lymphocytes meurent au bout de 2-3 jours, sauf environ 10% des cellules qui deviennent des lymphocytes B à mémoire pour pouvoir réagir plus rapidement à la prochaine infection.
   Proposer un diagramme de machine d'états modélisant le cycle de vie d'un lymphocyte B.

   Un diagramme de machine d'états représentant le cycle de vie du lymphocyte est proposé sur la figure 4. Rien de bien particulier pour cette question, j'ai utilisé des activités internes pour représenter les activités de la cellule prenant du temps.

2 Étude du *design pattern* Adaptateur

Lors du cours, lorsque nous voulions manipuler un ensemble d'objets, nous avons fait appel à des *collections*. En Java, Les collections sont un ensemble de classes appartenant au paquetage `java.util` et représentant différentes façons de stocker un ensemble d'objets : listes, tas, arbres etc. Nous considérerons dans toute la suite que toutes les classes et interfaces appartiennent à `java.util`. Il faudra en tenir compte lors de l'écriture du code.

Pour pouvoir accéder au contenu d'une collection, nous avons utilisé des itérateurs : un itérateur est un objet permettant d'accéder de façon uniforme aux objets d'une collection, sans s'occuper de la façon

dont ces objets sont stockés. Un itérateur est un objet implantant l'interface `java.util.Iterator`. Cette interface possède trois méthodes :
- `hasNext()` qui renvoie un booléen. Elle renvoie **`true`** si la collection possède encore des éléments non parcourus ;
- `next()` qui renvoie le prochain élément de la collection sous la forme d'une instance d'`Object` et qui avance d'un élément dans la collection ;
- `remove()` qui enlève l'élément courant.

1. représenter l'interface `Iterator` sur un diagramme UML ;

   Le diagramme est proposé sur la figure 5.

2. écrire en Java une méthode statique `affiche` qui prend un itérateur en paramètre et affiche toutes les chaînes de caractères se trouvant dans la collection correspondante ;

   Voici la méthode demandée. Rien de bien particulier ici, il fallait juste faire attention à utiliser **`instanceof`** pour trouver les instances de `String` dans la collection.

```
/**
 * <code>affiche</code> permet d'afficher toutes les
 * chaines contenues dans une collection.
 *
 * @param i est un iterateur (instance de
 *          <code>java.util.Iterator</code>) permettant de
 *          parcourir la collection
 */
public static void affiche(java.util.Iterator i) {
    Object o = null;

    while (i.hasNext()) {
        if ((o = i.next()) instanceof String) {
            System.out.println(o);
        }
    }
}
```

3. les anciennes classes représentant les collections en Java (par exemple `Vector` ou `Stack`) implantent une méthode `elements` qui renvoie un objet de type `Enumeration`. L'interface `Enumeration` permet également de parcourir ces collections, mais ne permet pas d'effacer un élément de la collection. Ses méthodes sont :
   - `hasMoreElements()` qui renvoie **`true`** si il y a encore des éléments à parcourir dans la collection ;
   - `nextElement()` qui renvoie le prochain élément sous la forme d'une instance d'`Object`.

   Représenter l'interface sur le diagramme UML précédent.

   L'interface est représentée sur la figure 6.

4. on dispose d'un objet de type `Enumeration` sur une instance de `Vector` par exemple et on souhaiterait pouvoir utiliser dessus la méthode `affiche` développée précédemment sans modifier les interfaces `Enumeration` et `Iterator`, ni la méthode `affiche(Iterator)` développée précédemment. Est-ce possible ? Pourquoi ? Quel « mécanisme » intervient ?

   Ce n'est évidemment pas possible. Supposons que nous ayons un objet `e` de type `Enumeration`, si l'on écrit dans une méthode l'appel `affiche(e)`, comme Java est un langage typé statiquement et que `Enumeration` n'est pas un sous type d'`Iterator`, le compilateur va refuser l'appel.

5. pour résoudre ce problème, nous allons utiliser un *design pattern*[2] : l'adaptateur. Il s'agit de réaliser l'interface `Iterator` par une classe `EnumerationIterator` (qui possédera donc toutes les méthodes de l'interface). `EnumerationIterator` utilisera l'instance de `Enumeration` que nous voulions utiliser : les méthodes de `EnumerationIterator` utiliseront celles de `Enumeration`.

