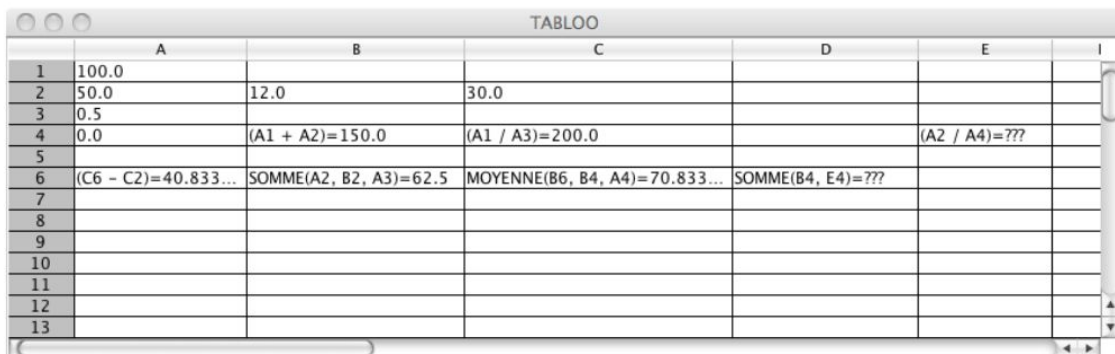


Projet de PPO

TABLOO : tableur orienté objet



	A	B	C	D	E	F
1	100.0					
2	50.0	12.0	30.0			
3	0.5					
4	0.0	(A1 + A2)=150.0	(A1 / A3)=200.0		(A2 / A4)=???	
5						
6	(C6 - C2)=40.833...	SOMME(A2, B2, A3)=62.5	MOYENNE(B6, B4, A4)=70.833...	SOMME(B4, E4)=???		
7						
8						
9						
10						
11						
12						
13						

Tuteur : Bernard Carré
Maxime Sauvage
Jordan Laplace

Sommaire

Introduction

1. Analyse/Conception du cahier des charges du noyau

1.1. Classe Grille

1.2. Classe Case

1.3 Hiérarchie de classe Formule

2. Schéma UML

2.1. UML créé à partir de Modelio

2.2 Explication des ajouts

3. Choix de SD

3.1. Justification du choix des structures de données

3.2 Déclaration dans les classes correspondantes

4. Principaux traitements

4.1. Evaluation des formules

4.2. Affectation/modification d'une case par une valeur

4.3. Affectation/modification d'une case par une formule

4.4. Accès à une case de la grille par ses coordonnées

Conclusion

Introduction

Dans le cadre du module PPO, nous sommes amenés à réaliser un noyau de représentation et de calcul pour grilles de tableurs, exploitable plus généralement pour faire des formulaires en lignes ou des “calculettes” en mode console.

Les cases d’une grille de calculs contiennent soit des valeurs fournies par l’utilisateur, soit des résultats de formules faisant référence à d’autres cases. Le jeu de données permises et le jeu de formules sont riches, cependant nous traiterons seulement le type de donnée double et un jeu de formule limité à la moyenne et la somme. Notre application doit cependant demeurer extensible.

Ce rapport aura pour but de présenter la démarche d’analyse et de conception que nous avons mené durant les deux premières séances, nous avons à disposition le cahier des charges de l’application, un schéma UML de départ incomplet présentant le diagramme de classe de l’application Java.

Dans ce compte rendu nous présenterons d’abord l’étude du cahier des charges qui nous a permis de savoir ce qui devait être fait et ce que nous devons ajouter comme méthode en plus de celles qui nous ont été suggéré pour atteindre le résultat demandé. Ensuite, nous nous pencherons sur le schéma UML que nous avons complété avec les méthodes et instances supplémentaires. Nous expliciterons également le choix de la structure de données. Enfin nous détaillerons la conception des principaux traitements en pseudo-code Java.

1. Analyse/conception générale du cahier des charges du noyau

1.1. Classe Grille

La grille de calcul sera un objet de classe *Grille* est en relation d'instance avec la classe *Case*, nous utiliserons une structure de données adaptée afin de stocker les objets *Case* que nous développerons en détail dans la partie 3 du rapport.

