Exercice 1.1 Hello World & Compilation

Pour vérifier que votre compilateur fonctionne.

1. Taper et compiler le fichier `HelloWorld.java` suivant.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

2. Exécuter le fichier `HelloWorld.class`.

Exercice 1.2 Structure d'une Classe

Pour réaliser un CV textuel avec très peu d'informations.

1. Taper et compiler le fichier `CV.java` suivant en faisant attention à la casse.

```
public class CV {
  String nom;
  String prenom;
  // Constructeur
  public CV (String leNom, String lePrenom){
    nom=leNom;
    prenom=lePrenom;
  }
  // Méthode principale
  public static void main(String[] args) {
    CV monCV = new CV (args[0],args[1]);
    CV sonCV = new CV (sonNom,sonPrenom);
    monCV.afficher();
  }
  // Méthode qui affiche le CV
  void afficher(){
    System.out.println("Nom : "+nom);
    System.out.println("Prénom : "+prenom);
  }
}
```

2. Qu'est-ce que `String[] args` signifie et que sont `args[0]` et `args[1]` ?
3. Exécuter le fichier `CV.class` avec deux arguments : vos nom et prénom.
4. Si le nombre d'arguments est différent, que se passe-t-il et pourquoi ?
5. Modifier la méthode principale pour que le constructeur de `CV` ne soit appelé que si le nombre d'arguments passé à l'exécution soit bien de deux. Si ce n'est pas le cas, indiquer qu'il faut deux arguments pour exécuter le programme.
6. Instancier un second `CV` avec pour arguments les nom et prénom de votre voisin, puis afficher le. Faire cette construction *en dur*, c'est-à-dire dans le code et non en passant ces informations par arguments du programme principal.
7. Compléter la méthode `afficher` pour améliorer le rendu visuel du pseudo-CV et indiquer vos compétences en informatique.
8. Ajouter votre date de naissance dans les arguments et le constructeur pour qu'elle soit affichée. Il faut utiliser la méthode `parseInt()` de la classe `Integer` qui prend en argument une chaine de caractères.

Exercice 1.3 Structure d'un programme Java

Pour séparer les tâches : programme principal et objets. (*Sauvegarde*)

1. Construire la classe principale dans le fichier `main.java` à partir de la méthode principale du fichier `CV.java` avec la structure suivante.

```java
class main{
  public static void main(String[] args){
    /* programme principal */
  }
}
```

2. Supprimer la méthode principale du fichier `CV.java`.
3. Compiler séparément les deux fichiers `main.java` et `CV.java` et exécuter le fichier `main.class`. Pourquoi avoir fait une séparation en deux fichiers ?
4. Ajouter un constructeur *par défaut* d'une instance de l'objet `CV` qui est appelé si le nombre d'arguments ne correspond pas. Remarquer que deux méthodes peuvent avoir le même nom mais un nombre d'arguments différent ; comment se nomme ce principe ?
5. Changer le *niveau de visibilité* de la méthode `afficher()` de `public` en `private`. Quel problème cela pose-t-il à l'exécution ? Expliquer pourquoi. Qu'est-ce que la notion d'encapsulation ?
6. Ajouter une méthode, sans argument et qui retourne un entier, pour calculer l'age de la personne du CV. Pour cela il faut savoir :
   — que la classe `GregorianCalendar` permet d'être appelée en incluant cette ligne avant la déclaration de la classe `CV`

   ```java
   import java.util.GregorianCalendar;
   ```

   — que le constructeur par défaut de cette classe instancie la date d'aujourd'hui ;
   — que dans un objet `GregorianCalendar`, les différentes valeurs : jour, mois, année ... sont stockées et référencées par un indice : un indice pour chaque champ ;
   — que `YEAR` est un attribut public de type `int` qui est l'indice du champ pour accéder à la variable année ;
   — que la méthode `get(int field)` retourne la valeur du champ pour l'indice passé en argument.
   Accéder à l'année en cours à l'aide de la méthode et de l'attribut décrit ci-dessus. En déduire l'âge de la personne du CV. Quel doit être le *niveau de visibilité* de la méthode ainsi construite ?
   Aide Adresse internet de la documentation des classes `Calendar` et `GregorianCalendar` qui permet de connaître les méthodes disponibles :
   `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Calendar.html`
   `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/GregorianCalendar.html`