   Compléter le diagramme UML avec la classe `EnumerationIterator`.

   Le diagramme est représenté sur la figure 7. Rien de bien particulier, la classe `EnumerationIterator` réalise l'interface `Iterator`, et délègue le traitement des méthodes à `Enumeration` (sens de l'association). Il ne fallait pas oublier le constructeur de la classe.

6. écrire la classe `EnumerationIterator` en Java. On constate que la méthode `remove` n'existe pas dans l'interface `Enumeration`. Indiquer comment écrire la méthode `remove` de `EnumerationIterator`.

   La seule difficulté était l'écriture de la méthode `remove`, `hasNext` et `next` se limitant en effet à déléguer les traitements nécessaires à `Enumeration`. On ne peut pas écrire `remove` en utilisant `Enumeration`, car on ne dispose pas d'une méthode similaire. Il faudrait donc lever une exception lors de l'appel de `remove` sr un objet de type `EnumerationIterator`. Comme `EnumerationIterator` réalise `Iterator`, elle doit respecter son contrat. Or `remove` dans `Iterator` ne lève pas d'exception. On est donc obligé en Java d'utiliser une exception hors contrôle pour ne pas à avoir à la spécifier. J'ai choisi ici `UnsupportedOperationException` qui hérite de `RuntimeException`.
   Le code de `EnumerationIterator` est le suivant :

```java
import java.util.Enumeration;
import java.util.Iterator;

/**
 * <code>EnumerationIterator</code> est une classe permettant d'utiliser
 * une enumeration comme un iterateur. Attention cependant, la methode
 * <code>remove</code> levera toujours une exception.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class EnumerationIterator implements Iterator {

    private Enumeration en;

    /**
     * Creer une instance de <code>EnumerationIterator</code>.
     *
     * @param en une <code>Enumeration</code> que l'on veut adapter
     *           a un <code>Iterator</code>
     */
    public EnumerationIterator(Enumeration en) {
        this.en = en;
    }

    // Implementation of java.util.Iterator

    public final boolean hasNext() {
        return this.en.hasMoreElements();
```

```java
    }

    public final Object next() {
        return this.en.nextElement();
    }

    /**
     * Ne doit pas etre appelee ! Leve obligatoirement une
     * UnsupportedOperationException !
     *
     * @throws UnsupportedOperationException
     */
    public final void remove() {
        throw new UnsupportedOperationException();
    }
}
```

7. proposer un diagramme de classe général pour le *design pattern*. L'objectif général de ce *pattern* est d'adapter une interface existante pour l'utiliser dans un nouveau contexte, c'est-à-dire avec une nouvelle interface.

   Un diagramme de classes représentant le *design pattern* est proposé sur la figure 8. Rien de bien particulier, il fallait bien comprendre que la classe `Adaptateur` devait :
   - réaliser l'interface cible pour que le principe de substitution puisse s'appliquer ;
   - déléguer ses appels à l'interface à adapter.

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 :

```java
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

[1] Il vaut mieux bien évidemment les faire soi-même, c'est meilleur...

(par exemple "pommes" ou "poires"). En fonction de cette chaîne de caractères, on construira la tarte correspondante.
Modifier la méthode commanderTarte en conséquence.

La méthode commanderTarte 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 tarte n'est pas reconnu, on aura une exception de type NullPointerException qui sera levée.

```java
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;
}
```

2. 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.

   Le diagramme de séquence est représenté sur la figure 11.

   (c) donner le code Java des classes FabriqueTarte et MachineTarte.

   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 FabriqueTarte dans MachineTarte.

```java
/**
 * <code>FabriqueTarte</code> represente des objets dont le role est de
 * construire des instances de tartes.
```

[2] 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.

