# Programmation Objet Avancée
# Examen UML

pakpake

4 Mai 2020

**Résumé**

Ce document est la synthèse du contrôle d'UML. Il comprend systématiquement les diagrammes demandés, tous réalisés avec le logiciel PlantUml. Les codes permettant de reproduire les figures sont donnés après chaque figure. Les codes sont exécutables en ligne sur le serveur de PlantUml à l'adresse suivante : .

## Table des matières

# Table des figures



# Liste des codes

# 1 Question 1

On distingue 3 cas d'utilisation :
— Jouer une partie de démineur
— Configurer le jeu
— Consulter l'aide en ligne

L'objectif principal est de jouer une partie. Les 2 autres cas d'utilisation sont secondaires. Ils auront donc la propriété «*extend*». On a donc le diagramme de cas d'utilisation représenté dans la figure 1 :

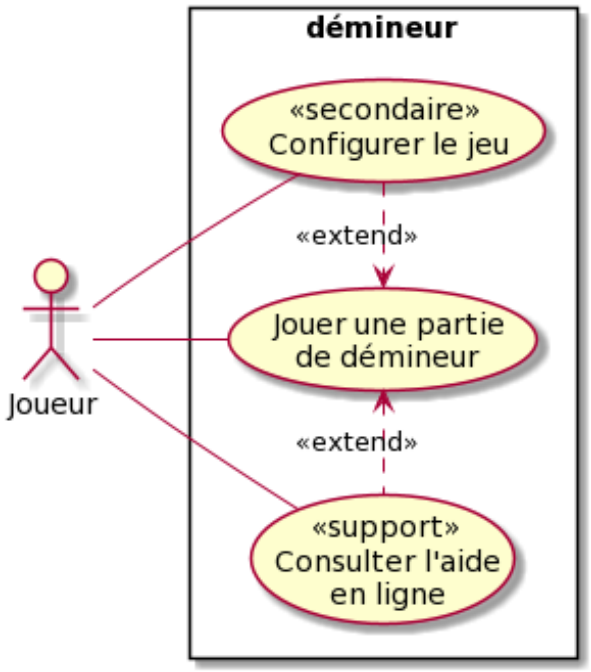


FIGURE 1 – Diagramme de cas d'utilisation pour le jeu de démineur

Dont voici le code réalisé grâce à PlantUml donné dans le listing 1

```
@startuml
left to right direction
skinparam packageStyle rectangle

actor Joueur as player

rectangle démineur {
    (<<secondaire>> \n Configurer le jeu) as config
    (Jouer une partie \n de démineur) as play
    (<<support>> \n Consulter l'aide \n en ligne) as help

    player -- play
    player -- config
    player -- help
    config .> play : <<extend>>
    play <. help : <<extend>>
}
@enduml
```

Listing 1 – Code du diagramme de cas d'utilisation

# 2 Question 2

## 2.1 a)

Le but du jeu est de découvrir/démasquer ou marquer des cases sans tomber sur des mines. Le joueur ne va donc faire que cela jusqu'à ce qu'il gagne dans le meilleur des cas. S'il gagne, il aura donc fait un certain temps qui sera enregistré dans les meilleurs scores. Tout ceci peut être fait via une boucle ("loop") grâce à laquelle le joueur à 2 choix réalisables dans n'importe quel sens grâce au bloc "alt". Ce diagramme est présenté dans la figure 2.

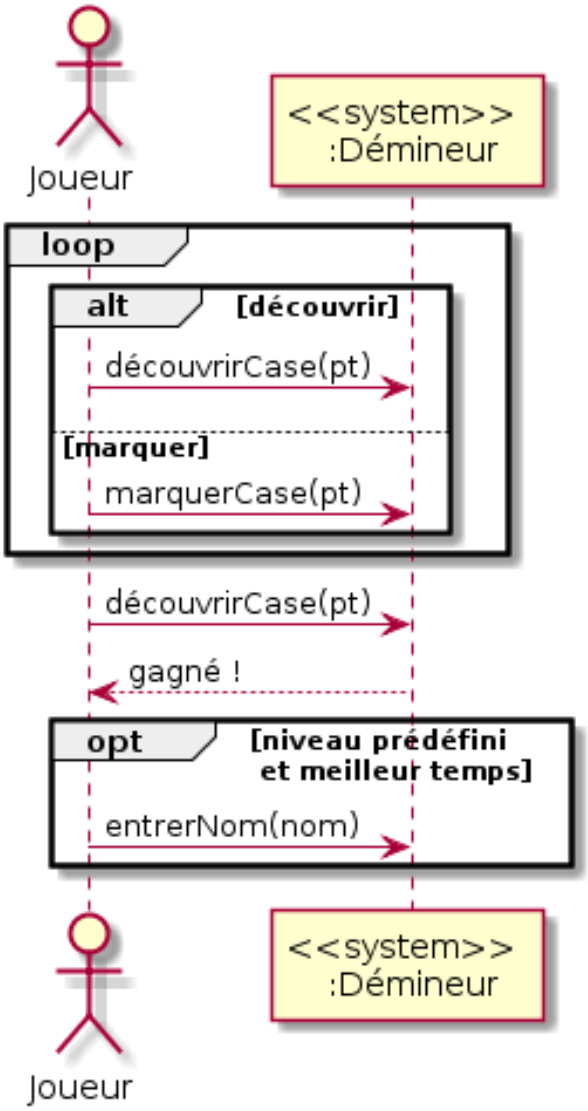


FIGURE 2 – Diagramme de séquence

Ce diagramme n'est pas complet, il manque le cas où le joueur perd, et avant cela la configuration du jeu, comme par exemple le niveau de difficulté.

