ECOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE

Cycle supérieur- 2CSSIL
Semestre 1 - PDC

2015 - 2016

~ TP 1 _ PDC ~

Les principes avancés de POO et les Patrons GRASP

Présenté par :

- CHEROUANA Wissem
- LERARI Sihem

Groupe : 2CSSIL

1. Sommaire

2. Introduction :

Afin de se familiariser avec les patrons de conception, il nous a été demandé d'analyser un système existant issu d'un projet de débutants en POO : il s'agit du jeu de l'oie.

Après une analyse approfondie du code source du jeu, nous avons pu desceller pas mal de problèmes de conception, ainsi que quelques portions de code que nous avons jugé être des failles de programmation impactant négativement la qualité du jeu livré à savoir : des dépendances abondantes entre classes, des méthodes non cohésives et surtout l'ampleur de l'espace mémoire utilisé.

Ainsi nous essayerons donc dans une première partie de tracer le diagramme de classes du système existant, ensuite nous recenserons et expliquerons les différents problèmes trouvés, nous enchaînerons avec les diverses alternatives de solutions en mettant l'accent sur les patrons de conception vus en cours, et retracerons à la fin le nouveau diagramme de classes , histoire de savoir si le code a été raffiné et s'adapte mieux aux changements.

3. Analyse du système :

3.1. Analyse du code source :

Le code source du jeu donné compte 26 classes qui couvrent les 2 aspects : graphique et fonctionnel (classes avec méthodes).

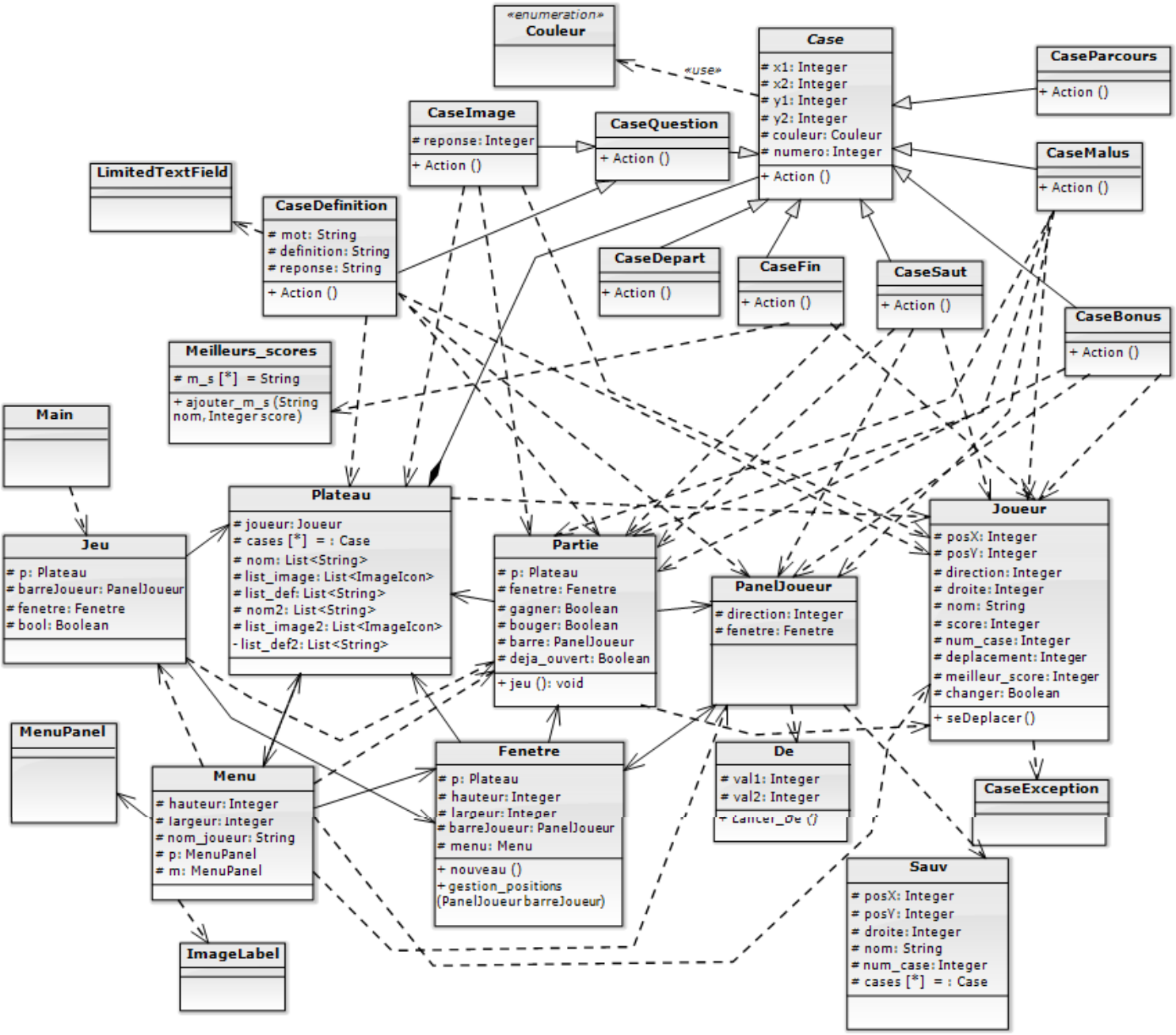
Nous estimons que toutes ces classes sont indispensables au bon fonctionnement du jeu. Certaines sont trop condensées et la qualité n'y est pas vraiment, à titre d'exemple : **Plateau et Joueur**, d'autres font des requêtes sans avoir au préalable la responsabilité de le faire, d'autres ne tiennent pas en compte la possibilité d'extension des fonctionnalités ainsi que l'évolution du code et bien plus encore…

Nous avons jugé également que **le principe d'encapsulation des attributs n'est pas du tout respecté**, ce point sera abordé en détails par la suite.