```java
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
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
/**
 * <code>MachineTarteSecond</code> est une machine a tartes utilisant
 * une fabrique de tartes.
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
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;
```

```java
    }

    /**
     * <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();

        return tarte;
    }
}
```

3. 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`.

   Que pensez-vous de cette solution ? En particulier, comment créer une machine pour un type de tarte particulier ? Contrôle-t-on l'utilisation qui sera faite des fabriques de tarte ?

   Avec cette solution, on va créer une machine pour les tartes sans gluten par exemple en passant en paramètre du constructeur de `MachineTarte` une instance de `FabriqueTarteSansGluten`. On n'a donc aucun contrôle sur le type réel de fabrique que l'on utilise dans une machine. De plus, les fabriques peuvent être utilisées en dehors des machines sans aucun problème.

4. pour pallier les problèmes soulevés précédemment, nous décidons de placer la méthode permettant de créer les tartes non pas dans une classe extérieure, mais dans les classes `MachineTarte`, `MachineTarteSimple` et `MachineTarteSansGluten`.

   Nous décidons donc d'utiliser le *design pattern*[2] Factory Method. Celui-ci est représenté sur la figure 12.

   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`.

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

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*.

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

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 `dactoryMethod` dans `ConcreteCreator`.

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

Une adaptation simple est proposée sur la figure 13. Rien de bien particulier ici.

(d) écrire les classes `MachineTarte` et `MachineTarteSimple`.

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

```java
/**
 * <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
```

```java
     */
    public Tarte commanderTarte(String type) {
        Tarte tarte = creerTarte(type);