Le code de la figure 2 en PLantUml est donné dans le listing 2.

```
@startuml
!pragma teoz true
actor Joueur


loop
  alt découvrir
    Joueur -> "<<system>> \n :Démineur" : découvrirCase(pt)
  else marquer
    Joueur -> "<<system>> \n :Démineur" : marquerCase(pt)
  end
end

Joueur -> "<<system>> \n :Démineur" : découvrirCase(pt)
"<<system>> \n :Démineur" --> Joueur : gagné !

opt niveau prédéfini \n et meilleur temps
    Joueur -> "<<system>> \n :Démineur" : entrerNom(nom)
@endumll
```

Listing 2 – Code du diagramme de séquence

## 2.2 b)

On ajoute à notre diagramme de séquence l'option permettant la configuration du jeu. Celle-ci est en amont de la séquence de jeu. Elle se place donc au dessus dans le diagramme de séquence.

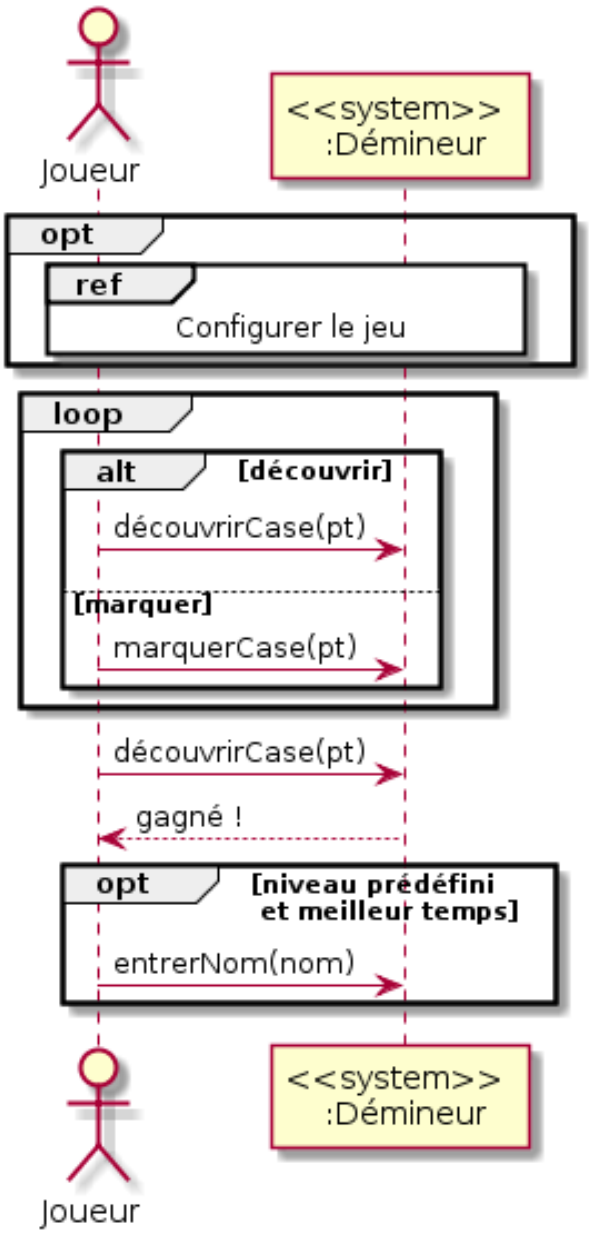


FIGURE 3 – Diagramme de séquence avec configuration

Le code de la figure 3 en PLantUml est donné dans le listing 3.

```
@startuml
!pragma teoz true
actor Joueur
participant "<<system>> \n :Démineur" as sys

opt
    ref over Joueur, sys : Configurer le jeu
end
loop
  alt découvrir
    Joueur -> sys : découvrirCase(pt)
  else marquer
    Joueur -> sys : marquerCase(pt)
  end
end

Joueur -> sys : découvrirCase(pt)
sys --> Joueur : gagné !

opt niveau prédéfini \n et meilleur temps
    Joueur -> sys : entrerNom(nom)
@enduml
```

Listing 3 – Code du diagramme de séquence avec configuration

## 2.3 c)

Et enfin, nous avons le diagramme final, qui prend bien en compte, le fait que le joueur puisse gagner ou perdre. Ceci est représenté avec l'opérateur "break".

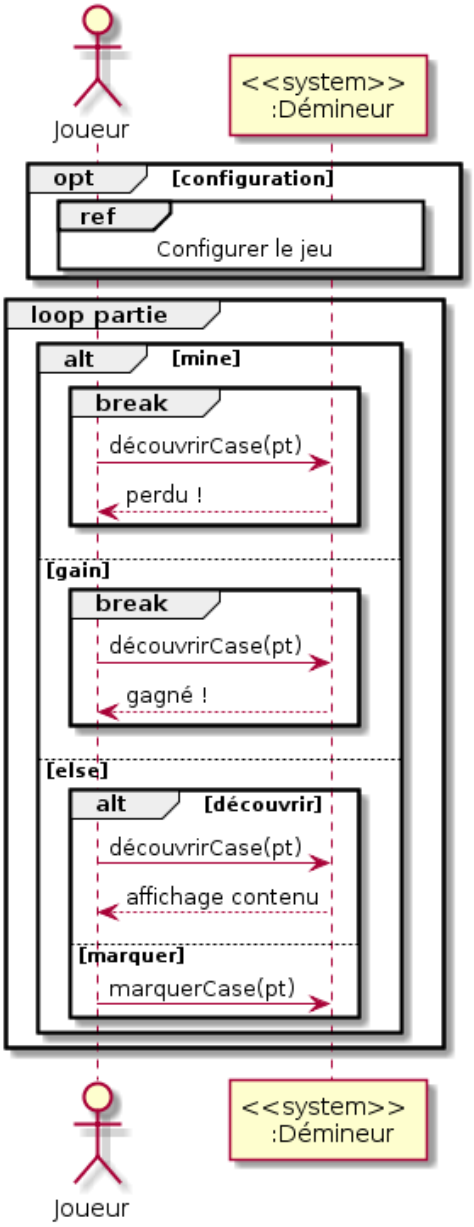


FIGURE 4 – Diagramme de séquence complet

Le code de la figure 4 en PLantUml est donné dans le listing 4.

```
@startuml
!pragma teoz true
actor Joueur
participant "<<system>> \n :Démineur" as sys

opt configuration
    ref over Joueur, sys : Configurer le jeu
end

group loop partie
    alt mine
        break
            Joueur -> sys : découvrirCase(pt)
            sys --> Joueur : perdu !
        end
    else gain
        break
            Joueur -> sys : découvrirCase(pt)
            sys --> Joueur : gagné !
        end
    else else
        alt découvrir
            Joueur -> sys : découvrirCase(pt)
            sys --> Joueur : affichage contenu
        else marquer
            Joueur -> sys : marquerCase(pt)
    end
end
@enduml
```

Listing 4 – Code du diagramme de séquence complet

# 3 Question 3

Le diagramme de classe d'analyse (voir figure 5) nous permet de représenter le jeu avec une configuration complète. C'est-à-dire le nombre minimal de mines, le nombre de mines restantes, le niveau de jeu, et le résultat. Mais aussi, la taille du plateau, le nom du joueur, le temps passé depuis le début de la partie, les coordonnées de la case et des ses voisines, et si la case est minée ou non.