Toutes les classes se trouvent dans le même package par défaut ce qui réduit considérablement la compréhension et la lisibilité du code. En effet, une des meilleures pratiques de la POO est de scinder les classes entre classes techniques et celles relatives au graphisme, et mettre chacune d'elles dans le package adéquat **(comportements / interfaces graphiques).**

3.2. Reconstitution du diagramme de classes :

L'analyse du code nous a mené à tracer le diagramme de classes du système. Ceci dit, il s'est avéré que ce DCU compte un nombre énorme de dépendances, nous nous sommes focalisées alors sur **un diagramme montrant explicitement les dépendances entre les différentes classes, leurs attributs et méthodes**. Pour une meilleure visibilité, et histoire de bien voir les problèmes, nous n'avons pas mentionné les attributs relatifs aux interfaces graphiques (Jbutton, JTextField, JLabel…).



Comme vous pouvez le voir, ce DCU compte énormément de dépendances.

3.3. Différents problèmes recensés :

Dans cette partie, nous essayerons d'expliquer en détails les problèmes de conception dans le programme de ce jeu, ceci peut être parfois accompagné de captures des bouts de code tirés du code source original :

a. Plateau :

- **1er problème : Comptant près de 10 dépendances** avec les autres classes : Celle-ci ne répond pas du tout au **principe de faible couplage**. Ainsi cela augmente considérablement son impact si des changements seront effectués sur elle par la suite.

- **2ème problème :** On constate que l'étudiant s'est beaucoup focalisé sur une interface utilisant les pixels pour faire bouger le pion sur le plateau, **ce qu'il l'a mené à écrire des boucles répétitives** essayant de déterminer toutes les positions des cases dans le plateau. Voici un aperçu :

```
for(int i=2;i<12;i++)
{
    if(cases_couleurs[i]==Couleur.autre) cases[i]=new CaseParcours(i,cases[i-1].x1+75,cases[i-1].x2+75,40,88);
    else if(cases_couleurs[i]==Couleur.vert) cases[i]=new CaseBonus(i,cases[i-1].x1+75,cases[i-1].x2+75,40,88);
    else if(cases_couleurs[i]==Couleur.rouge) cases[i]=new CaseMalus(i,cases[i-1].x1+75,cases[i-1].x2+75,40,88);
    else if(cases_couleurs[i]==Couleur.orange) cases[i]=new CaseSaut(i,cases[i-1].x1+75,cases[i-1].x2+75,40,88);
    else if(cases_couleurs[i]==Couleur.bleu) cases[i]=new CaseDefinition(i,cases[i-1].x1+75,cases[i-1].x2+75,40,88);
    else if(cases_couleurs[i]==Couleur.rose) cases[i]=new CaseImage(i,cases[i-1].x1+75,cases[i-1].x2+75,40,88);
    cases[i].couleur=cases_couleurs[i];

}
for (int i=12;i<21;i++)
{
    if(cases_couleurs[i]==Couleur.autre) cases[i]=new CaseParcours(i,cases[11].x1,cases[11].x2,cases[i-1].y1+60,cases[i-1].y2+60);
    else if(cases_couleurs[i]==Couleur.vert) cases[i]=new CaseBonus(i,cases[11].x1,cases[11].x2,cases[i-1].y1+60,cases[i-1].y2+60);
    else if(cases_couleurs[i]==Couleur.rouge) cases[i]=new CaseMalus(i,cases[11].x1,cases[11].x2,cases[i-1].y1+60,cases[i-1].y2+60);
    else if(cases_couleurs[i]==Couleur.orange) cases[i]=new CaseSaut(i,cases[11].x1,cases[11].x2,cases[i-1].y1+60,cases[i-1].y2+60);
    else if(cases_couleurs[i]==Couleur.bleu) cases[i]=new CaseDefinition(i,cases[11].x1,cases[11].x2,cases[i-1].y1+60,cases[i-1].y2+60);
    else if(cases_couleurs[i]==Couleur.rose) cases[i]=new CaseImage(i,cases[11].x1,cases[11].x2,cases[i-1].y1+60,cases[i-1].y2+60);
    cases[i].couleur=cases_couleurs[i];

}
```

Ces boucles répétitives ainsi que blocs de conditions mis en place pour savoir quel type de case instancier avec ses coordonnées, ne respectent pas en réalité 2 principes :

Le principe OCP n'est pas respecté : en effet si on veut rajouter d'autres types de cases **➔ ceci nous mènera à toucher au code pour en rajouter d'autres blocs de conditions.** Si on souhaiterait également augmenter par la suite le nombre de cases dans le plateau **➔ d'autres boucles « For » seront ajoutées** pour instancier les cases et adapter leurs dispositions sur le plateau : Donc ce bout de code n'est pas fermé à la modification.

Le principe DRY (Don't Repeat Yourself) : Ces boucles sont très répétitives avec à chaque fois un seul changement effectué sur l'un des paramètres du constructeur de la case à savoir (**x1, ou x2 ou y1 ou y2** qui déterminent les coordonnées de la case).

Ceci dit ceci pouvait être remplacé simplement par une fonction paramétrée.