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

        return tarte;
    }
}
```

```java
/**
 * <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;
    }
}
```

Références

[1] Microbiology Department of Pathology and School of Medicine Immunology at the University of South Carolina. Microbiology and immunology on-line. http://pathmicro.med.sc.edu/book/welcome.htm.

[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] Wikipedia. Immune system. http://en.wikipedia.org/wiki/Immune_System.

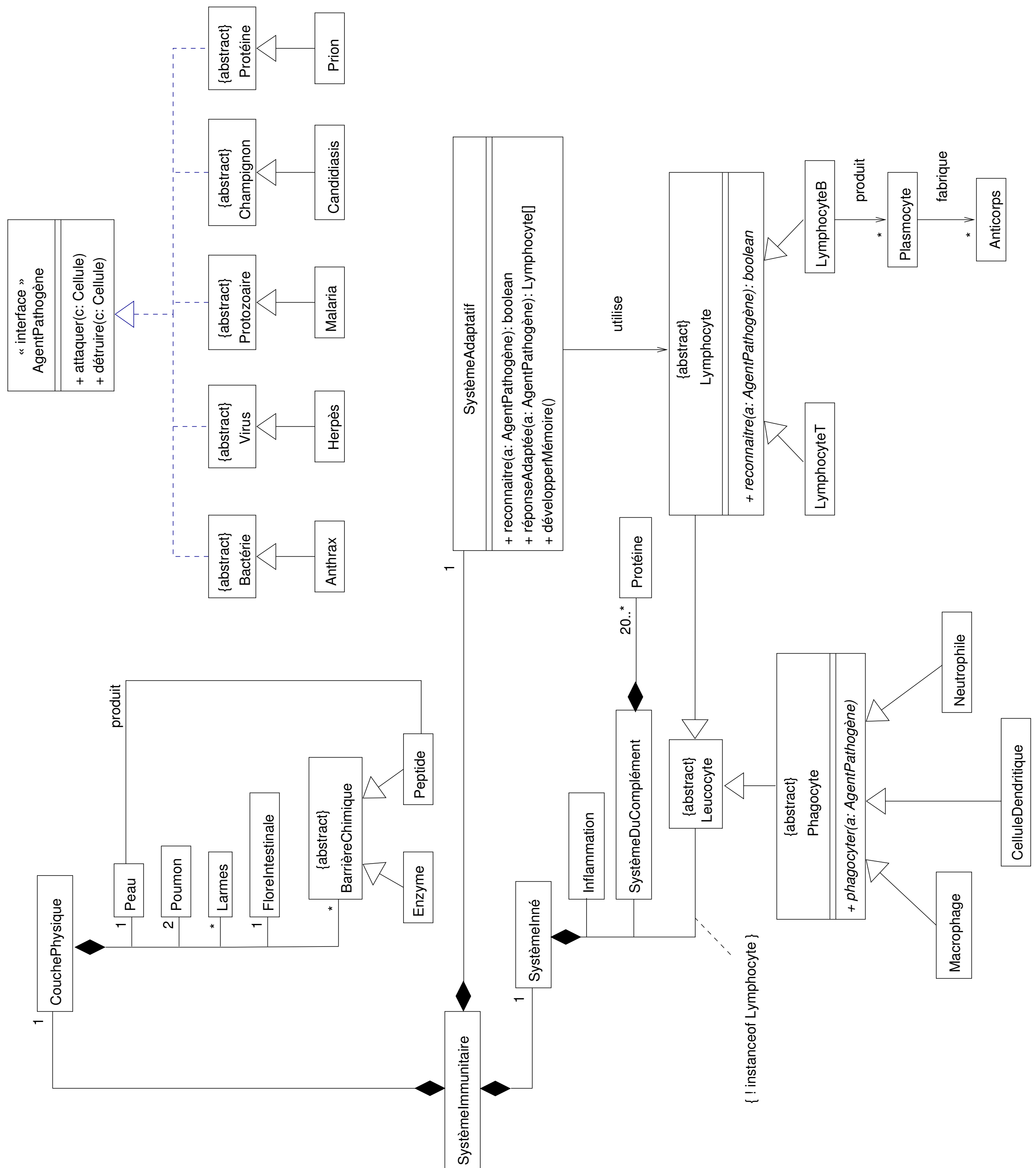


FIG. 2 – Diagramme de classes représentant le domaine du problème

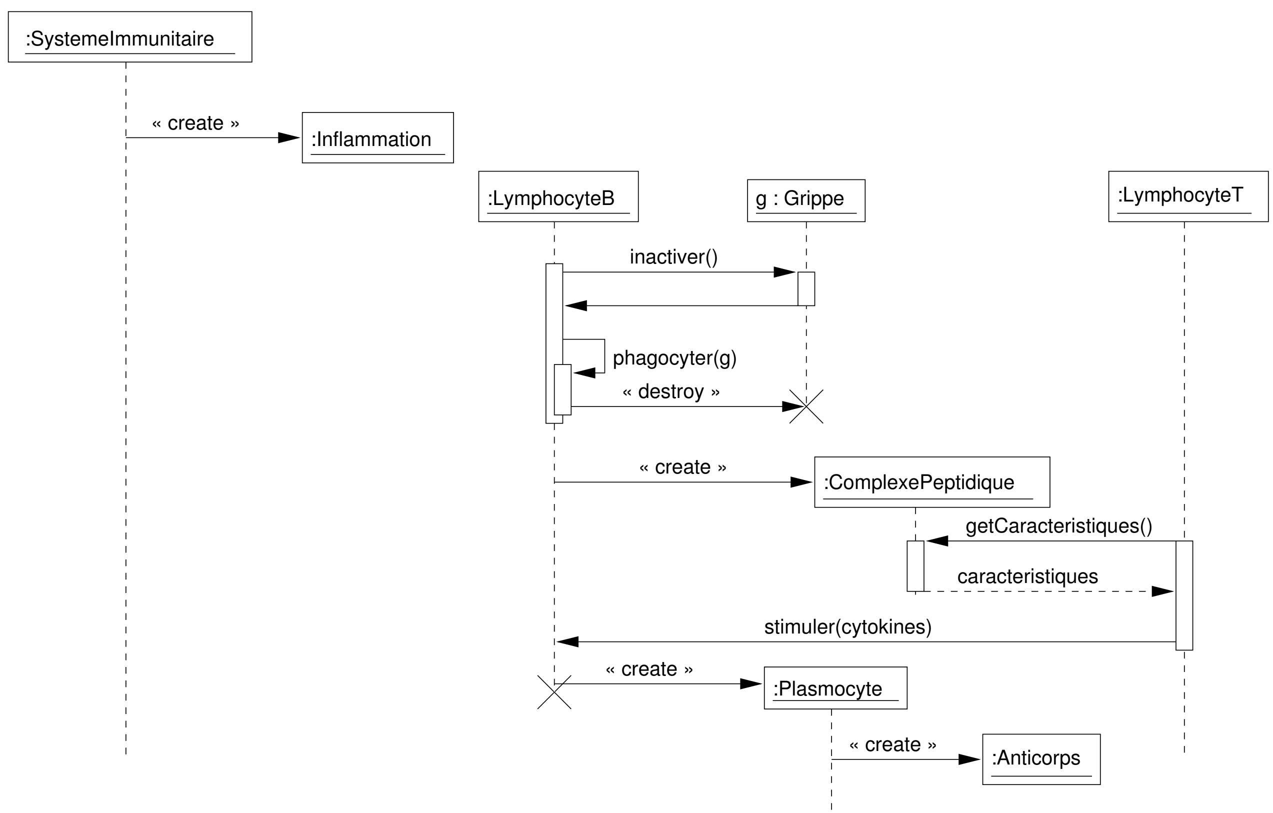


Fig. 3 – Diagramme de séquence représentant la destruction du virus de la grippe par des lymphocytes B