1.1. Classe Case

Les cases de la grille seront des objets de classe *Case*, elle aura comme attributs :

- *colonne* de type *String*
- *ligne* de type *int*
- *valeur* de type *double*

Elle aura comme méthodes :

- *valeur()*
- *fixerValeur(double x)*
- *setFormule(Formule f)*

La classe *Case* est en relation d'instance avec la classe *Formule*, en effet elle possède l'attribut *formule* qui peut être instanciée ou non via la méthode *setFormule(Formule f)*.

Lors de l'utilisation de la méthode *void setFormule(Formule)* les valeurs des cases qui en dépendent devront être automatiquement mises à jour. Or dans le schéma UML de départ, il n'y a pas de relation de dépendance entre les cases, nous pensons donc à en créer une permettant de lier une case aux cases qui ont besoin d'elle.

Ainsi, grâce à cette relation de dépendance entre les cases nous pouvons détecter les cycles direct et indirect. Nous déclencherons alors des erreurs que nous aurons créées pour indiquer le problème à l'utilisateur à l'utilisateur.

Pour obtenir le "contenu" et le "contenu développé" d'une case, nous ajouterons des méthodes en plus dans la classe *Case*, qui appelleront des méthodes permettant de mettre sous forme de chaînes de caractères le contenu de l'instance *formule* qui sera implémenté dans la classe *Formule*.

1.3. Hiérarchie de classe Formule

Les formules sont représentées par une hiérarchie de classes de racine abstraite *Formule* parmi celles ci on distingue les opérations binaires (i.e. la classe *OperationBinaire* : +, -, *, /) qui ont deux arguments *droite et gauche* qui instancie la classe *Case* et les fonctions numériques à *n* arguments (i.e. la classe *Fonction* : Somme, Moyenne). Elles auront comme méthode commune double *eval()*.

Le traitement des langues du contenu des cases sera effectué dans les méthodes *toString()* et *ToStringDev()* des formules. Ceci en fonction de la variable global comptenu dans la classe *Grille*.

C'est aussi dans via les méthodes *toString()* et *ToStringDev()* que nous obtiendrons en partie le contenu et contenu developpe des cases.

Nous instancierons des classes *Exception* afin de nommer des erreurs tel que que la division par 0.