- **3ème problème :** Le constructeur du plateau fait appel à la fonction : **genererPlateau(Couleur[] cases_couleurs) ,** voici un bout de code :

```
public void genererPlateau(Couleur[] cases_couleurs)
{
    int i=0,j=0;
    for(i=1;i<100;i++)
    {
        cases_couleurs[i]=Couleur.autre;
    }
    i=0;
    while(i<5)
    {
        j=((int) (Math.random()*99))+2;
        if(cases_couleurs[j]==Couleur.autre)
        {
            cases_couleurs[j]=Couleur.vert;
            i++;
        }
    }
    i=0;
    while(i<5)
    {
        j=((int) (Math.random()*99))+3;
        if(cases_couleurs[j]==Couleur.autre && cases_couleurs[j-2]!=Couleur.vert)
        {
            cases_couleurs[j]=Couleur.rouge;
            i++;
        }
    }
```

Dans cette méthode, on constate que l'étudiant ne génère pas des cases aléatoirement, mais plutôt leurs couleurs aléatoirement, **la couleur est selon lui un identifiant de la case**, et donc lors de l'appel de cette méthode dans le constructeur du plateau, il essaye d'instancier les cases selon leur couleurs stockées dans un tableau de couleurs :

```
genererPlateau(cases_couleurs);

 /*initialisation des cases */

cases[0]=new CaseDepart(0,200,270,30,80);
cases[0].couleur=Couleur.jaune;
if(cases_couleurs[1]==Couleur.autre) cases[1]=new CaseParcours(1,285,354,40,88);
else if(cases_couleurs[1]==Couleur.vert) cases[1]=new CaseBonus(1,285,354,40,88);
else if(cases_couleurs[1]==Couleur.rouge) cases[1]=new CaseMalus(1,285,354,40,88);
else if(cases_couleurs[1]==Couleur.orange) cases[1]=new CaseSaut(1,285,354,40,88);
else if(cases_couleurs[1]==Couleur.bleu) cases[1]=new CaseDefinition(1,285,354,40,88);
else if(cases_couleurs[1]==Couleur.rose) cases[1]=new CaseImage(1,285,354,40,88);
```

Appel de la fonction genererPlateau (cases_couleurs) où cases couleurs contient la couleur de chacune des cases du plateau

Selon la couleur générée aléatoirement ➔ On instancie la case adéquate

Ceci ne respecte pas **le principe OCP.** Effectivement si dans le futur, on décide de changer la couleur d'une case donnée ➔ Cela mènera forcément à changer toutes les parties du code d'instanciation de la case concernée.

Exemple : donner la couleur Violet à la place de Rose pour la case définition, donc les changments à effectuées sont les suivants :

if(cases_couleurs[1]==**Couleur.rose**) cases[1]=new CaseImage(1,285,354,40,88);

if(cases_couleurs[1]==**Couleur.violet**)cases[1]=new CaseImage(1,285,354,40,88);

Ceci se fait dans toutes les parties du code d'instanciation des cases vu qu'elle se repose sur les couleurs ➔ Donc ouverture à la modification.

- **4ème problème:** Le chargement des images qui vont apparaitre dans une fenêtre de question image se fait dans le constructeur du plateau. Le fichier contenant les définitions qui vont apparaitre dans une fenêtre de question définition se charge également dans le constructeur de plateau. Ceci ne respecte pas 2 principes :

Le principe Expert en information n'est pas respecté : En effet le plateau n'étant pas du tout la classe à qui nous devons assigner la responsabilité de chargement des images et fichier de définitions.

Par ailleurs, imaginons le scénario qu'au cours du jeu, le joueur n'est tombé sur aucune des cases images ou définitions, ainsi le chargement a été fait pour rien ➔ donc consommation d'espace mémoire pour rien.

Le principe forte cohésion n'est pas respecté : En effet, le plateau compte désormais plus d'une responsabilité (génération aléatoire des cases + chargement des images et fichier des définitions) ➔ Les méthodes ne sont pas cohésives et ça diminue considérablement la compréhension de la classe.

- **5ème problème :** L'étudiant a déclaré pas mal d'attributs statiques **ne respectant pas ainsi le principe d'encapsulation**, et n'importe quelle autre pourra leur accéder et les utiliser par la suite.

```
protected static List<String> nom;
protected static List<ImageIcon> list_image;
protected static LinkedList<String> list_def;
protected static LinkedList<String> nom2=new LinkedList<String>();
protected static LinkedList<ImageIcon> list_image2=new LinkedList<ImageIcon>();
protected static LinkedList<String> list_def2=new LinkedList<String>();
```

b. Joueur :

- **1er problème : Presque la quasi-totalité des autres classes dépendent sur elles** ➔ Donc elle ne répond pas au **principe de faible couplage**.

- **2ème problème :** Pour faire diriger le pion dans le plateau, l'étudiant a implémenté une méthode **seDeplacer()** qui est en réalité un ensemble de blocs de conditions if…else répétitifs où quelques paramètres seulement changent ➔ Donc elle ne répond pas au **principe DRY**.

- **3ème problème :** L'étudiant a déclaré pas mal d'attributs statiques ne respectant pas ainsi **le principe d'encapsulation**, et n'importe quelle autre pourra leur accéder et les utiliser par la suite.

- **4ème problème :** Dans l'implémentation de la fonction seDeplacer(), l'étudiant fait appel continuellement à Plateau.cases[num_cases] comme les tableau des cases dans plateau est déclaré **static** ➔ Donc ça ne répond pas au **principe Ne pas parler au inconnus**.

```
if(getPosX()>=905 && getPosY()==52) droite++;
else if(getPosX()>=Plateau.cases[num_case].x1-175 && Plateau.cases[num_case].y1-getPosY()<100)
{
    posX=(int) ((Plateau.cases[num_case].x1+Plateau.cases[num_case].x1-265)/2);
    img=images[3];
    img2=null;
}
```

c. Différents types de cases :

- **1er problème :** On remarque que presque tous les types de cases dépendent d'autres classes et y accèdent facilement : ➔ ça ne répond pas au **principe Ne pas parler au inconnus, encapsulation et faible couplage.**

```
public void Action()
{
    /* avancer le joueur de deux cases */
    if(Plateau.cases[Joueur.num_case-2].couleur!=Couleur.vert && Joueur.num_case>2)
    {
        JOptionPane.showMessageDialog(null,"Allez à la case "+(Joueur.num_case-2) ,"Case Malus" ,
        if (Joueur.score>=10) Joueur.score-=10;
        Joueur.num_case-=2;
        PanelJoueur.direction=2;
        Partie.bouger=true;
    }

}
```