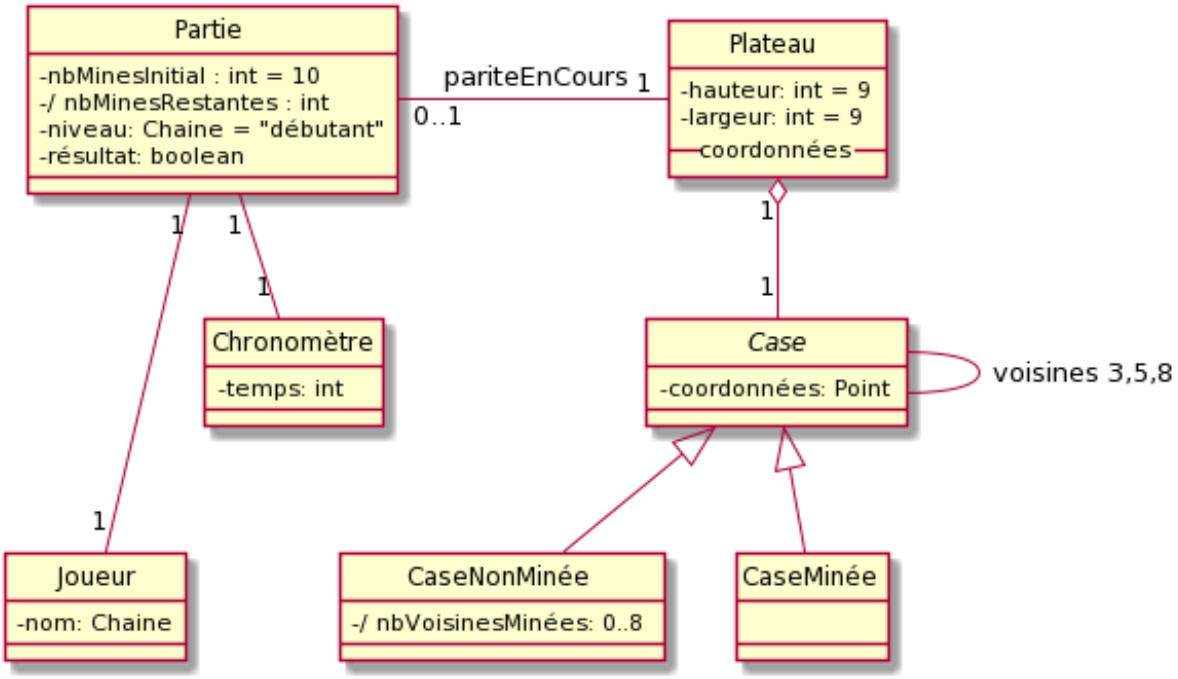


FIGURE 5 – Diagramme de classe d'analyse

Le code de la figure 5 en PLantUml est donné dans le listing 5.

```
@startuml
hide circle
skinparam classAttributeIconSize 0

class Partie {
    - nbMinesInitial : int = 10
    -/ nbMinesRestantes : int
    - niveau: Chaine = "débutant"
    - résultat: boolean
}

class Joueur {
    - nom: Chaine
}

class Chronomètre {
    - temps: int
}

class Plateau {
    - hauteur: int = 9
    - largeur: int = 9
    __coordonnées__
}

class "//Case//" {
    - coordonnées: Point
}

CaseMinée : \t

class CaseNonMinée {
    -/ nbVoisinesMinées: 0..8
}

Partie "1" --- "1" Joueur
Partie "1" -- "1" Chronomètre
Partie "0..1" - "1" Plateau : "\n pariteEnCours"
"Plateau" "1" o-down- "1" "//Case//"
"//Case//" - "//Case//" : "voisines 3,5,8"
CaseMinée -up-|> "//Case//"
CaseNonMinée -up-|> "//Case//"
@enduml
```

Listing 5 – Code du diagramme de classe d'analyse

Le diagramme d'état de la classe Case, nous permet quant à lui, de connaitre l'état de la case après l'action du joueur. Ce diagramme est représenté dans la figure 6.

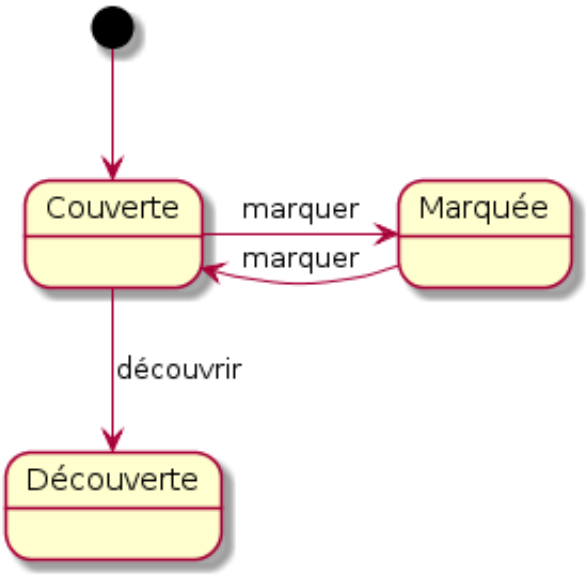


FIGURE 6 – Diagramme de classe d'analyse

Le code de la figure 6 en PLantUml est donné dans le listing 6.

```
@startuml
[*] --> Couverte
Couverte -right-> Marquée : marquer
Marquée -left-> Couverte : marquer
Couverte -down-> Découverte : découvrir
@enduml
```

Listing 6 – Code du diagramme d'état de la classe Case

# 4 Question 4

Nous devons donc réaliser un diagramme de séquence, qui permet de représenter dynamiquement les échanges entre les différents objets et acteurs en fonction du temps. Ici, ce diagramme (voir figure 7) permet de représenter l'opération système *marquerCase(pt)*, ce qui nous permet entre autre d'avoir une gestion presque complète des actions faites sur chaque case.