Exercice 2.1 Vocabulaire

Soit le fichier `ClassePrincipale.java`

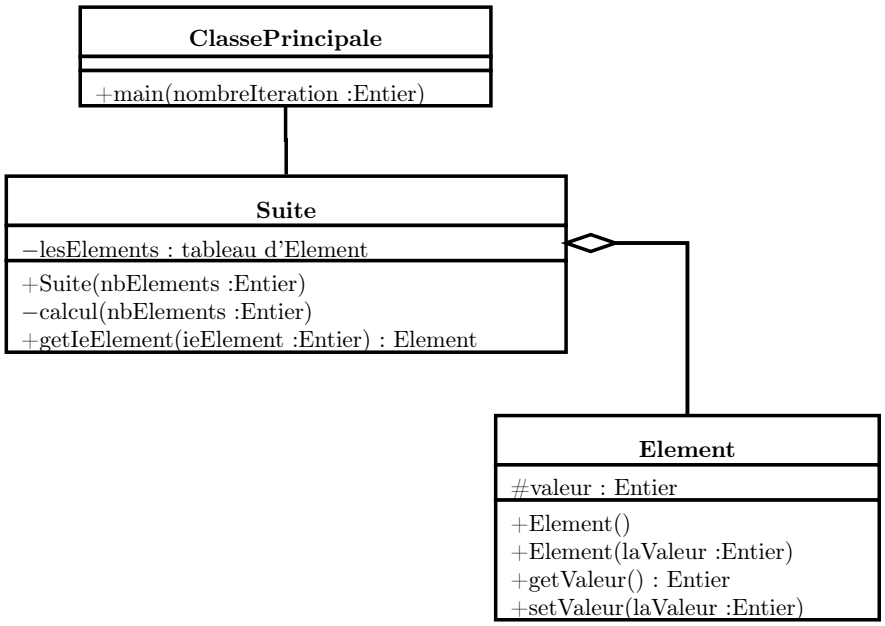
```
class ClassePrincipale{
  public static void main(String[] args){
    int annee,jour,mois;
    CV monCV;
    if(args.length==5){
      jour = Integer.parseInt(args[2]);
      mois = Integer.parseInt(args[3]);
      annee = Integer.parseInt(args[4]);
      monCV = new CV (args[0],args[1],annee,mois,jour);
    }
    else{
      System.out.println("Entrer 5 arguments : Nom Prénom Jour Mois Année");
      monCV = new CV();
    }
    monCV.afficher();
  }
}
```

1. Pour le fichier `ClassePrincipale.java`, indiquer les lignes où il y a :
   (a) l'appel d'une méthode ;
   (b) la déclaration d'un objet ;
   (c) l'appel d'un constructeur ;
   (d) la déclaration d'un ou plusieurs arguments ;
   (e) l'instanciation d'un objet ;
   (f) le début et la fin d'un même bloc d'instructions ;
   (g) la déclaration d'un tableau ;
   (h) l'appel d'un constructeur par défaut.

Exercice 2.2 Suite de Fibonacci, boucles et tableaux

Programmation de la *suite* de Fibonacci définie par la relation $u_{n+2} = u_{n+1} + u_n$ avec $u_0 = 0$ et $u_1 = 1$.

1. Programmer le calcul du $i^{\text{ème}}$ *élément* (u_i) de cette suite à l'aide d'une seule classe ne contenant qu'une seule méthode principale. La valeur de i doit être passée en argument à l'exécution. Afficher à l'écran la valeur de u_{25}.
2. Proposer une représentation objet de ce problème à l'aide du langage UML (diagramme de classes) ;



3. Programmation des différentes classes :
   (a) `ClassePrincipale.java` est la classe principale qui contient le programme principal. Elle prend en argument la valeur du rang pour lequel on veut calculer la suite de Fibonacci. Le programme principal instancie une `Suite` avec pour paramètre la valeur du rang de l'élément
   (b) `Suite.java` est la classe qui va permettre d'instancier l'objet `suiteDeFibonacci` dans le programme principal. Elle est composé d'un tableau de 30 `Element`s que l'on nommera `lesElements`. Voici toutes les méthodes dont il faut implémenter le squelette :
   — `Suite(Entier nbElements)` : le constructeur ;
   — `calcul(Entier nbElements)` : une méthode qui ne retourne rien et qui calcule la suite ;
   — `getIeElement(Entier ieElement)` : une méthode qui retourne le $i^{\text{ème}}$ `Element` de la suite.
   (c) `Element.java` qui a pour membre un entier nommé `valeur`. Voici toutes les méthodes dont il faut implémenter le squelette :
   — `Element()` : le constructeur par défaut ;
   — `Element(Entier laValeur)` : le constructeur ;
   — `getValeur()` : une méthode qui retourne la `valeur` de l'`Element` ;
   — `setValeur(Entier laValeur)` : une méthode qui ne retourne rien mais qui modifie la `valeur` de l'`Element`.
   (d) La méthode principale dialogue avec l'objet instacié pour obtenir l'`Element` voulu. Il faut ensuite dialoguer avec cet objet pour obtenir afficher le résultat dans la *sortie standard*.

   Vérifier les *niveaux de visibilité* des méthodes des classes. Compiler séparément les trois fichiers et exécuter le `ClassePrincipale.class`. Retrouver le résultat de la question 1.
4. Implémenter une méthode de la classe `Suite` qui permet d'afficher tous ses `Element`.
5. *BONUS* Surcharger la méthode précédente pour qu'elle affiche les éléments de i à j si deux entiers sont passés en paramètres.
6. Étant donné que les deux premiers éléments de la suite sont connus, utiliser la structure `switch` pour savoir s'il est nécessaire de faire appel au calcul de la suite de Fibonacci pour retourner la bonne valeur.
7. Les deux programmes donnent les mêmes résultats et sont implémentés en Java. Comment s'appellent ces deux différentes façon de programmer ? Donner le nom qui englobe tous les styles de programmation.
8. Comparer les temps de calculs pour ces deux programmes à l'aide la commande `time` dans le terminal. Il faut le faire pour calculer l'élément à un rang très élevé. Quel sont les deux problèmes à l'exécution ? Proposer et appliquer des corrections.
9. L'utilisation d'un langage objet pour résoudre ce problème est elle judicieuse ? Justifier.
10. Documenter votre code Java.