Dans cette méthode brève par exemple qui est propre à la case Malus, on a accédé aux attributs de 4 classes distinctes du fait que tous leurs attributs sont déclarés **static ➔ On voit que la notion orienté objet (instaciation) n'existe pas vraiment dans ce code.**

d. Classe Sauv :

```
public class Sauv implements Serializable {
    /* Les parametres à sauvgarder */
    protected int posX;
    protected int posY;
    protected int droite;
    protected String nom;
    protected int score;
    protected int num_case;
    protected Case[] cases;

    /* Constructeur */
    public Sauv(int posX, int posY, int droite, String nom, int score,
            int num_case, Case[] cases)
    {
```

Cette classe se contente seulement de déclarer les paramètres à sauvegrder d'une partie donnée sans avoir au préalable les fonctions de sauvegarde. ➔ Expert en information de sauvegarde, mais qui se trouve isolée et 2 classes dépendent d'elle : Menu et PanelJoueur.

e. Classe PanelJoueur :

- **1er problème :** C'est un panel servant au graphisme qui dépend de plusieurs classes : toujours les mêmes problèmes : encapsulation + fort couplage + parler beaucoup aux inconnus.
- **2ème problème :** Elle implémente la fonction de sauvegarde d'une partie sans avoir même les informations nécessaires pour le faire ➔ du coup il dépend de la classe Sauv. ➔ **N'est pas l'expert en information**, ceci mène aussi à **la non cohésion.**

```
public void mouseClicked(MouseEvent ev) {
    try{
        /* Si le joueur clique sur le boutton savegarde */
        if(ev.getSource()==sauvegarde)
        {
            /* afficher le FileChooser */
            int returnVal = chooser.showOpenDialog(getParent());
            if(returnVal == JFileChooser.APPROVE_OPTION) { }
            try {
            /* sauvgarder les objets */
            out = new ObjectOutputStream(
            new BufferedOutputStream(
            new FileOutputStream(
            new File(chooser.getSelectedFile().getAbsoluteFile().toString())));
            Sauv s=new Sauv(fen.p.joueur.posX,fen.p.joueur.posY,fen.p.joueur.droite,
            //écriture les objets dans un fichier
            out.writeObject(s);
```

- **3ème problème :** Dans son constructeur la classe PanelJoueur s'occupe de lancer les dès ➔ hors on n'instancie pas vraiment un objet de De, on n'y accède directement et on untilise sa méthode publique Lancer_De() vu qu'elle est déclarée statique. ➔ **Encore une fois : Ne pas parler aux inconnus, encapsulation ne sont pas respectés.**

```
if(Partie.bouger==false) // pour eviter que
{
    Partie.deja_ouvert=false;
    Partie.bouger=true;
    int depla=De.Lancer_De();
    label1.setIcon(des[De.val1-1]);
    label2.setIcon(des[De.val2-1]);
```

- Aussi nous estimons que cette classe est relative au graphisme, il serait préférable de ne pas lui assigner la responsabilité de créer les dés ➔ **N'est pas le créateur.**

f. Classe De :

- On remarque la déclaration des attribus **static** ;
- On a attribué à un dé 2 valeurs et la méthode Lancer_De() retourne la somme des deux valeurs ➔ L'étudiant n'a pas pensé à l'extension du nombre de dès pouvant être lancés dans une partie et a figé ainsi le nombre des valeurs à 2 ; et lors de la création (ou plutôt accès direct à la méthode statique Lance_De() depuis la classe PanelJoueur, il récupère les 2 valeurs (val1 et val2 déclarés aussi **static**), ie : imaginons qu'on souhaiterait désormais lancer 3 dès alors selon son approche, on devra déclarer dans la classe De un nouvel attribut val3 et l'additionner avec les 2 autres valeurs dans la méthode Lancer_De() ; et ajouter aussi dans la classe PanelJoueur un bout de code permettant de récupérer val3 ➔ **OCP n'est pas du tout respecté .**

```
public class De implements Serializable
{
    protected static int val1=0,val2=0;
    public static int Lancer_De()
    {
        int i=0;
        /* recuperer la valeur du premier dé */
        i=Math.min(6, ((int) (Math.random()*6))+1);
        val1=i;
        /*recuperer la valeur du deuxiemme dé */
        i=Math.min(6, ((int) (Math.random()*6))+1);
        val2=i;
        /* retourner la somme des deux valeurs */
        return val1+val2;
    }
}
```

g. Classe Jeu :

```
protected static Boolean bool=false; //**
public Jeu()
{
    /* Creer le menu principal */
    Menu menu=new Menu();
    while(bool==false)
    {

        try { Thread.sleep(50); } catch (InterruptedException e) { e.printStackTrace(); }
    }
    /* Lancer la partie */
    Partie.jeu();
```