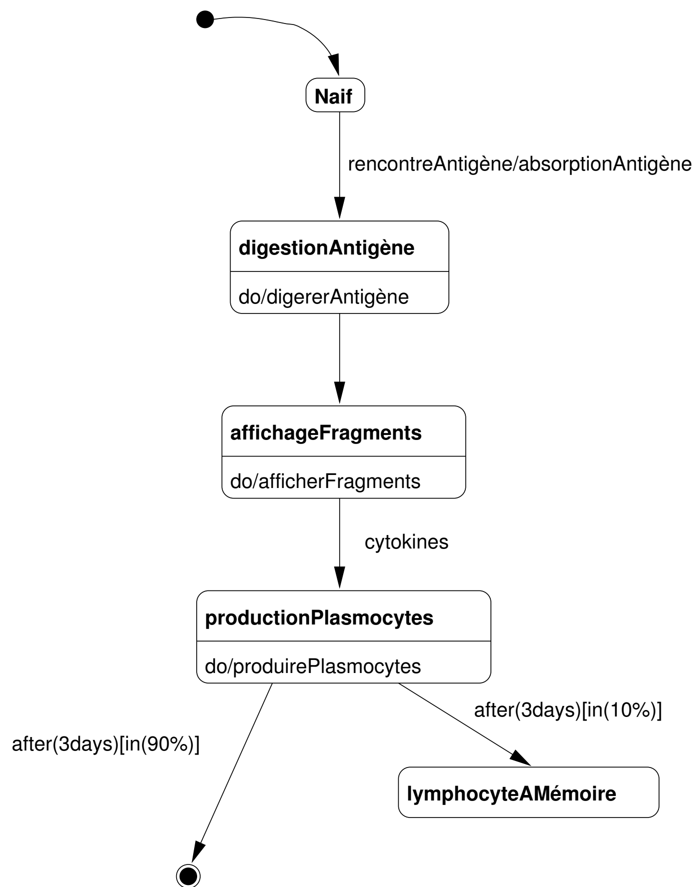


Fig. 4 – Diagramme de machine d'états représentant le cycle de vie d'un lymphocyte B

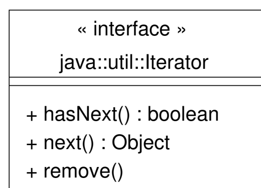
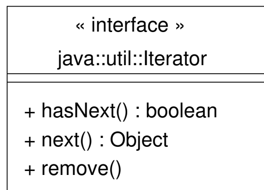


Fig. 5 – Diagramme de classe représentant l'interface `Iterator`



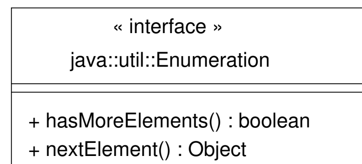


Fig. 6 – Diagramme de classes présentant l'interface `Enumeration`