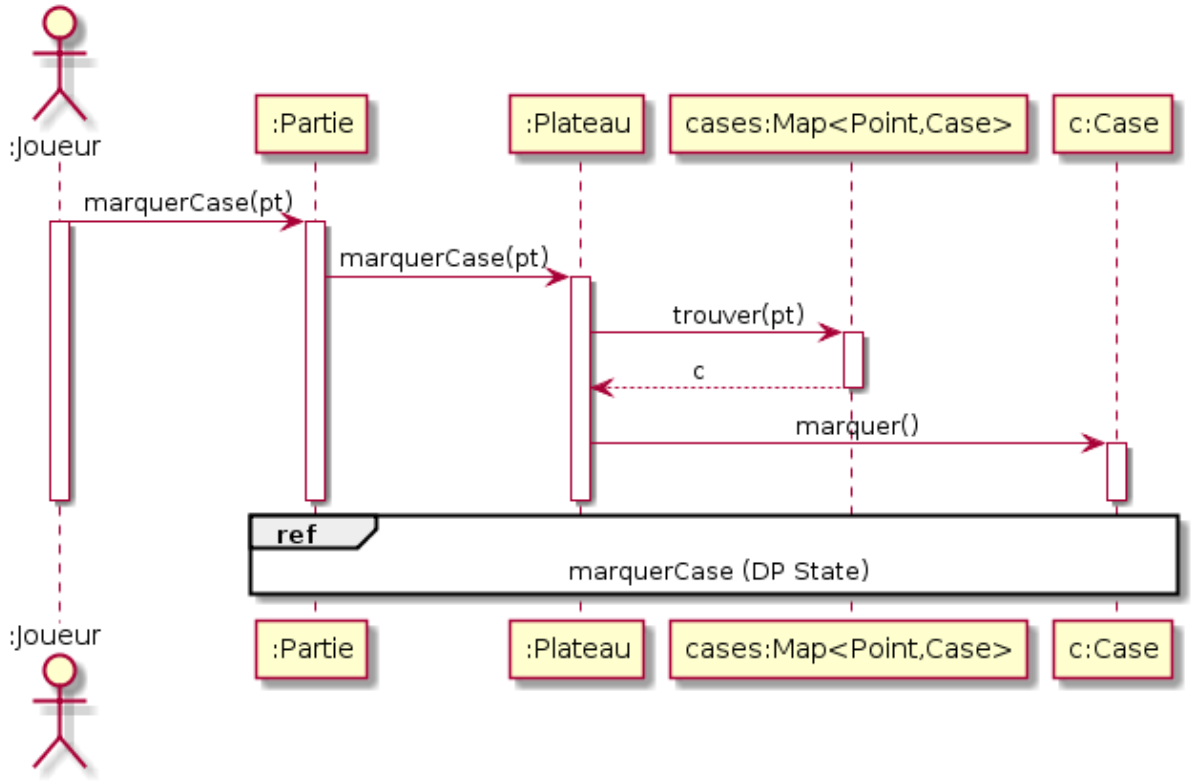


FIGURE 7 – Diagramme de séquence pour l'opération système *marquerCase(pt)*

Le code de la figure 7 en PLantUml est donné dans le listing **??**.

```
@startuml
actor ":Joueur" as player
participant ":Partie" as game
participant ":Plateau" as plateau
participant "cases:Map<Point,Case>" as point
participant "c:Case" as case

player -> game : marquerCase(pt)
activate player
activate game
game -> plateau : marquerCase(pt)
activate plateau
plateau -> point : \t trouver(pt)
activate point
point --> plateau : \t c
deactivate point
plateau -> case : \t \t \t marquer()
activate case


deactivate case
deactivate plateau
deactivate game
deactivate player

ref over game, case : marquerCase (DP State)
@enduml
```

Listing 7 – Code du diagramme de séquence pour l'opération *marquerCase(pt)*

Mais il faut l'améliorer, c'est pourquoi il y a un 2ème diagramme qui permet donc la gestion complète qui est faite à chaque action sur chaque case. Par exemple, décrémenter ou incrémenter le nombre de mines restante au cours de la partie, en fonction de l'état de la case, si elle est découverte, couverte ou bien marquée.