- Comme nous le remarquons ici, la classe Jeu contient parmi ces attributs un **Booléen static : bool** initialisé à false ;
- Son constructeur quant à lui, instancie la classe Menu qui est est la fenêtre du début ;
- Le bout de code qui suit est une boucle **tant que (bool ==false) {Thread.sleep(50)}.** Elle sert en fait, à boucler infinément jusqu'à ce qu'on ait changé bool à **true,** à partir de la classe Menu. En effet, en cliquant sur nouvelle partie une fenêtre demandant au joueur de saisir son nom s'affiche, ce n'est qu'après avoir cliquer sur valider, qu'on accède à ce booléen vu qu'il est static et on le rend à **true**, ce qui permet d'afficher la fenêtre principale (On sort de la boucle tant que et on apelle Partie.jeu()) ; En grossomodo, ce paramètre est un point bloquant l'affichage de la fenêtre principale en créant une boucle infinie qui la stoppe et si l'on puisse dire l'endormir avnt d'âtre affichée. Voici le bout de code de Menu qui change la valeur de bool en true :

```
/* Si le joueur clique sur valider on affiche la fentre du jeu */
else if(arg0.getSource()==ok)
{
    Partie.bouger=false;
    Partie.gagner=false;
    PanelJoueur.direction=0;
    this.setVisible(false);
    fen=null;
    fen=new Fenetre(this);
    PanelJoueur.label_nom.setText(PanelJoueur.label_nom.getText()+texte.getText());
    nom_joueur=texte.getText();
    Jeu.bool=true;
    Partie.p=fen.p;
```

Toujours les mêmes problèmes sont présents (fort couplage / pas d'encapsulation…) auquels s'ajoute un vrai problème qui est la consommation énorme de mémoire en créant ce point bloquant.

NOTA : Presque toutes les autres classes présentent des problèmes de dépendances et d'encapsulation (attributs déclarés **protected static**), donc nous n'allons pas nous attarder dessus.

4.Solution :

4.1. Solutions des problèmes identifiés :

Après avoir survolé divers problèmes présents sur le code, nous pouvons les résumer dans les points suivants :

- Pas d'encapsulation : presque la totalité des attributs sont déclarées static ce qui les rend accessible de n'importe quelle partie dans le code.
- Parler aux inconnus et fort couplage découlent directement du point cité juste en dessous.
- **Très important !! Dans tout le code on a négligé l'instanciation des classes (Le créateur ne figure pas trop dans le code)**, vu que cela a été simplifié avec les attributs et méthodes **static**.
- Quelques classes ne sont pas les expertes en information.
- Certaines classes ne sont pas très cohésives.
- Ne pas prendre en considération l'extension du code.
- Ouverture aux modifications.
- ….

Nous essayerons donc maintenant de proposer des solutions plus ou moins pertinentes pour rendre les classes plus compréhensibles, moins dépendantes, reduisant l'impact des changements et possibilité de réutiliser les classes et modules.

a. Plateau :

- Pour ce qui est du problème des boucles répétitives servant à instancier les cases avec les coordonnées adéquates sur le plateau (OCP et DRY non respectés). Cette boucle sera carrément enlevée car l'instanciation dépend sur les couleurs.
- La fonction genererPlateau (Cases[] cases_couleurs) qui génère un tableau de couleurs (couleur estimée comme étant un identifiant) pour ensuite instancier les cases : ➔ Elle sera remplacée par une fonction :
  Public void RemplirPlateauAleatoirement () : qui s'occupe de générer elle-même les cases aléatoirement, elle appellera à son tour une petite fonction : **public Case ChoixCase (int choix, int val)** qui instancie explicitement une case adéquate (choix : servant à définir quel type de case et val sa valeur) ;
  Donc par la suite, si on veut changer de couleur, cela ne posera plus de problème au programmeur vu que la couleur est propre à la case, et dès que cette dernière est instanciée, la couleur lui est attribuée (changement et affectation de la couleur dans la case elle-même). ➔ Respect d'OCP.

  Voici quelques captures de la nouvelle implémentation :

```
public Case ChoixCase(int choix, int val) //Choix d'une case
{
    Case cas = null;
    int x1 = 0,x2 = 0,y1 =0,y2 = 0;
    switch (choix)
    {
    case 1:
        CaseDepart dep = new CaseDepart(val, x1,x2,y1,y2);
        cas= dep;
        break;
    case 2:
        CaseMalus mal = new CaseMalus(val, x1,x2,y1,y2);
        cas= mal;
        break;
```

```
public void RemplirPlateauAleatoirement() //Remplir le tableau aléatoirement
{
    int i ;
    ajouterCase(ChoixCase(1,0),0); // ajout de la case départ
    ajouterCase(ChoixCase(7,100),100); // ajout de la case fin

    for (i=0; i<5; i++) // 5 cases de bonus
    {
        int indice = ((int)(Math.random()*98))+1;
        while(cases[indice] != null)
        {
            indice = ((int)(Math.random()*98))+1;
        }
        ajouterCase(ChoixCase(3,indice+1),indice);
    }

    for (i=0; i<5; i++) // 5 cases de malus
    {
        int indice = ((int)(Math.random()*97))+2;
```

- Pour ce qui du chargement des images et fichiers de définitions dans le constructeur de Plateau, ça ne sera plus fait dans cette classe respectant ainsi le principe SRP et forte cohésion, cette tâche de chargement sera déléguée par contre aux cases image et (respectivement definition) qui sont les expertes en information réelles.**(chargement plus précisément dans la méthode Action()) ➔** Cela permettra aussi d'éviter la consommation d'espace mémoire pour rien.
- Les attributs **static** inutiles seront enlevés, les autres seront remplacés par des attributs privés avec setters et getters ➔ Encapsulation respectée.

Remarque : Les images seront désormais stockés dans un fichier texte ou chaque ligne correspond au nom de l'image+ son emplacement.

b. Joueur :

- Tous les attributs propres au joueur sont rendus private pour respecter l'encapsulation.