Exercice 3.1 Interface Graphique

1. Soit les fichiers `Formulaire.java` et `ClassePrincipale.java` à recopier. Ajouter le fichier `CV.java` au répertoire. Compiler et exécuter.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Formulaire extends JFrame {

  private JPanel fond;
  private JTextField nomRecup = new JTextField("",30);
  private JTextField prenomRecup = new JTextField("",30);
  private JPanel choixDate;
  private JTextField jourRecup=new JTextField("",30);
  private JTextField moisRecup=new JTextField("",30);
  private JTextField anneeRecup=new JTextField("",30);
  private JButton creerCV = new JButton ("Créer le CV");
  private CV leCV;

  public Formulaire(){
    super("Formulaire");
    this.setSize(400, 400);
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setResizable(false);

    creerCV.addActionListener(new CreerCVEcouteur());

    // Panneau de saisie de la date
    choixDate = new JPanel(new GridLayout(1,3,2,1));
    choixDate.add(jourRecup);
    choixDate.add(moisRecup);
    choixDate.add(anneeRecup);

    // Panneau principal
    fond = new JPanel (new GridLayout(8,1,1,1));
    fond.add(new JLabel("FORMULAIRE DE CREATION DE CV",JLabel.CENTER));
    fond.add(new JLabel("Nom",JLabel.CENTER));
    fond.add(nomRecup);
    fond.add(new JLabel("Prénom",JLabel.CENTER));
    fond.add(prenomRecup);
    fond.add(new JLabel("Date de naissance JJ/MM/AAAA",JLabel.CENTER));
    fond.add(choixDate);
    fond.add(creerCV);

    this.setContentPane(fond);
    this.setVisible(true);

  }
  class CreerCVEcouteur implements ActionListener{
    public void actionPerformed (ActionEvent e){
      leCV = new CV(
         nomRecup.getText(),
         prenomRecup.getText(),
         Integer.parseInt(anneeRecup.getText()),
         Integer.parseInt(moisRecup.getText()),
         Integer.parseInt(jourRecup.getText())
      );
      leCV.afficher();
      dispose();
    }
  }
}
```

```
public class ClassePrincipale {
  public static void main(String args[]){
    Formulaire creation = new Formulaire();
  }
}
```

2. L'affichage du CV se fait pour l'instant dans la *sortie standard* de manière textuelle. Nous allons créer une fenêtre contenant les informations du `CV` instancié par le formulaire.
   (a) Création d'une nouvelle classe (`VisualisationCV`) qui hérite de la classe `JFrame`. Mettre les informations ne contenant que le strict minimum pour faire une fenêtre avec un panneau (`JPanel`) et un texte (`JLabel`).
   (b) Vérifier le bon fonctionnement de cette classe en instanciant un objet à partir de la classe principale.
   (c) Dans la classe `Formulaire` ajouter dans les attributs la déclaration

```
private VisualisationCV monCVGraphique;
```

   et l'instanciation après avoir appelé la méthode `afficher` de la classe `CV`

```
monCVGraphique = new VisualisationCV(leCV);
```

   Ajouter ensuite, en paramètre, un objet `CV` dans le constructeur de `VisualisationCV`.
   (d) Construction du panneau dans la partie `NORTH` du `BorderLayout` :
   — déclarer un `JPanel` nommé `nord` dans les attributs.
   — l'instancier dans le constructeur avec un `gridLayout` à 3 colonnes et 1 ligne.
   — Ajouter, en instanciant directement, 3 `JLabel` à `nord` contenant respectivement les chaines de caractères `"Photo"`, `"Nom"` et `"Age"`.
   (e) Dans la classe `CV`, construire les accesseurs de tous les attributs et un accesseur pour obtenir l'age de la personne. Remplacer le contenu des `JLabel` de la question précédente avec les informations relatives au `CV` passé en argument dans le constructeur de `VisualisationCV`.
   (f) Ajouter un panneau dans la partie `WEST` du `BorderLayout`. Le panneau est structuré avec un `GridLayout` de 3 lignes et 1 colonne. Il contient 3 boutons avec les textes : Formations, Stages, Compétences.
   (g) Pour remplir le reste de la fenêtre, on construit 3 `JTextArea` contenant les informations relatives aux 3 boutons construits à la question précédente. Ajouter un `JPanel` contenant l'un de ces `JTextArea` à la fenêtre.
   (h) En s'inspirant de `Formulaire.java`, construire la réactivité des boutons de telle sorte que lorsqu'un bouton est cliqué, le panneau correspondant est affiché.