2. Schéma UML

2.1. UML créé à partir de Modelio

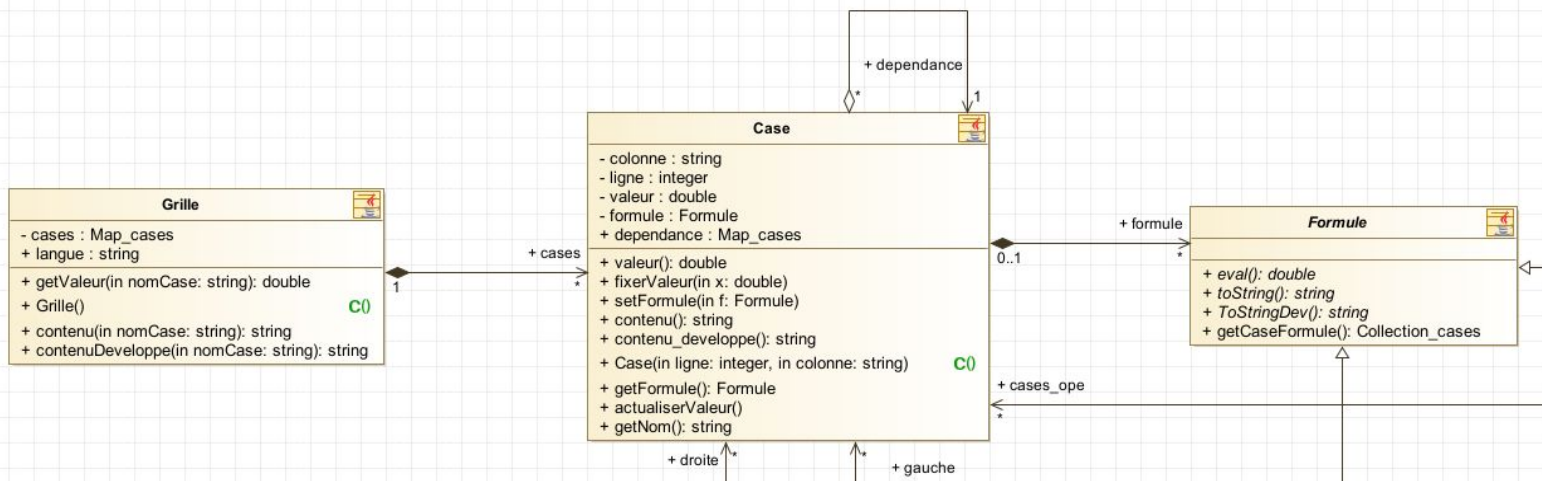


Figure 1. Relation entre les classes

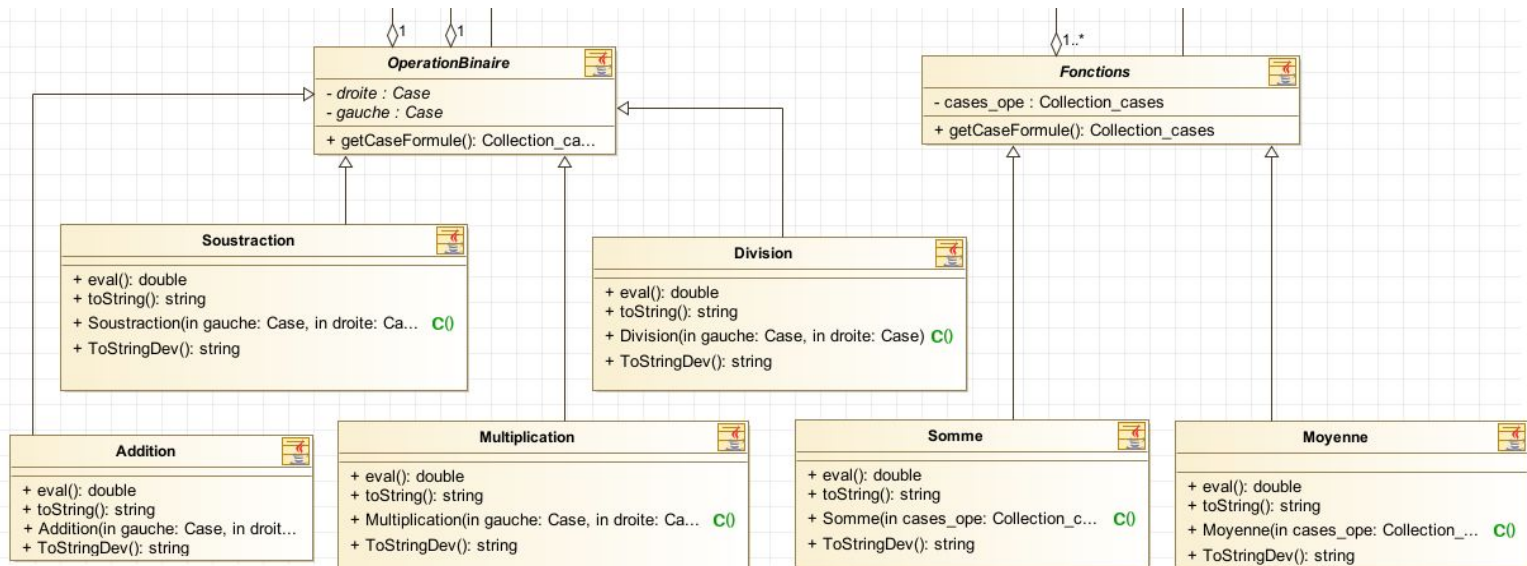


Figure 2. Hiérarchie de classes de racine Formule

2.2 Explication des ajouts

Suite à notre analyse du cahier des charges nous avons décidé d'effectuer des ajouts dans la structure de l'application.

En effet dans la classe *Grille* nous avons ajouté la variable *langue* de type *string* qui stockera la langue choisie par l'utilisateur pour l'affichage du tableur et nous avons ajouté également les méthodes *contenu()* et *contenu_developpe()* qui nous permettra de sélectionner le contenu ou le "contenu" ou le "contenu développé" pour une case.

Dans la classe *Case* nous avons pensé à créer une variable *dépendance* de type *Map_cases*, la méthode *actualiserValeur()* permettant de faire les propagations par un parcours de la variable de dépendance propre à chaque case. Nous avons également ajouté des getter afin de récupérer des nom de cases pour *getNom()* et la formule pour *getformule()*.

Dans la classe *Formule* nous avons ajouté *getCasesFormules()* qui retournera la liste de toutes les cases utilisées par la formules, pour pouvoir détecter plus facilement les cas de cycles dans *setFormule()*. De plus nous avons ajouté les méthodes *toString()* et *ToStringDev()*. La méthode *ToStringDev()* permettra d'obtenir sa formule développée récursivement alors que *toString()* permettra d'obtenir sa formule.