- Les boucles répététives servant au déplacement du joueur sur le plateau seront quant à elles remplacées par une seule fonction paramétrée ➔ DRY respecté.

c. Les cases :

- La classe case est déclarée comme étant **abstract** et contient la signature d'une méthode abstaraite Action() qui est redéfinie dans tous les fils de Case afin de différencier le comportement et agir de manière polymorphique ➔ Principe de polymorphisme respecté.
- La méthode Action() redéfinie dans toutes les cases, prendra désormais un paramètre qui est Partie : **public void Action (Partie p),** sachant que partie contient des références vers les classes suivantes : (Plateau, PanelJoueur, Joueur), ➔ Donc au lieu d'accéder aux attributs des autres classes montrés un peu plus haut du fait qu'ils soient déclarés **static**, cela se fera désormais, seulement avec le paramètre p (Partie) passé à la méthode, et avec les getters on pourra leur accéder facilement. ➔ Respect de ne pas parler aux inconnus et diminuer le couplage entre les types de cases et autres classes.

d. Classe Sauv :

Cette classe Sauv comme mentionné dans les problèmes, on doit lui assigner la reponsabilté de sauvegarder une partie. On a préféré la considérer comme étant une pure fabrication c'est-à-dire une classe artificielle implémentant la méthode de sérialisation des objets. ➔ Cette classe sera donc cohésive et aura une méthode générique réutilisable.

e. PanelJoueur & Partie :

- La méthode de sauvegarde étant désormais implémentée dans la classe sauv, et ce n'est qu'en cliquant sur le bouton sauvegarder du PanelJoueur, que l'on fera appel à ce méthode ➔ Plus de cohésion et respect du principe SRP.
- PanelJoueur va être seulement rajouté dans la classe Fenetre (fenêtre principale du jeu).
- L'instanciation des dès par contre se fera désormais dans la classe **Partie**, en même temps que l'instanciation du joueur et plateau ➔ Créateur + Encapsulation. La méthode Lancer_De() quant à elle sera appelée à partir de PanelJoueur **(clic sur le bouton Lancer dés).**

f. Classe De :

- On ne va plus déclarer deux attributs valeurs (val1 et val 2) dans la classe De mais plutôt une seule valeur ; la méthode Lancer_De() s'occupera de lui affecter une valeur aléatoire comprise entre 1 et 6 et de la retourner. Notons que la classe Partie comme cité juste avant est son créateur, et donc dans notre cas on instanciera 2 objets de De ; et si on souhaiterait par la suite lancer 3 dès, on se contentera d'instancier un 3ème objet de type De seulement ➔ OCP respecté.

Voici une comparaison entre l'ancienne et la nouvelle implémentation :

```java
public class De implements Serializable
{
    protected static int val1=0,val2=0;
    public static int Lancer_De()
    {
        int i=0;
        /* recuperer la valeur du premier dé */
        i=Math.min(6, ((int) (Math.random()*6))+1);
        val1=i;
        /*recuperer la valeur du deuxiemme dé */
        i=Math.min(6, ((int) (Math.random()*6))+1);
        val2=i;
        /* retourner la somme des deux valeurs */
        return val1+val2;
    }
}
```

```java
public class De implements Serializable
{
    private int valeur=0;
    public int Lancer_De()
    {
        int i=0;
        /* recuperer la valeur du premier dé */
        valeur=Math.min(6, ((int) (Math.random()*6))+1);

        return valeur;//+val2;
    }
}
```

Avant : il y avait 2 attributs val1, val2 : tous les 2 sont **static**
La méthode est statique retournant la somme des 2 valeurs

Après : il y a un seul attribut valeur privé
La méthode n'est plus statique et retourne une valeur comprise entre 1 & 6

```java
int depla=De.Lancer_De();
label1.setIcon(des[De.val1-1]);
label2.setIcon(des[De.val2-1]);
```

```java
private De de1 = new De();
private De de2 = new De();

int val1 = de1.Lancer_De();
int val2 = de2.Lancer_De();
int depla= val1 + val2;
label1.setIcon(des[val1-1]);
label2.setIcon(des[val2-1]);
```

Avant : Appel de la fonction **static**

Après : Instanciation de deux objets De et appel de la fonction Lancer_De () chacune à part ;

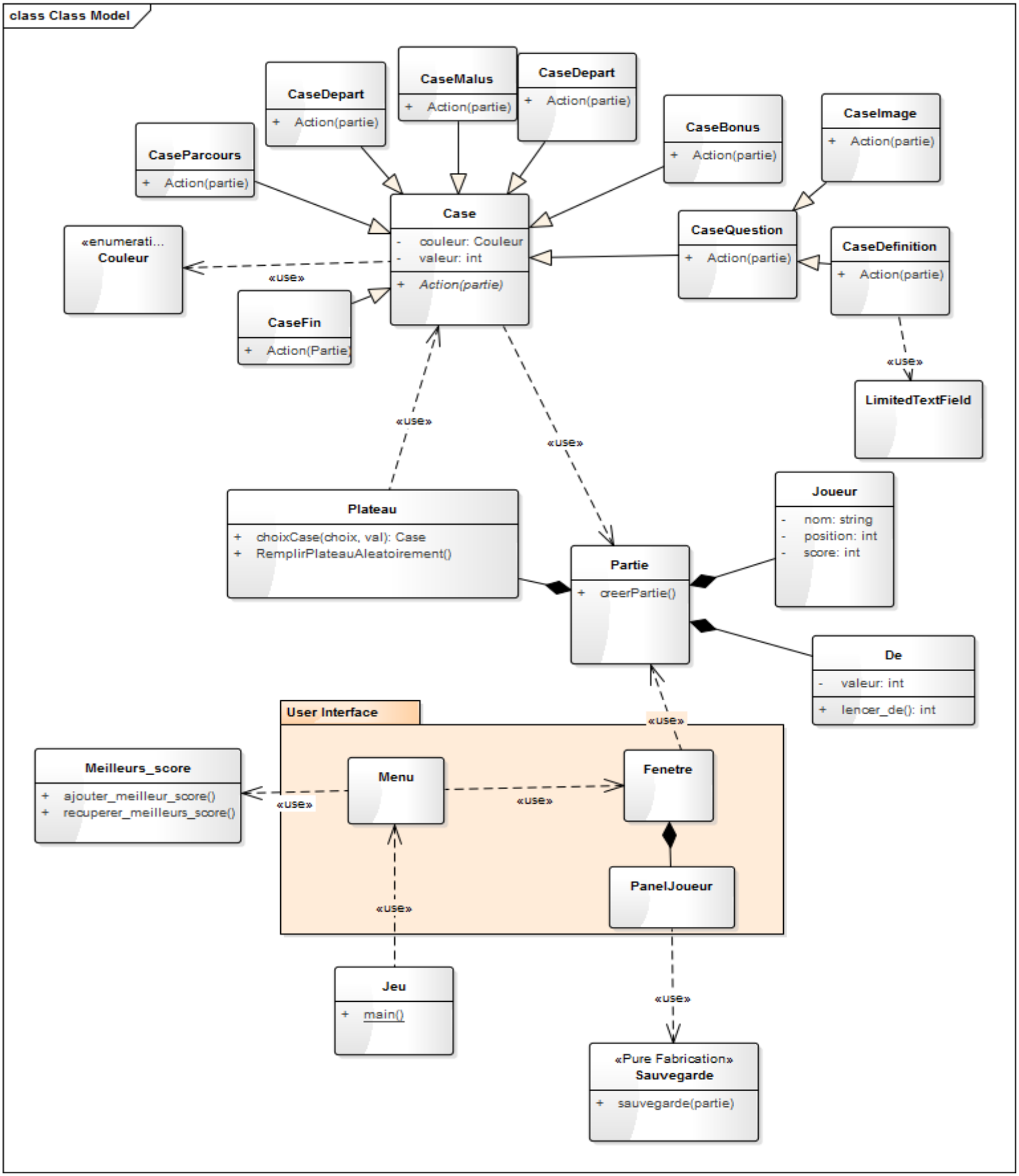
g. Classe Jeu :