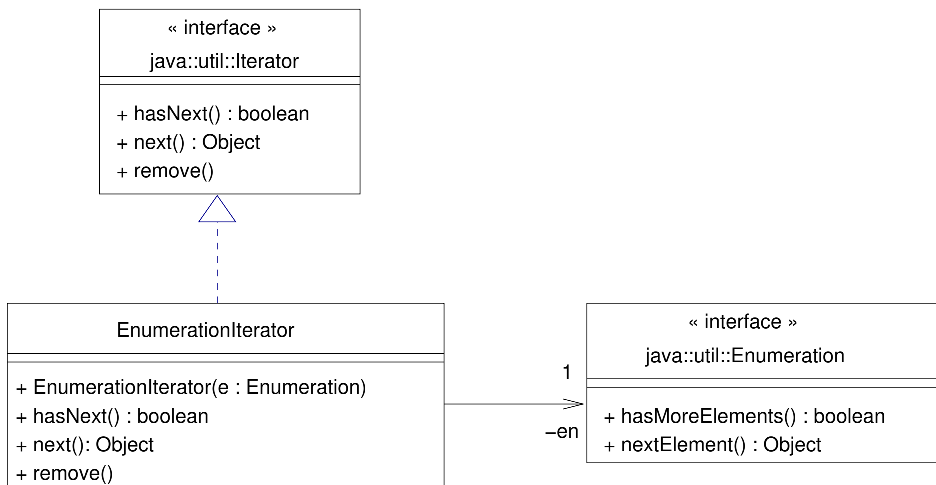


Fig. 7 – Diagramme de classe incluant la classe `EnumerationIterator`

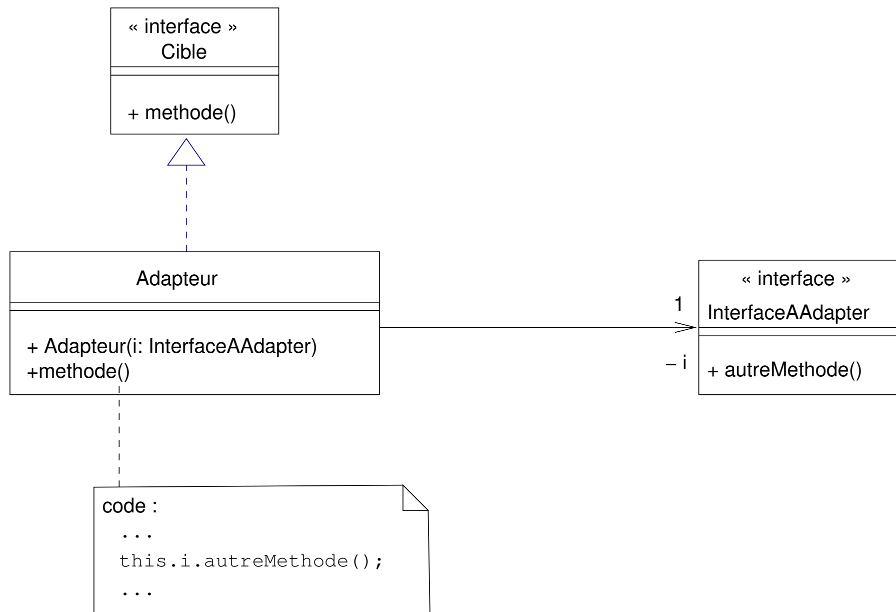


FIG. 8 – Diagramme de classes représentant le *design pattern* Adaptateur

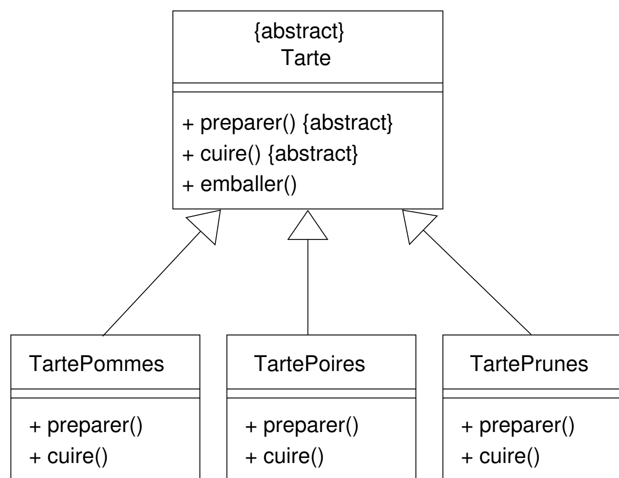


FIG. 9 – Hiérarchie de classes représentant les tartes

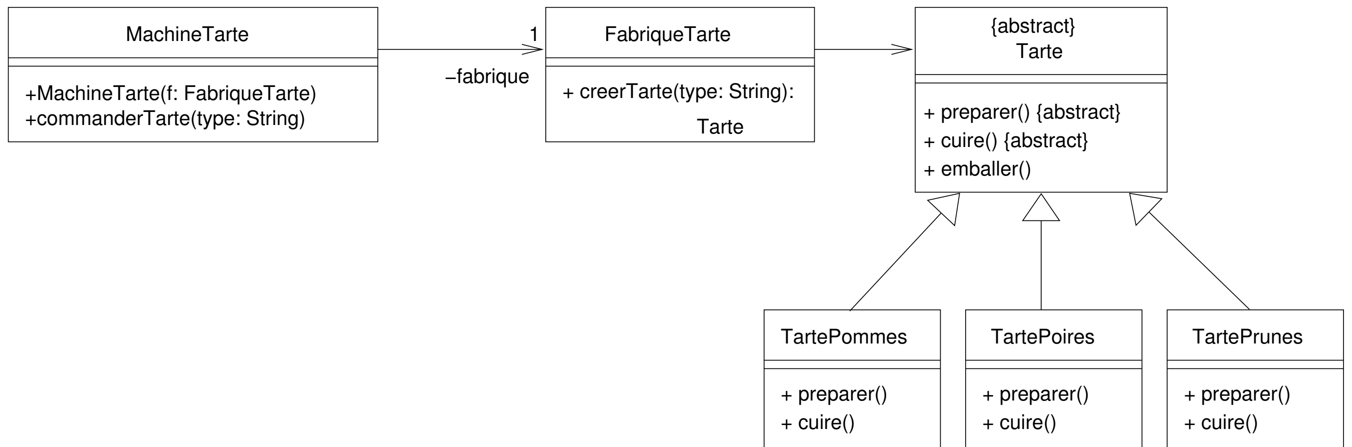


Fig. 10 – Diagramme de classes représentant `FabriqueTarte`