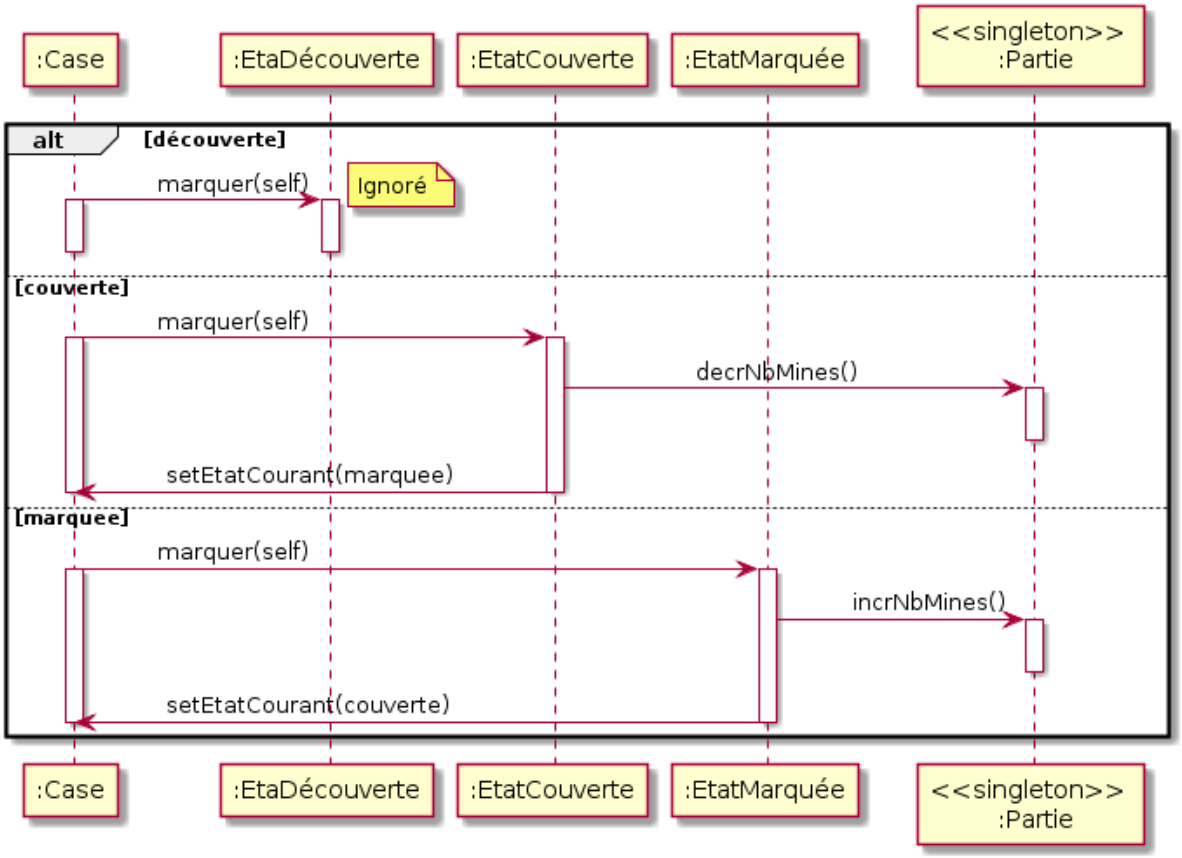


FIGURE 8 – Diagramme de séquence

Le code de la figure 8 en PLantUml est donné dans le listing 8.

```
@startuml
participant ":Case" as case
participant ":EtaDécouverte" as discover
participant ":EtatCouverte" as cover
participant ":EtatMarquée" as marked
participant "<<singleton>> \n :Partie" as party

alt découverte
    case -> discover : \t marquer(self)
    activate case
    activate discover
    deactivate case
    deactivate discover
    note right : Ignoré
else couverte
    case -> cover : \t marquer(self)
    activate case
    activate cover
    cover -> party : \t \t decrNbMines()
    activate party
    deactivate party
    cover -> case : \t setEtatCourant(marquee)
    deactivate cover
    deactivate case
else marquee
    case -> marked : \t marquer(self)
    activate case
    activate marked
    marked -> party : \t incrNbMines()
    activate party
    deactivate party
    marked -> case : \t setEtatCourant(couverte)
    deactivate marked
    deactivate case
end
@enduml
```

Listing 8 – Code du diagramme de séquence

# 5 Question 5

Nous avons ici 3 diagrammes.

On se replace maintenant au niveau plus général avec le joueur, la partie, le plateau, la collection des cases et une instance particulière de case. Dans ce diagramme “Partie” délègue à “Plateau” l’action “découvrirCase” qui en s’adressant à la collection de cases va pouvoir savoir si une case est découverte. L’action *découvrirCase* est maintenant un groupe “ref” (*Design Pattern State*).