```
class formationEcouteur implements ActionListener{
  public void actionPerformed (ActionEvent e){
    autre.removeAll();
    autre.add(t_formation);
    autre.setBackground(Color.ORANGE);
    autre.revalidate();
    repaint();
  }
}
```

   Changer la couleur des panneaux.

3. Améliorer le rendu en modifiant les composants (cf. Liste fournie dans le cours), leur place, les couleurs, le contenu des textes ...

Exercice 4.1 Système Multi-Agent modélisant l'apparition et l'évolution d'un cancer

Un modèle simplifié de l'apparition et de l'évolution d'un cancer a été construit à partir d'un modèle [**?**]. Le mécanisme décrit par ce modèle se déroule au sein d'un lieu précis du corps humain composé de cellules saines matures. Au cours du cycle cellulaire, et plus particulièrement au moment de la réplication de l'ADN, des erreurs peuvent intervenir et endommager l'ADN de la cellule et ceci avant la mitose. Lorsque les mécanismes de séparation de l'ADN n'agissent pas suffisamment pour que toutes les lésions soient éliminées, il y a alors accumulation de ces erreurs au sein du noyau. Cela va aboutir à la modification du phénotype de la cellule qui devient alors une cellule cancéreuse ; il s'agit du processus de cancérisation. Malgré les erreurs dans son ADN, la cellule tumorale ne subit pas les mécanismes de l'apoptose (une mutation dans son ADN permet de bloquer le processus) ; elle poursuit son chemin dans le cycle cellulaire et peut donc entrer en mitose et transmettre son patrimoine génétique à deux cellules filles (et ainsi de suite) ; et ce, même si les conditions environnementales (pas assez de nutriments, ni de place) ne s'y prêtent pas. Cela entraine la formation d'amas de cellules cancéreuses au milieu des cellules saines.

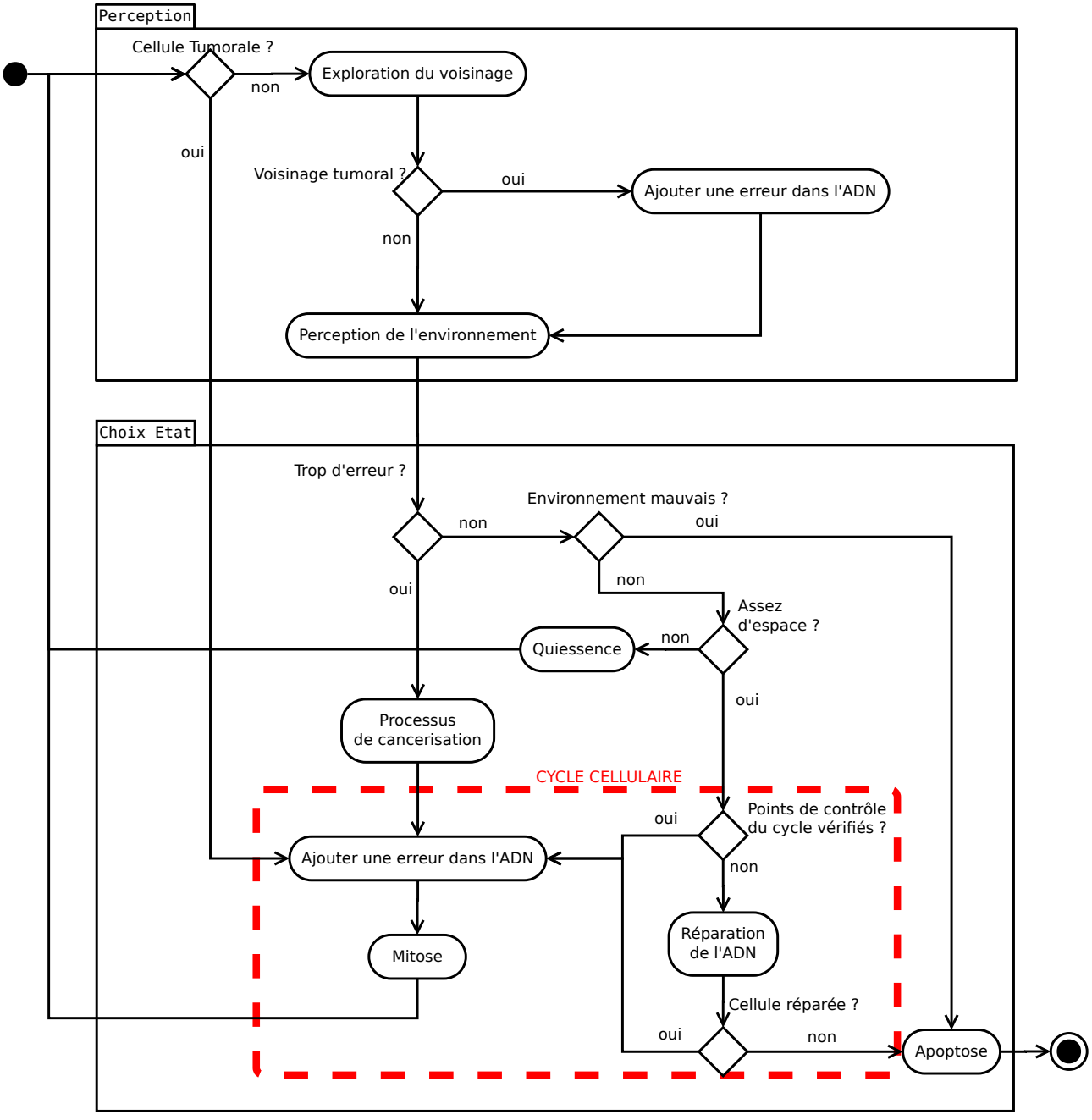


FIGURE 1 – Cycle de vie d'une cellule représenté par un diagramme d'état en langage UML.

De plus les cellules saines sont influencées par la présence de cellules cancéreuses à proximité, dû aux interactions cellule-celulle, cela favorise l'apparition d'erreurs lors de la réplication de l'ADN. Le choix fait dans ce modèle consiste à augmenter les erreurs dans l'ADN si de trop nombreuses cellules cancéreuses sont à proximité d'une cellule saine. Les cellules sont immobiles.

L'ensemble du cycle de vie d'une cellule, qu'elle soit cancéreuse ou saine, est synthétisé par l'intermédiaire d'un diagramme d'état (Fig. **??**).

Système Multi-Agent Un système multi-agent est un système composé d'un ensemble d'*agents* qui évoluent dans un *environnement*. Ces agents interagissent entre eux et avec l'environnement. Dans le modèle présenté ci-dessus, les cellules sont des agents tropiques car il s'agit de cellules dont le comportement est dicté par la perception qu'elles ont de leur environnement.

Voici une représentation textuelle de l'environnement initial avec des étapes de la progression du cancer.