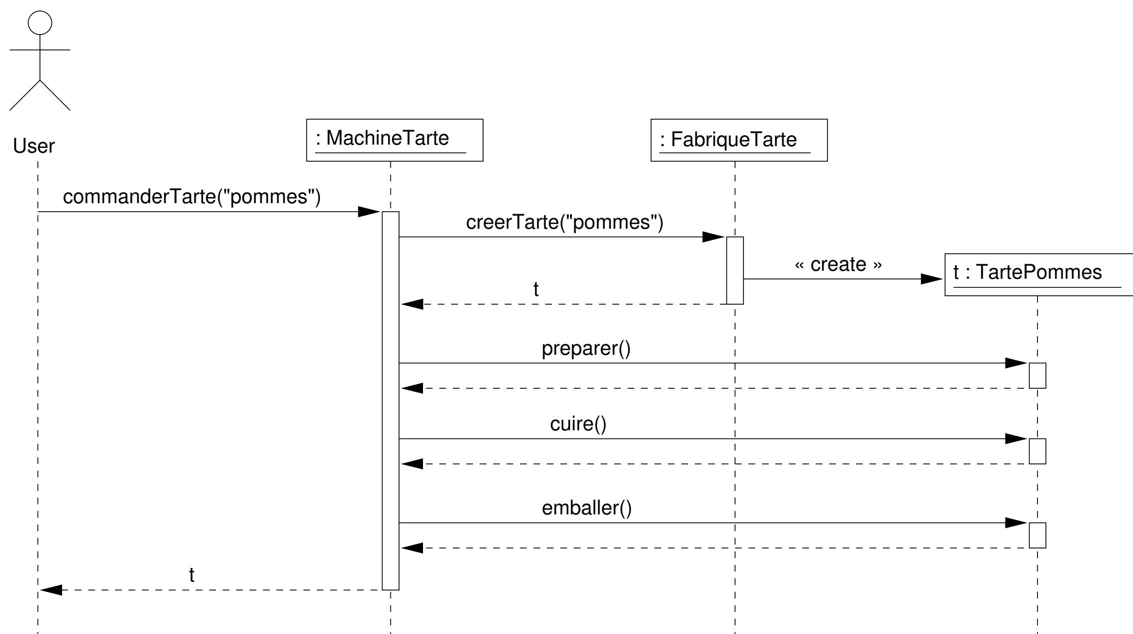


Fig. 11 – Diagramme de séquence représentant l'interaction entre `MachineTarte`, `FabriqueTarte` et `TartePommes`

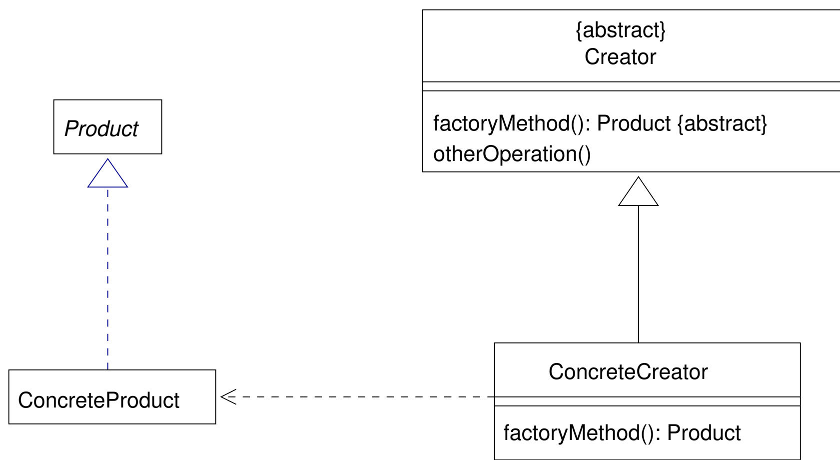


FIG. 12 – Le *Design Pattern* Factory Method

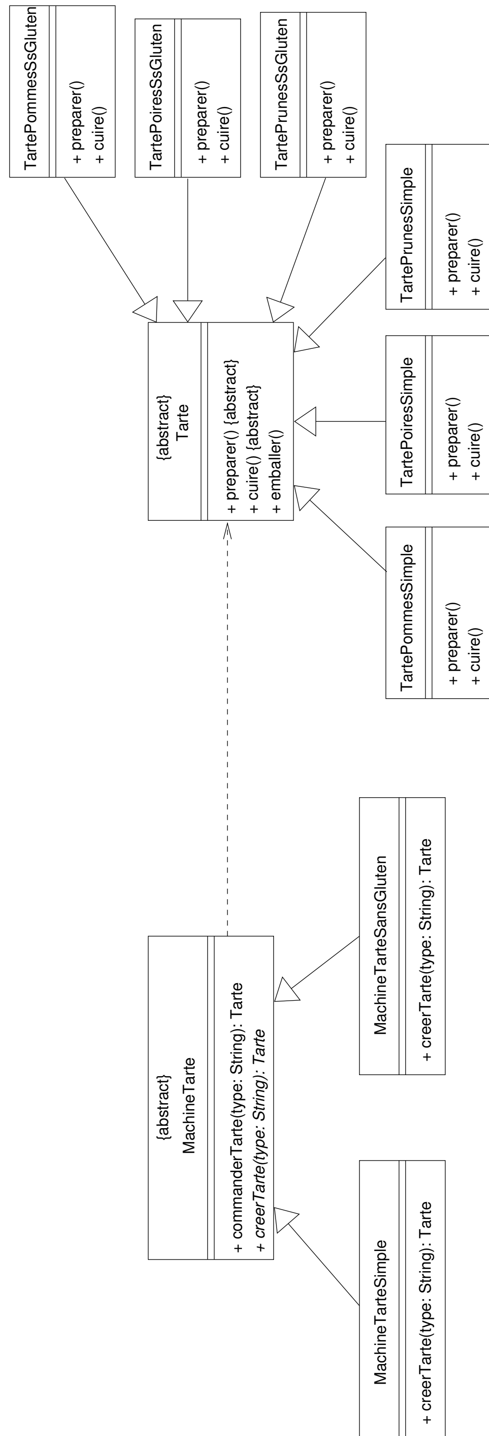


Fig. 13 – *Design pattern factory method* adapté au problème des tartes