Le problème de boucle infinie créée pour empêcher d'appeler Partie.jeu() et afficher la fenêtre principale sera détourné et le booléen **bool** sera supprimé .

NOTA :

- De manière globale nous essayerons de respecter au mieux les principes d'encapsulation, ne pas parler aux inconnus, et faible couplage en changeant la quasi-totalité des attributs de **protected static** en **private.**
- **La notion d'instanciation (patron créateur) existera vraiment.**

4.2. Nouveau diagramme de classe :



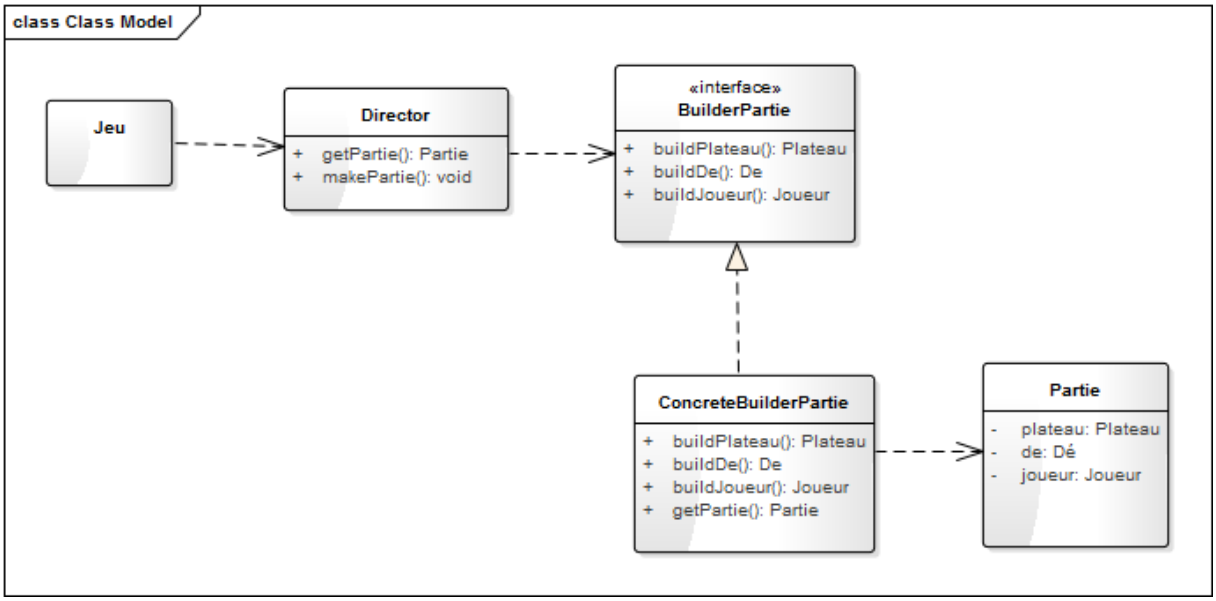
5. Patron GoF :

Même si ce n'est pas demandé dans l'énoncé, mais nous tenions à mettre l'accent sur quelques patrons du Gof qui pouvaient être appliqués dans le but d'améliorer la solution existante.

a. Builder:

Une Partie est composée d'un joueur, un plateau et un dé. Pour simplifier la création d'une Partie qui est un composant complexe, le patron Builder peut être utilisé.