```
Environnement initial : des cellules saines
      o      o      o      o      o
      o      o      o      o      o
      o      o      o      o      o
      o      o      o      o      o
      o      o      o      o      o
```

⇓

```
Environnement avec des cellules saines endommagées
      o      o      o      ---    ---
      ---    ---    --     ---    ---
      --     --     ---    --     -
      ---    ---    ---    -      ---
      o      ---    o      -      ---
```

⇓

```
Environnement avec des cellules cancéreuses
      ---    ---    -****  --*    ---
      ---    ****   ****** -****- --*
      ---    ---*   ----   -**    ---
      ---    ---    ***    --***  -*****
      o      ---    ----   ---    ----
```

⇓

```
Environnement avec uniquement des cellules cancéreuses : fin de vie
      *****  *****  *****  ****** *****
      ****   ****   ****** ****** *****
      *****  *****  ****** ****** *****
      ****   ****   ****** *****  *****
      *****  *****  ****** ****** *****
```

1. En vous aidant du diagramme d'état (Fig **??**), proposer une représentation objet du problème. Voici les questions à se poser pour y parvenir :
   — Combien de classes sont nécessaires ?
   — Que doit faire la classe principale ?
   — Parcourir de diagramme d'état d'une cellule et identifier toutes ses interactions. Avec qui ? Quelles informations sont nécessaires ?
   — Comment être sûr que les objets vont pouvoir dialoguer entre eux ? Simuler un dialogue entre eux. En déduire les attributs et méthodes qu'ils devront avoir.
   — Penser à utiliser l'héritage pour représenter les différents type de cellules et éviter les répétitions entre deux classes qui se ressemblent.

Références

[1] P. Gerlee, and A.R.A. Anderson, An evolutionary hybrid cellular automaton model of solid tumour growth, *Journal of theoretical biology*, **246**, (4), 583–603, 2007

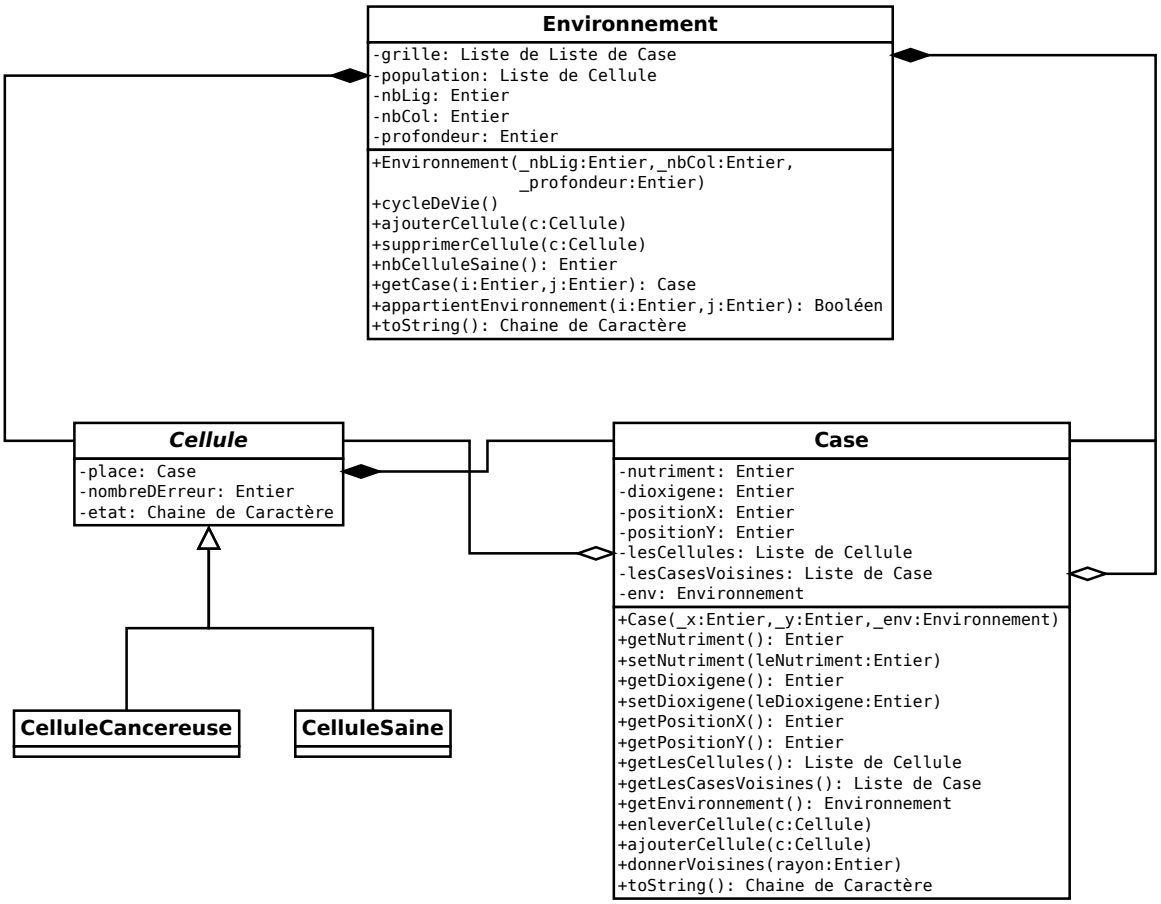


FIGURE 2 – Diagramme de classes partiel du modèle en langage UML.

2. Voici le diagramme de classes construit avec 5 classes : `Environnement`, `Case`, `Cellule`, `CelluleSaine` et `CelluleCancereuse`. Détailler les 7 relations entre toutes ces classes : il y a des liens — traits ou flèches — de différents types entre les classes sur la Fig. **??**, il faut les identifier et expliquer ce que cela implique.
   Exemple : Dans notre modèle l'`Environnement` est **composé** de plusieurs `Case`s organisées de manière matricielle. Il peut y avoir plusieurs `Cellule`s dans une même `Case`.
   Attention, la classe `Cellule` est différente des autres classes.

Détails de conception du modèle ainsi construit. L'`etat` d'une `Cellule` indique si la cellule déclenche une mitose, une quiescence ou une appoptose. C'est l'`Environnement`, qui à partir de cet état, va effectuer les modifications : création de cellule fille ou destruction d'une cellule. Pour simuler la simultanéité de l'évolution des cellules au sein de l'`Environnement`, une des `Cellule`s est choisie aléatoirement et une itération de son cycle de vie (Fig. **??**) est réalisée.