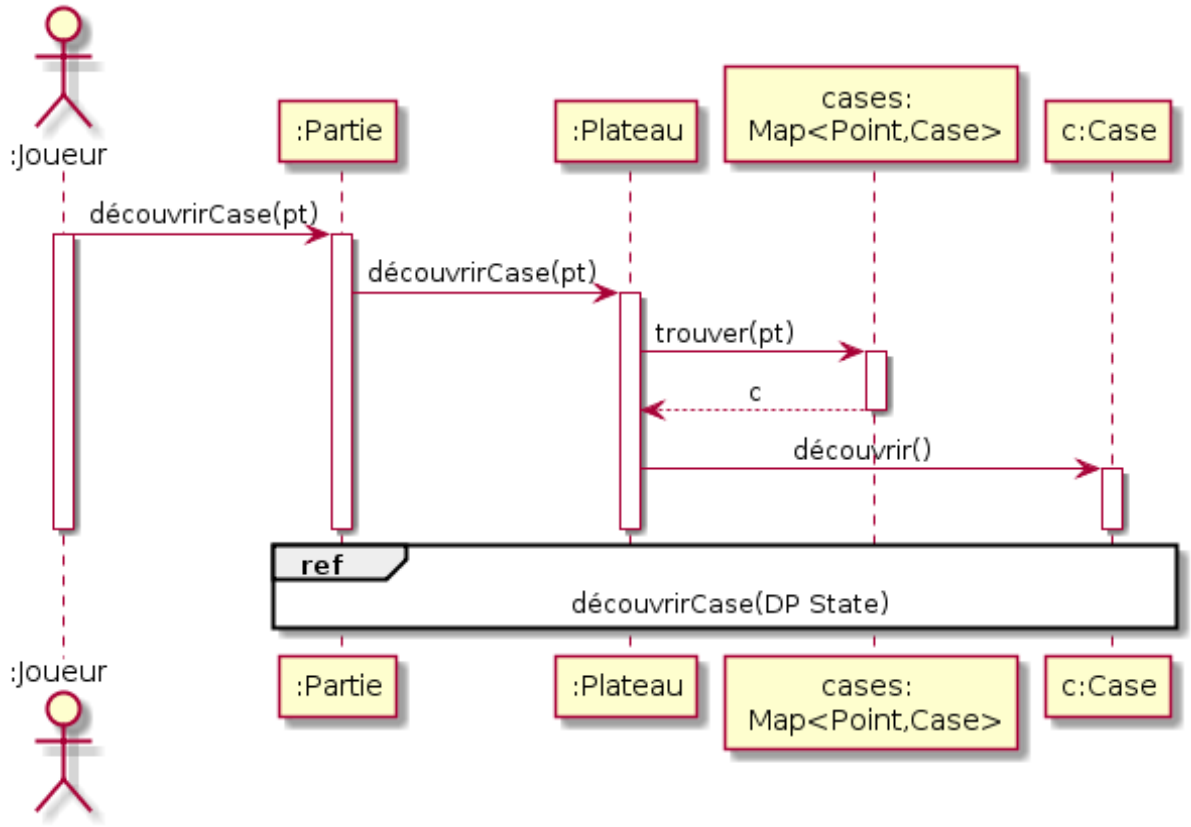


FIGURE 9 – Diagramme de séquence de conception pour *découvrirCase(pt)*

Le code de la figure 9 en PLantUml est donné dans le listing 9.

```
@startuml
actor ":Joueur" as player
participant ":Partie" as party
participant ":Plateau" as tray
participant "cases: \n Map<Point,Case>" as point
participant "c:Case" as case

player -> party : découvrirCase(pt)
activate player
activate party
party -> tray : découvrirCase(pt)
activate tray
tray -> point : trouver(pt)
activate point
point --> tray : \t c
deactivate point
tray -> case : \t \t découvrir()
activate case
deactivate case
deactivate tray
deactivate party
deactivate player

ref over party,case : découvrirCase(DP State)
@enduml
```

Listing 9 – Code du diagramme de la séquence de conception pour *découvrirCase(pt)*

Le diagramme de la figure 10 est l'implémentation du groupe référence du diagramme de la figure 9. La situation ici a plus de conséquences car le fait de dévoiler une case dépend de son état (case avec une mine ou non) et cela pourrait soit interrompre le jeu, soit permettre de continuer la partie. Ainsi, l'opération "dévoiler" a plusieurs modes de fonctionnement qui dépendent de la situation du jeu. Le groupe référence *dévoiler* est donc une opération polymorphe. Elle sera détaillée dans le diagramme de la figure 11.

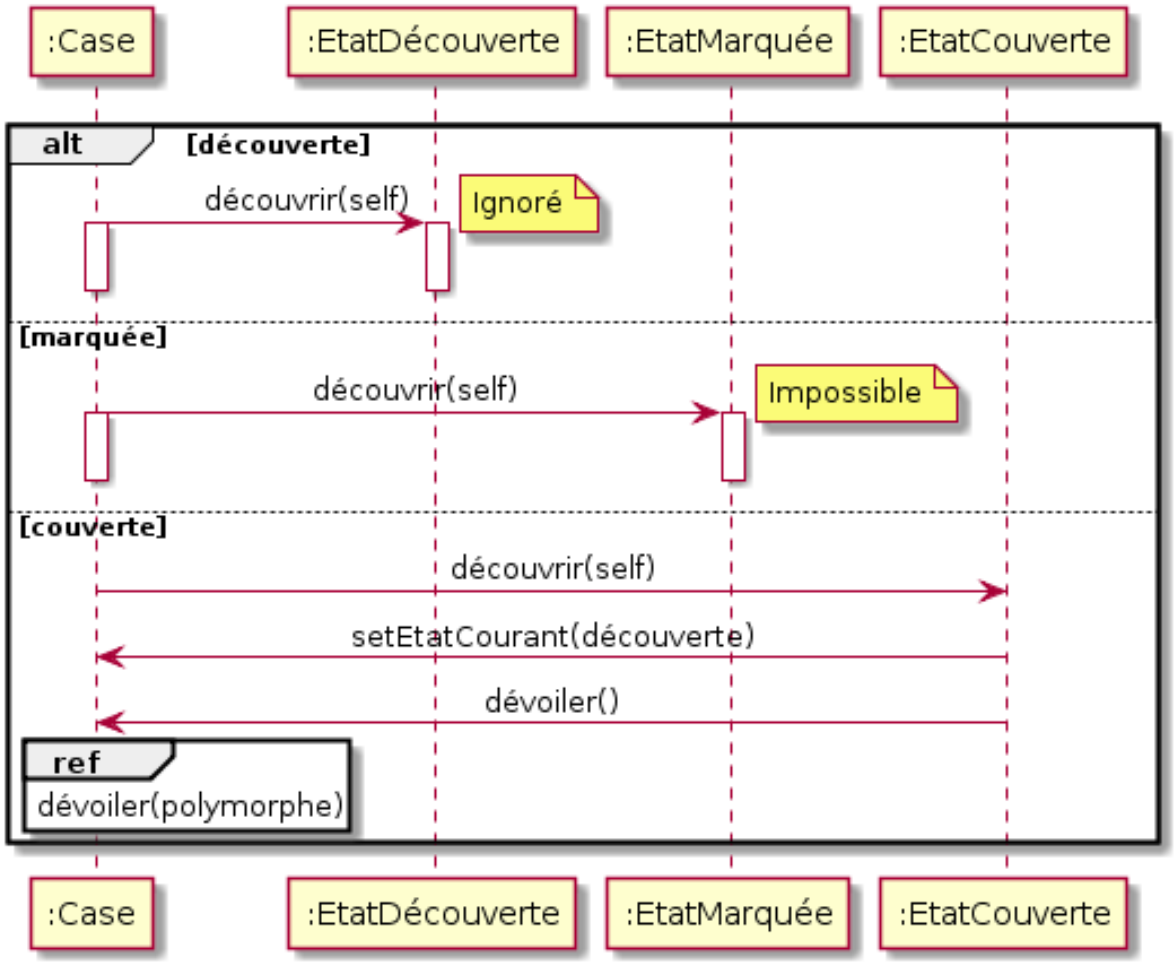


FIGURE 10 – Diagramme de séquence

Le code de la figure 10 en PLantUml est donné dans le listing 10.

```
@startuml
skinparam sequenceMessageAlign center

participant ":Case" as case
participant ":EtatDécouverte" as discover
participant ":EtatMarquée" as mark
participant ":EtatCouverte" as cover

alt découverte
    case -> discover : \t découvrir(self)
    activate case
    activate discover
    deactivate case
    deactivate discover
    note right : Ignoré

else marquée
    case -> mark : découvrir(self)
    activate case
    activate mark
    deactivate case
    deactivate mark
    note right : Impossible
    deactivate case
    deactivate mark
else couverte
    case -> cover : découvrir(self)
    cover -> case : setEtatCourant(découverte)
    cover -> case : dévoiler()
    ref over case : dévoiler(polymorphe)
end
@enduml
```