Nous avons créé la classe *Fonctions* dans la hiérarchie de classe de racine *Formule* elle aura pour attribut *cases_ope* de type *Collection_cases* et une méthode *getCasesFormules()* qui est une extension de celle déjà présente dans la classe *Formule*, cela permet de convertir les cases de la Formules en listes.

De même dans la classe *OperationBinaire* nous avons ajouté une extension de la méthode *getCasesFormules()*, or ici la liste ne contiendra que 2 éléments.

Dans les classes *Addition*, *Soustraction*, *Multiplication*, *Division*, *Moyenne* et *Somme* nous avons ajouté les extensions de *toString()* et *ToStringDev()* permettant pour chacune d'entre elle d'avoir leur spécificité pour l'écriture du contenu.

3. Choix de SD

3.1. Justification du choix des structures de données

Dans cette application nous allons principalement utiliser des Hashmap et des arraylist en tant que structure de données.

La Hashmap permet d'avoir une relation clé-valeur entre le nom de la case et son contenu, le nom de la case sera constitué de la lettre de la colonne et du numéro de la ligne.

Les Hasmap nous permettront d'acquérir un gain de mémoire très important. En effet une matrice contenant l'ensemble du tableur serait gourmande en mémoire et de plus cela ne sera pas très utile car elle sera creuse.

Les ArrayList sont principalement utilisées par les formules. Ce format permet de stocker simplement les cases que nous utilisons dans nos formules.

3.2 Déclaration dans les classes correspondantes

3.2.1. Déclaration de la Map dans la classe Case

```
private Map<String,Case> cases
```

Déclaration de la Map cases dans la classe Grille

3.2.2. Déclaration de la Map dans la classe Case

```
private Map<String,Case> dependance
```

Déclaration de la Map dependance dans la classe Case

3.2.3. Déclaration de la List dans la classe Fonction

```
private List<Case> cases_ope
```

Déclaration de la liste de case dans la classe Fonction

3.2.4. Retour de la List dans la classe Formule

```
public abstract List<Case> getCasesFormule()
```

Retour de la List dans la classe Formule

4. Principaux traitements

4.1. Evaluation des formules

4.1.1. Classe Formule

```
public abstract class Formule {  
  
    /**  
    *  
    * @return résultat de la formule  
    * @throws IndexOutOfBoundsException cas d'une liste  
    * @throws ArithmeticException  
    */  
    public abstract double eval() ArithmeticException, ListeVideException  
  
    public abstract String toString()  
  
    public abstract String ToStringDev()  
  
    // converti les cases utilisées par la formule en une liste de cases  
    public abstract List<Case> getCasesFormule()  
}
```

Classe formule

4.1.2. Constructeur des classes Moyenne et Somme

```
@param cases_ope : liste de cases utilisées pour calculer la moyenne  
public Moyenne(List<Case> cases_ope) {  
    this.cases_ope = cases_ope  
}
```

Constructeur de la classe Moyenne

```
@param cases_ope : liste de cases utilisées pour calculer la moyenne  
public Somme(List<Case> cases_ope) {  
    this.cases_ope = cases_ope  
}
```

Constructeur de la classe Somme

4.1.3. Méthode *eval()* des classes Moyenne et Somme

```
@ return résultat de la fonction
public double eval() throws ListeVideException{
    double res = 0
    if(cases_ope.size() == 0 ) throw ListeVideException() ;
    for(Case c : cases_ope){
        res += c.valeur()
    }
    return res / cases_ope.size()
}
```

Méthode eval() de la classe Moyenne

```
@ return résultat de la fonction
public double eval() throws ListeVideException{
    double res = 0 ;
    if(cases_ope.size() == 0 ) throw ListeVideException() ;
    for(Case c : cases_ope){
        res += c.valeur() ;
    }
    return res ;
}
```

Méthode eval() de la classe Somme

4.1.4. Méthode *eval()* des classes Moyenne et Somme

```
@ return résultat de la fonction
public double eval() throws ArithmeticException {
    if(droite.valeur() == 0.0 ) throw new ArithmeticException() ;

    return gauche.valeur() / droite.valeur() ;
}
```

Méthode eval() de la classe Division

```
@ return résultat de la fonction
public double eval() {
    return gauche.valeur() * droite.valeur() ;
}
```