3. Récupérer les fichiers `Environnement.java` et `Case.java`. Regarder la structure de la documentation. Dans la classe `Environnement`, expliquer comment fonctionne le constructeur et la méthode `cycleDeVie()`. Expliquer comment fonctionne l'opérateur `instanceof`.
4. Seuls les attributs des classes `Cellule`, `CelluleSaine` et `CelluleCancereuse` sont donnés dans ce diagramme de classes (Fig. **??**) : les méthodes ne sont pas indiquées. En parcourant le diagramme d'état (Fig. **??**), identifier les méthodes nécessaires au fonctionnement du programme. Penser à introduire les :
   — constructeurs ;
   — accesseurs ;
   — surcharges de la méthode `toString()` ;
   — méthodes relatives à l'évolution d'une `Cellule`.
   Compléter le diagramme de classes avec les nouvelles méthodes.
5. La classe principale, à ajouter au diagramme de classe, de ce programme réalise les actions suivantes :
   — Déclaration d'un `Environnement` ;
   — Instanciation de cet `Environnement` (choisir des paramètres raisonnables) ;
   — Tant qu'il y a des `CelluleSaine` dans cet `Environnement`, continuer à appeler la méthode `cycleDeVie` de l'`Environnement` et afficher cet `Environnement` toutes les 10 itérations.

Détails de conception du modèle ainsi construit. La cellule saine a deux chances sur trois d'être réparée complètement et si elle n'est pas réparée elle a autant de chance de déclencher l'apoptose ou la mitose (avec les erreurs de réplication que cela introduit). Enfin, sachant que les cellules de ce modèle sont immobiles ; pour éviter qu'un trop grand nombre de cellules tumorales s'accumulent au même endroit, l'apoptose se déclenche automatiquement dès qu'une cellule cancéreuse se trouve sur une case où il y a déjà 5 autres cellules.

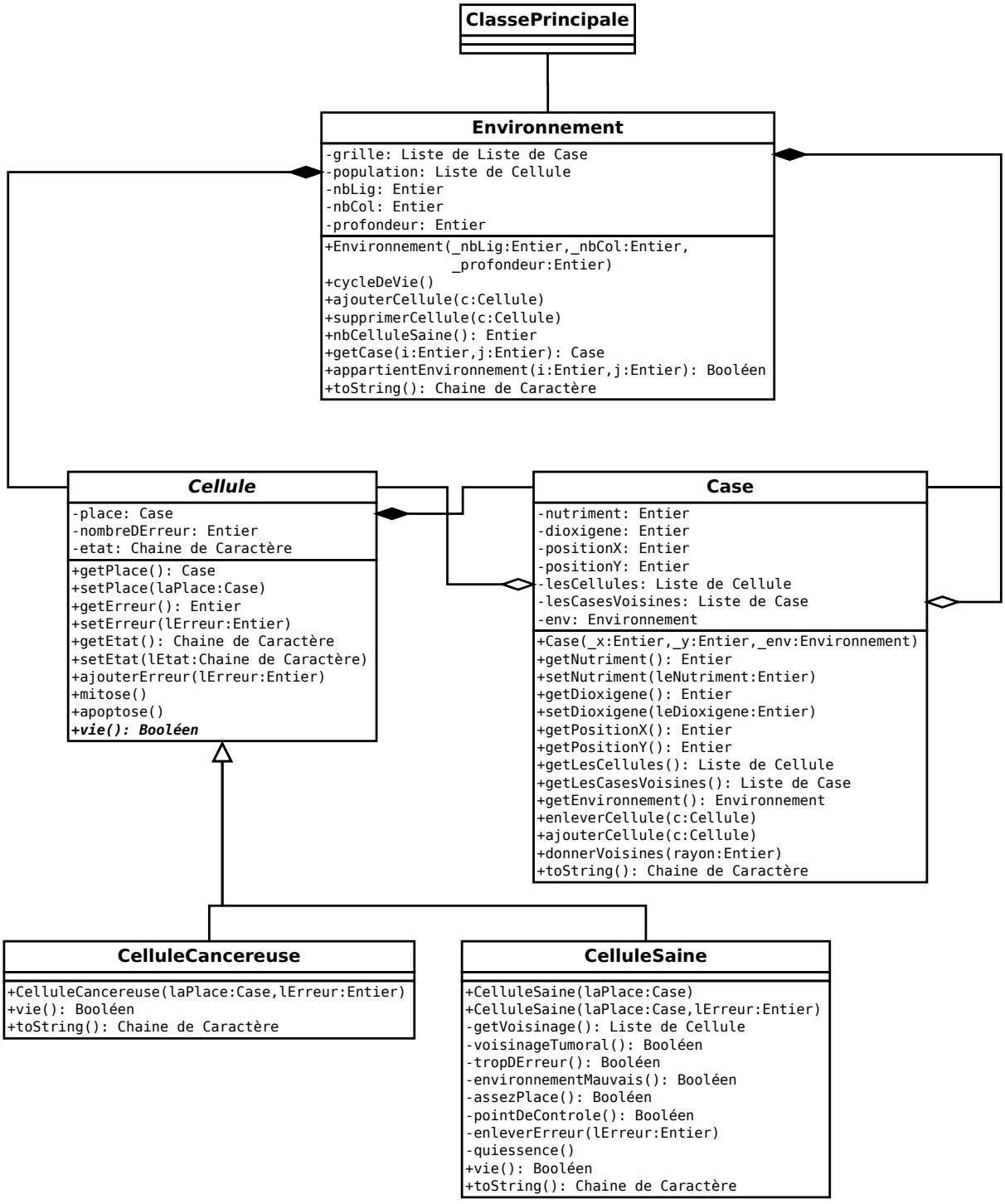


FIGURE 3 – Diagramme de classes complet du modèle en langage UML.

6. Programmer le diagramme de classes complet Fig. **??**.
7. Compiler et exécuter le programme pour obtenir le résultat sur un environnement de taille 10×10.
8. Modifier les paramètres pour observer des comportements différents.
9. Modifier le programme pour passer les paramètres du système en arguments de la `ClassePrincipale` à l'exécution. Cela permet de ne plus avoir à compiler les fichiers pour réaliser plusieurs expérience avec des paramètres différents.

N° identification :

Documents et accès à internet autorisés.
Copier les fichiers sources dans une archive nommée : `identification.tar`.

Exercice 5.1 (/**10)** Gestion des appareils médicaux dans un hôpital

Vous êtes ingénieur biomédical dans un hôpital et vous devez gérer un *stock* d'*appareils* pour différents *services*. Pour réussir cette tâche, vous avez besoin d'un programme informatique que vous allez réaliser en java et dont voici la description et le diagramme de classe.