Le diagramme suivant résume la situation :



b. Prototype :

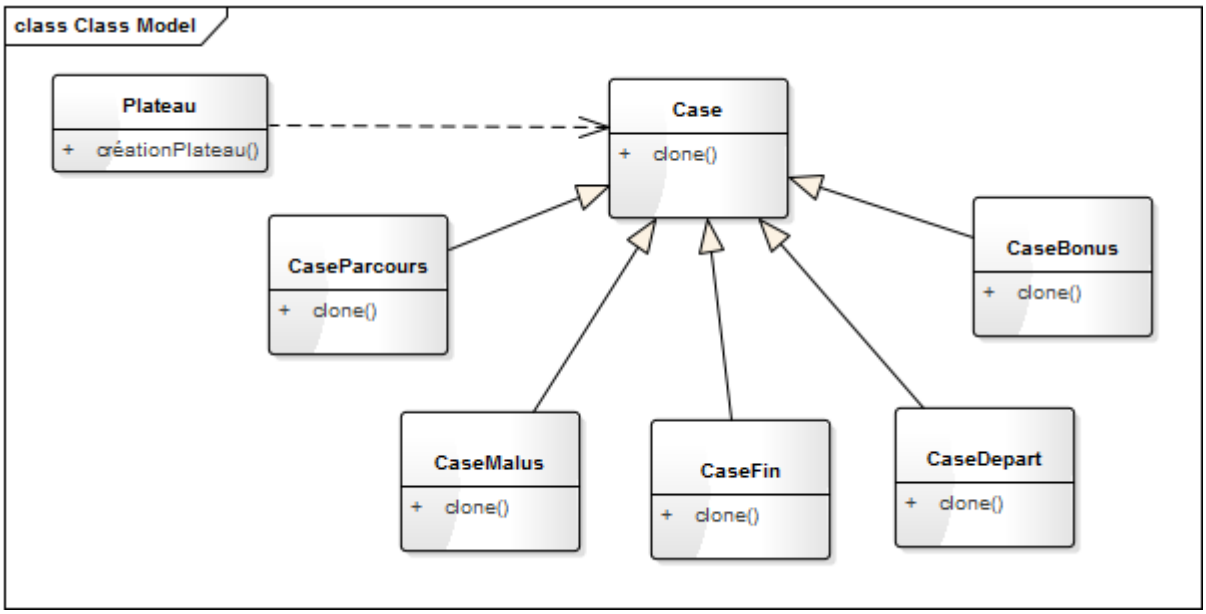
Le plateau est composé de 100 cases réparties en spirale, on distingue 7 types de case (case départ, bonus, malus, fin …).

Plutôt que de créer 5 instances de la classe CaseMalus, 5 autres de CaseBonus, il suffit d'instancier une seule case puis la dupliquer 4 autres fois.

Exemple :

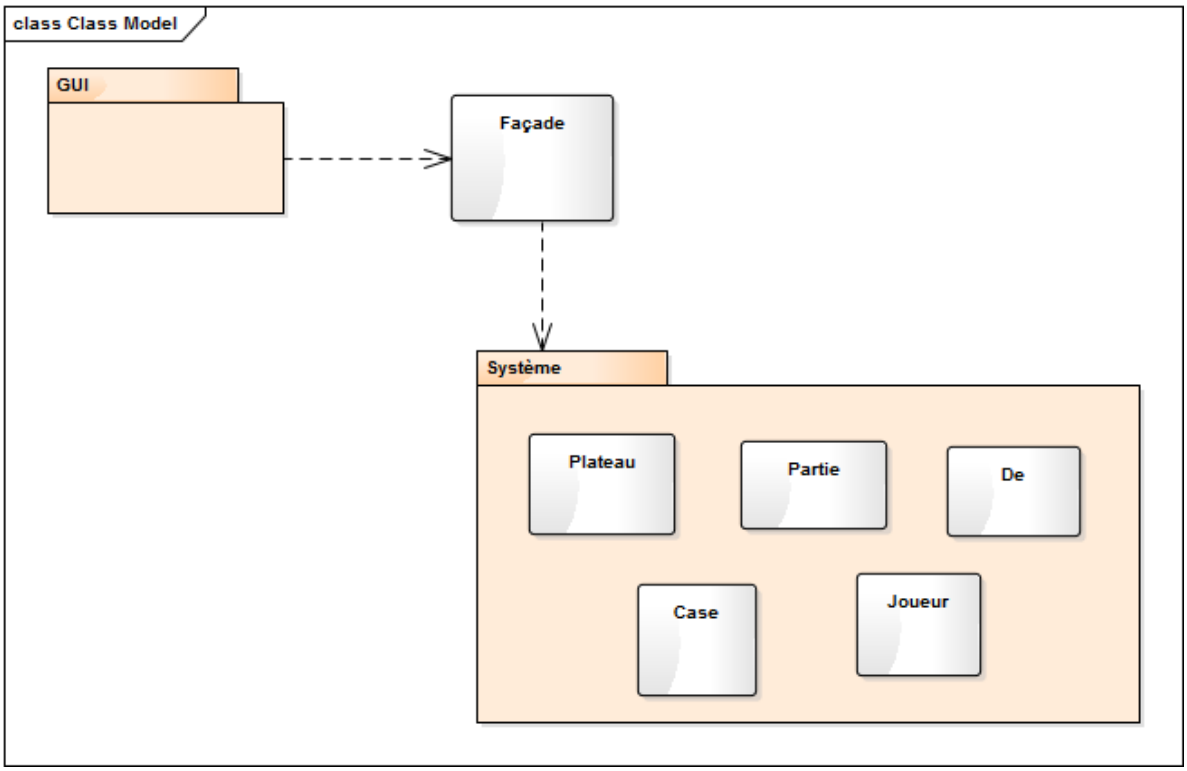
Le client (la classe Plateau), au lieu d'écrire du code invoquant directement l'opérateur "new" sur la classe CaseBonus, appellera la méthode clone() de cette classe.

Le diagramme si dessous en résume la situation.



c. Façade :

L'interface graphique, ayant uniquement besoin de la classe Partie, la mise en place du patron Façade facilite l'utilisation de la classe Partie par l'interface utilisateur.



d. Singleton :

Partie, étant la classe la plus importante dans le jeu, il n'est pas utile d'avoir plusieurs instances, ainsi l'utilisation de l'instance unique facilitera son accès de n'importe quelle autre classe. Il en est de même pour les classes **Plateau et Jeu.**