Méthode eval() de la classe Multiplication

```
@ return résultat de la fonction
public double eval() {
```

```
return gauche.valeur() - droite.valeur() ;
```

```
}
```

Méthode eval() de la classe Soustraction

@ return résultat de la fonction

```
public double eval() {  
    return gauche.valeur() + droite.valeur() ;  
}
```

Méthode eval() de la classe Addition

4.2. Affectation/modification d'une case par une valeur

4.2.1. Constructeur

```
public Case(int ligne, String colonne) {  
    this.ligne = ligne  
    this.colonne = colonne  
    dependance = new HashMap< String , Case>()  
  
    //valeur de base d'une case est de 0  
    this.valeur = 0  
}
```

Constructeur de la classe Case

4.2.2. Méthode *fixerValeur()* dans la classe *Case*

```
public void fixerValeur(double x) {  
    valeur = x  
    // Une fois la valeur modifié, il y a une propagation qui se fait grâce à la méthode  
    //actualiserValeur  
    this.actualiserValeur()  
}
```

Déclaration de la méthode fixerValeur(double x) dans la classe Case

4.2.3. Méthode *actualiserValeur()* dans la classe *Case*

```
public void actualiserValeur() {  
    // si la case courante utilise une formule  
    if(formule != null) {  
        valeur = formule.eval()  
    }  
    // parcours d'une hashMap  
    Set cle = dependance.keySet()  
    Iterator it = cle.iterator()  
    while (it.hasNext()){  
        String cle = (String) it.next()  
        Case valeur = dependance.get(cle)  
        // récursion  
        valeur.actualiserValeur()  
    }  
}
```

Déclaration de la méthode actualiserValeur() dans la classe Case

4.3. Affectation/modification d'une case par une formule

4.3.1. Méthode *setFormule(Formule f)* dans la classe *Case*

```
public void setFormule(Formule f) throws CycleException {
    List<Case> cs
    cs = f.getCasesFormule()

    //Vérification que les cases de la formule ne sont pas dans notre hashmap de
    dépendance et que les élément de la formule aussi
    for(Case c : cs){
        if(dependance.get(c.getNom()) != null || c == this )
            throw new CycleException()
    }

    //Si la case possède déjà une fonction nous devons supprimer ses dépendances dans les
    autres cases
    if(formule != null){
        cs = formule.getCasesFormule()
        for(Case c : cs ){
            c.getDependance().remove(this.getNom())
        }
    }

    this.formule = f

    for(Case c : cs ){
        c.getDependance().put(this.getNom() , this)
    }
    // actualisation des valeurs
    actualiserValeur()
}
```

Déclaration de la méthode setFormule(Formule f) dans la classe Case

4.3.2. Méthode *getNom()* dans la classe *Case*

```
public String getNom() {
    return colonne + Integer.toString(ligne)
}
```

Déclaration de la méthode getNom() dans la classe case

4.3.4. Propagation des formules

La propagation de l'affectation/modification de formule dans une case se fait grâce à la méthode *actualiserValeur()* qui a été présenté précédemment.

4.4. Accès à une case de la grille par ses coordonnées

4.4.1. Méthode d'accès aux cases dans la class Grille

```
/**
 *
 * @param nomCase nom de la case à sélectionner
 * @return valeur de la case
 * @throws CaseNotFoundException
 */
public double getValeur(String nomCase) throws CaseNotFoundException {
    if (cases.get(nomCase) == null)
        throw new CaseNotFoundException()

    return cases.get(nomCase).valeur()
}
/**
 *
 * @param nomCase nom de la case à sélectionner
 * @return chaîne de caractères correspondant au contenu
 * @throws CaseNotFoundException
 */
public String contenu(String nomCase) throws CaseNotFoundException {

    if (cases.get(nomCase) == null)
        throw new CaseNotFoundException()
    return cases.get(nomCase).contenu_developpe()
}
/**
 *
 * @param nomCase nom de la case à sélectionner
 * @return chaîne de caractères correspondant au contenu développe
 * @throws CaseNotFoundException
 */
public String contenuDeveloppe(String nomCase) throws CaseNotFoundException
{

    if (cases.get(nomCase) == null)
        throw new CaseNotFoundException()
    return cases.get(nomCase).contenu()
}
```

4.4.2. Méthode d'accès à la valeur dans class Case

```
/**
 * @return valeur de la case
 */
public double valeur() {
    return valeur
}
```