1. Implémenter la classe `Appareil`, elle possède
   - 3 attributs : 2 chaînes de caractères, son nom et son statut et 1 entier, son prix ;
   - 2 constructeurs qui changent le statut de l'appareil en lui affectant la valeur `"libre"` :
     (a) le constructeur par défaut qui affecte des valeurs nulles
     (b) le constructeur qui prend en arguments le nom et le prix et les affecte aux attributs.
   - 4 méthodes :
     (a) `estLibre` qui ne prend pas d'argument et retourne un booléen indiquant si l'appareil libre.
     (b) le mutateur de l'attribut du statut
     (c) l'accesseur de l'attribut du nom
     (d) l'accesseur de l'attribut du prix
2. Implémenter la classe `Stock`, elle possède
   - 1 attribut : une liste d'`Appareil`
   - 1 constructeur : il instancie l'attribut et y ajoute les `Appareil` suivant en les instanciant :
     - `"pousseSeringue"` à 250€
     - `"irm"` à 600000€
     - `"scanner"` à 400000€
     - `"appDialyse"` à 80000€
     - `"thermometre"` à 60€
     - `"ventilateur"` à 40000€
     - `"tensiometre"` à 100€
   - 1 méthode `getUnAppareil` qui prend en argument une chaine de caractère et qui retourne un `Appareil`. Cette méthode a pour but de vérifier si l'appareil est disponible en stock en utilisant son nom (passé en argument). Pour cela, il faut parcourir la liste des `Appareil` du `Stock` et *dès qu'*un élément de cette liste est *à la fois* libre *et* qu'il se nomme de la même façon, il faut le rendre occupé (changer son statut) et le retourner. Par défaut, cette méthode retourne un `Appareil` instancié par le constructeur par défaut de la classe `Appareil`.

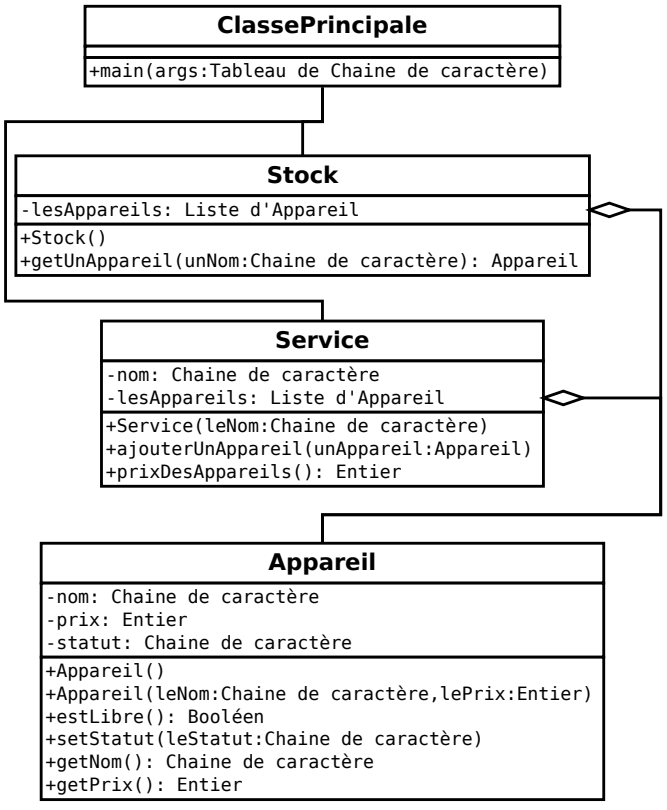


Diagramme de classe du progamme.

3. Implémenter la classe `Service`, elle possède
   - 2 attributs : une chaine de caractère qui est son nom et une liste d'`Appareil` qui sont affectés au `Service`.
   - 1 constructeur qui prend en paramètre une chaine de caractère affecté à l'attribut et qui instancie la liste d'`Appareil`.
   - 2 méthodes
     (a) `ajouterUnAppareil` qui prend en argument un `Appareil` et qui l'ajoute à la liste des `Appareil` du service
     (b) `prixDesAppareils` qui ne prend pas d'argument et qui retourne le prix total des appareils du service

4. Implémenter la classe principale, elle possède une méthode principale qui fait les actions suivantes
   — Instancier un `Stock` ;
   — Instancier les services *imagerie*, *ventilation* et *dialyse* ;
   — Ajouter les appareils du stock aux services concernés.
   — Calculer et afficher la valeur de la totalité des appareils pour chacun des trois services.
5. Ajouter la ligne nécessaire à la gestion des `Liste de ...` dans les classes concernées (`import`).
6. Compiler et exécuter votre programme.

Exercice 5.2 Améliorations

Cocher les cases des améliorations apportées. Chaque amélioration réalisée rapporte les points indiqués.

☐ **(/3)** Le code est bien indenté, compile avec la commande :