Listing 10 – Code du diagramme de séquence

Ce 3eme diagramme, détaille l'implémentation de l'opération *dévoiler*. Si la case contient une mine, alors on va enchaîner les actions suivantes :

— arrêter le chronomètre

— découvrir toutes les mines non trouvées

— marquer avec un "X" les cases faussement marquées comme contenant une mine

Si la case ne contient pas de mine et est numérotée, alors il faut vérifier si la partie est gagnée ou non.

La dernière alternative est une case vide et dans ce cas, on lance l'opération *découvrir* à toutes les cases voisines.

La figure 11 donne le diagramme de conception de l'opération polymorphe *dévoiler*.

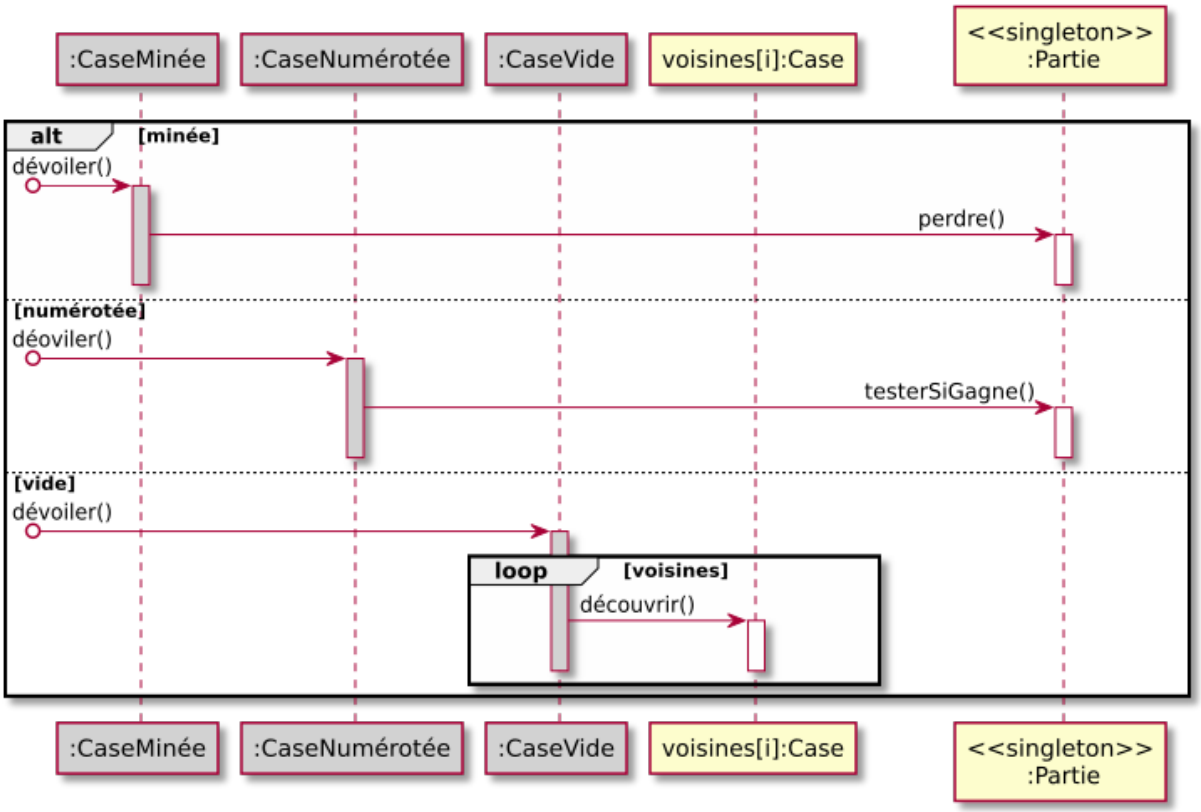


FIGURE 11 – Diagramme de séquence de l'opération polymorphe *dévoiler*

Le code de la figure 11 en PLantUml est donné dans le listing 11.

```
@startuml
participant ":CaseMinée" as mine #lightgrey
participant ":CaseNumérotée" as num #lightgrey
participant ":CaseVide" as vide #lightgrey
participant "voisines[i]:Case" as case
participant "<<singleton>>\n :Partie" as party

alt minée
    [o-> mine ++ #lightgrey : dévoiler()
    mine -> party ++ : \t \t \t \t \t \t \t \t \t \t \t \t perdre()
    party --
    mine --
else numérotée
    [o-> num ++ #lightgrey : déoviler()
    num -> party ++ : \t \t \t \t \t \t \t \t \t testerSiGagne()
    party --
    num --
else vide
    [o-> vide ++ #lightgrey : dévoiler()
    loop voisines
        vide -> case ++ : découvrir()
        case --
        vide --
    end
end
@enduml
```

Listing 11 – Code du diagramme de séquence de l'opération polymorphe *dévoiler*

# 6 Question 6

En combinant toutes les opérations des diagrammes de séquence précédents dans les bonnes classes, on peut construire ce dernier diagramme de classe de conception du jeu de démineur. On peut apercevoir l'héritage de *CaseMinée*, *CaseVide*, *CaseNumérotée* de *Case* ainsi que l'héritage de *EtatDécouverte*, *EtatMarquée*, *EtatCouverte* de *EtatCase*.

Le diagramme est représenté dans la figure 12.