Déclaration de la méthode valeur() dans la classe Case

4.4.3. Méthode d'accès au contenu dans class Case

```
/**
 * @return chaîne de caractères correspondant au contenu
 */
public String contenu() {
    if (formule == null) {
        return Double.toString(valeur)
    } else {
        return formule.toString()
    }
}
```

Déclaration de la méthode contenu() dans la classe Case

```
public String toString() {
    String strRes
    strRes = ""
    if (Grille.langue.equals("Francais")){
        strRes += "Somme ("
    }else {
        strRes += "Sum ("
    }
    for (int i = 0 ; i < cases_ope.size() -1 ; i ++ ){
        strRes += cases_ope.get(i).getNom() + ", ";
    }
    strRes += cases_ope.get(cases_ope.size() -1 ).getNom()
    return strRes + ")";
}
```

Déclaration des méthode toString() dans la classe Moyenne (Cette fonction est presque identique pour toutes les classes fils de Fonctions)

```
public String toString() {  
    return gauche.getNom() + " / " + droite.getNom();  
}
```

Déclaration des méthode toString() dans la classe Division (Cette fonction est presque identique pour toutes les classes fils de OpérationBinaire)

4.4.4. Méthodes d'accès au contenu développé dans la classe Case

```
/**  
 *  
 * @return chaîne de caractères correspondant au contenu développe  
 */  
public String contenu_developpe() {  
    if (formule == null) {  
        return Double.toString(valeur);  
    } else {  
        return formule.ToStringDev();  
    }  
}
```

Déclaration de la méthode contenu_developpe() dans la classe Case


```
public String ToStringDev() {  
  
    String strRes ;  
  
    strRes = "" ;  
    if (Grille.langue.equals("Francais") ){  
        strRes += "Moyenne (" ;  
    }else {  
        strRes += "Mean (" ;  
    }  
  
    for (int i = 0 ; i < cases_ope.size() -1 ; i ++ ){  
  
        if(cases_ope.get(i).getFormule() != null){  
            // Si la case utilise une formule, nous opérons la récursion  
  
            strRes += cases_ope.get(i).getFormule().ToStringDev() + ", " ;  
        }else{  
            // Sinon nous afficherons l'identifiant de la case  
            strRes += cases_ope.get(i ).getNom() + ", " ;  
        }  
    }  
  
    //Permet de ne pas finir avec une virgule  
  
    if(cases_ope.get(cases_ope.size() -1).getFormule() != null){  
        strRes += cases_ope.get(cases_ope.size() -1  
).getFormule().ToStringDev() ;  
    }else{  
        strRes += cases_ope.get(cases_ope.size() -1 ).getNom() ;  
    }  
    return strRes + ")" ;  
}
```

Déclaration des méthode ToStringDev() dans la classe Moyenne (Cette fonction est presque identique pour toutes les classes fils de Fonctions)

```
public String ToStringDev() {  
    String str = "" ;  
    if(gauche.getFormule() != null ){  
        // Si la case utilise une formule, nous opérons la récursion  
        str += gauche.getFormule().ToStringDev() ;  
  
    }else{  
        str += gauche.getNom() ;  
    }  
    str += " / " ;  
    if(droite.getFormule() != null){  
        // Si la case utilise une formule, nous opérons la récursion  
        str += droite.getFormule().ToStringDev() ;  
  
    }else{  
        str += droite.getNom() ;  
    }  
  
    return str ;  
}
```

Déclaration des méthode ToStringDev() dans la classe Division (Cette fonction est presque identiques pour toutes les classes fils de OpérationBinaire)

Conclusion

Lors de ses deux premières séances d'analyse et de conception nous avons pu étudier le cahier des charges ainsi que le modèle de départ proposé dans le sujet. Ceci nous a permis de connaître les méthodes et attributs à ajouter dans les classes afin de pouvoir réaliser les fonctionnalités demandées dans le cahier des charges qui étaient manquantes dans le schéma UML de départ.

Nous avons également réfléchi sur la structure de données les plus adaptée au problème, nous avons fait le rapprochement entre la notion de coordonnées de case associées à une valeur et la notion de clé associée à une valeur pour choisir les HashMap comme structure de donnée principales.

Pour l'élaboration du pseudo-code Java qui s'assimile à du code Java, nous avons fait le choix de coder directement les classes demandées avec les méthodes et attributs associées. Nous avons pu ainsi avoir une ébauche de l'application qui fonctionne.