```
moi@machine:~$
```

et s'exécute (sans erreur) avec la commande (sans indiquer les arguments) :

```
moi@machine:~$
```

☐ **(/2)** Documenter le code

☐ **(/5)** Une interface graphique a été mise en place pour la saisie de nouvel appareil au `Stock` ou pour l'affichage des `Appareil` relatif à chaque `Service`.

☐ **(/2)** Surcharger les méthodes `toString` des objets pour réaliser l'affichage dans la *sortie standard*.

☐ **(/3)** Mettre en place la saisie à l'exécution du programme d'arguments pour ajouter un appareil au `Stock`. Indiquer les arguments à ajouter à l'exécution

```
moi@machine:~$
```

☐ **(/7)** Mettre en place de l'héritage pour spécialiser les `Appareil` en deux classes filles :
   — les `AppareilCourant` qui doivent être présent dans le `Stock` en grande quantité ;
   — les `AppareilSpecialise` qui coûtent cher et qu'il faut entretenir régulièrement.
   Ces derniers ont un *taux d'usure* (compris entre 0 et 1) qu'il faut surveiller et réparer si nécessaire, tandisque les `AppareilCourant` ne peuvent être réparés.

   1. Produire le diagramme de classe associé.
   2. Implémenter les deux classes filles et faire la modification de la classe `Appareil`
   3. Ajouter une méthode, dans la classe `Stock`, qui calcule le coût de réparation des `AppareilSpecialise` en parcourant les `Appareil` en `Stock`. Le coût de réparation est le prix de l'`Appareil` multiplié par le *taux d'usure*.