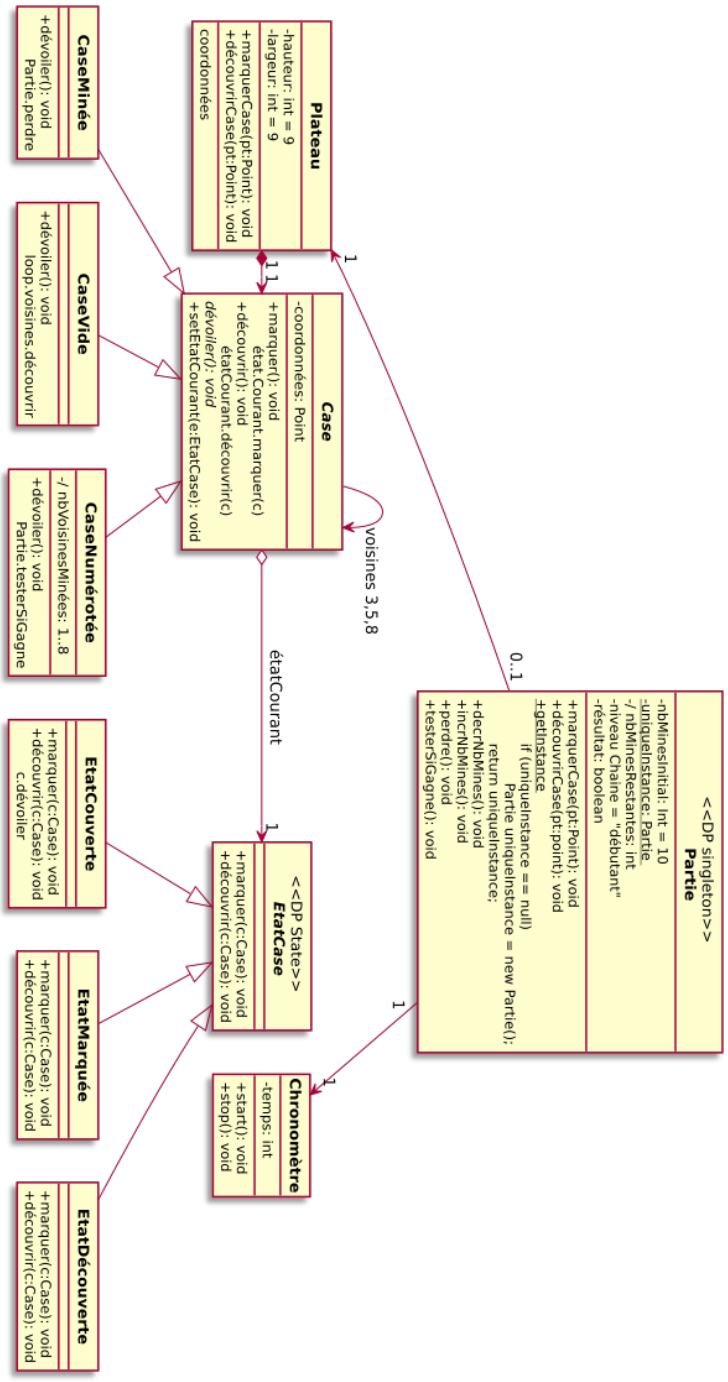


FIGURE 12 – Diagramme de classe de conception du jeu complet

Le code de la figure 12 en PLantUml est donné dans le listing 12.

```
@startuml
hide circle
skinparam classAttributeIconSize 0

class "<<DP singleton>> \n **Partie**" as party {
    - nbMinesInitial: Int = 10
    {static} - uniqueInstance: Partie
    -/ nbMinesRestantes: int
    - niveau Chaine = "débutant"
    - résultat: boolean

    --
    + marquerCase(pt:Point): void
    + découvrirCase(pt:point): void
    {static} + getInstance
    \t if (uniqueInstance == null)
    \t \t Partie uniqueInstance = new Partie();
    \t return uniqueInstance;
    + decrNbMines(): void
    + incrNbMines(): void
    + perdre(): void
    + testerSiGagne(): void
}


class "**Plateau**" as tray {
    - hauteur: int = 9
    - largeur: int = 9

    --
    + marquerCase(pt:Point): void
    +découvrirCase(pt:Point): void

    --
    coordonnées
}

class "**Chronomètre**" as chrono {
    - temps: int

    --
    + start(): void
    + stop(): void
}

class "**//Case//**" as case {
    - coordonnées: Point

    --
    + marquer(): void
    \t état.Courant.marquer(c)
```

```
47      + découvrir(): void
48      \t étatCourant.découvrir(c)
49      {abstract} dévoiler(): void
50      + setEtatCourant(e:EtatCase): void
51  }
52
53  class "<<DP State>> \n **//EtatCase//**" as state {
54      + marquer(c:Case): void
55      + découvrir(c:Case): void
56  }
57
58  class "**CaseMinée**" as mine {
59      --
60      + dévoiler(): void
61      \t Partie.perdre
62  }
63
64  class "**CaseVide**" as vide {
65      --
66      + dévoiler(): void
67      \t loop.voisines.découvrir
68  }
69
70  class "**CaseNumérotée**" as num {
71      -/ nbVoisinesMinées: 1..8
72      --
73      + dévoiler(): void
74      \t Partie.testerSiGagne
75  }
76
77  class "**EtatCouverte**" as cover {
78      --
79      + marquer(c:Case): void
80      + découvrir(c:Case): void
81      \t c.dévoiler
82  }
83
84  class "**EtatMarquée**" as mark {
85      + marquer(c:Case): void
86      + découvrir(c:Case): void
87  }
88
89  class "**EtatDécouverte**" as discover {
90      + marquer(c:Case): void
91      + découvrir(c:Case): void
92  }
93
```

```
94
95  party "0..1" --> "1" tray
96  party "1" -down-> "1" chrono
97  'case "1" <-down* "1" tray
98  tray "1" *-down> "1" case
99  case o-right-> "1" state : -étatCourant
100 case -> case : \n \n voisines 3,5,8
101
102 mine -up-|> case
103 vide -up-|> case
104 num -up-|> case
105
106 cover -up-|> state
107 mark -up-|> state
108 discover -up-|> state
109 @enduml
```

Listing 12 – Code du diagramme de classe de conception du jeu complet