RAPPORT

Projet PPO : TabObj

Mehdi Moussaif et Ibrahim Hajroun
IS4
2020/2021

1. Analyse/Conception

1. Cahier des charges

Aujourd'hui tout est consigné dans un fichier Excel que l'on partage.

1.1.1 Objectif du projet

L'objectif de notre projet est de programmer par objets un noyau de grilles de calcul telles qu'on les voit dans les tableurs ou les formulaires en ligne.

Les cases d'une grille de calcul contiennent soit des valeurs fournies par l'utilisateur, soit calculées automatiquement par une formule faisant référence à d'autres cases (et récursivement).

Le jeu de données qu'on manipule contient des entiers, des réels, des chaînes de caractères, des booléens, des URLs et des images, ainsi que notre ensemble de formules qui renvoie des données numériques, logiques, statistiques, textuelles et graphiques.

1.1.2 Limite du projet

On se limitera dans notre projet à des grilles de calcul numériques (type double) avec un jeu limité de formules de calcul mais qui doit rester facilement extensible.

De plus, on limite le contrôle de l'utilisateur sur le jeu de formules. En effet, l'utilisateur ne peut éditer que les valeurs numériques et ne peut utiliser que les formules disponibles (Somme, Moyenne,...).

Une fois le tableur généré, l'utilisateur ne peut pas créer d'autres cases. Il se limite à éditer les cases existantes.

Dans notre prototype, on se limite à une dénomination en colonne de "A à Z", et d'une numérotation en ligne de "1 à 20" mais le logiciel doit être extensible au-delà.

Toute case a une valeur double par défaut, qui change soit par la modification d'un utilisateur, soit par le résultat d'une formule.

On ne peut affecter une formule à une case si celle-ci peut être cyclique.

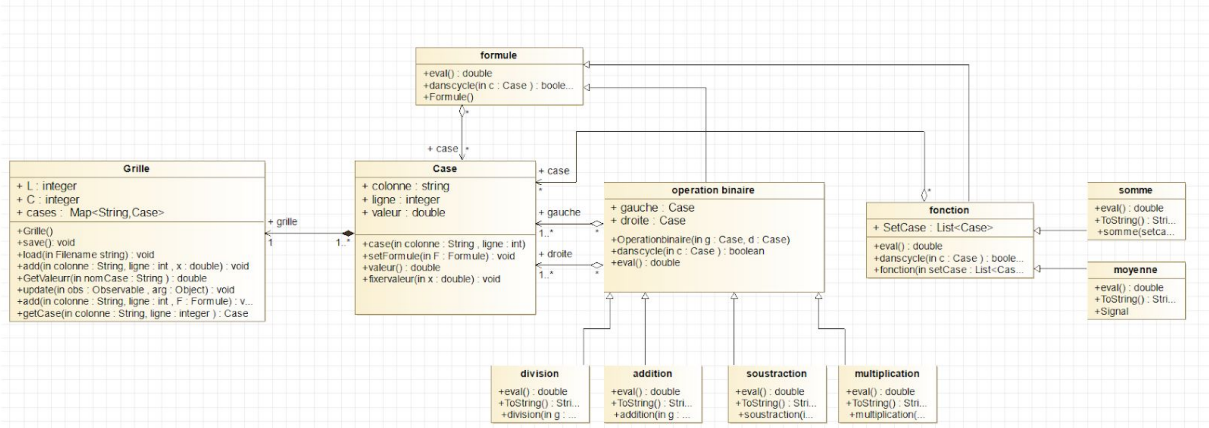
1.1.3 Description fonctionnelle des besoins

Fonction principale : Créer, éditer, enregistrer un ensemble de grilles de calcul

Sous fonctions :

- Éditer chaque case.
- Appliquer les opérations de base (+,-,x,/).
- Support des formules à n arguments : "SOMME" et "MOYENNE" d'un ensemble de cases.

- Consulter chaque case, en affichant sa valeur, son contenu et son contenu développé récursivement



1.2 Schéma UML

Analysons le schéma UML ci-dessus :

- L'objet grille possède les attributs L et C de type ainsi de cases de type Map et les méthodes suivantEs :
  - load(String fileName) permettant de charger un fichier
  - save() permettant de sauvegarder un fichier
  - getCase(string nomCase) permettant d'avoir le nom d'une case
  - getCase(strind colonne , int ligne) permettant aussi d'avoir le nom d'une case, la différence entre ces deux methode réside dans les paramètre utilisé et cela pourrait être pratique dans le cas ou nous aurons à utiliser le nom d'une case qui est en string dans d'autre méthode sans passer par sa ligne et sa colonne
  - add(String colonne, int ligne, double x) qui permet d'ajouter une valeur à une case
  - add(String colonne, int ligne, formule F) qui permet d'utiliser cette fois une formule
  - GetValeur(String nomCase) pour avoir la valeur d'une case (utilisation de nomCase et pas de la colonne et de la ligne)
  - GetContenu(String nomCase) pour avoir le contenu d'une case
  - Update (Observable obs, Object arg) qui lors de la modification d'une case alerte la classe grille et permet de vérifier si les modifications apportées sont possibles.

  Grille est en relation composite avec l'objet case, si grille venait à être supprimé Case ne pourrait plus exister, aussi une grille possède plusieurs ou zero cases, le fait d'avoir zero case nous semble aberrant c'est pour cela que nous avons choisi 1..* entre case et grille.
- La classe Case possède les attributs Colonne "string", Ligne "integer", Valeur "double" et des méthodes suivantes
  - setFormule(formule f) qui permet de mettre une formule dans une case

    - setvaleur(double x) qui permet de mettre une valeur dans une case
    - valeur() qui retourne une valeur pour pouvoir l'utiliser sachant que l'attribut valeur est "private"

Elle est en relation avec plusieurs autres classes :

- Une relation d'agrégation avec Formule, une ou plusieurs case possède 0 ou une formule
- Une relation d'agrégation avec Fonction, une ou plusieurs case possède 0 ou une fonction
- Deux relations d'agrégation avec OpérationBinaire (gauche et droite), une case possède zéro ou une opération binaire

- Les classes Fonction et OperationBinaire héritent de la classe formule
- Les classes addition, soustraction, multiplication et division héritent de l'objet OperationBinaire
- Les classes Moyenne et Somme héritent de la classe fonction.
- On retrouve la méthode eval() qui retourne un double dans plusieurs classes. En effet elle est abstraite (abstract) dans les classes "mères" et développé dans les classes héritées
- à noter aussi les méthodes danscycle() et toString() cette dernière permet d'afficher ce qu'on fait dans la case en fonction de la classe ou elle se trouve. Par exemple si on utilise toString de la classe addition on affichera A4 + B5
- Chacune des classes possède aussi son propre constructeur avec le même nom de sa classe et les mêmes attributs que sa classe.

1.3 Choix de collection

Le choix d'une collection HashMap pour les stocker les cases dans la grille se justifie par le fait qu'on n'a pas besoin d'un ordre précis dans le stockage des cases et un coût en mémoire plus faible que TreeMap car cette dernière se préoccupe de l'ordre de sauvegarde.

Créons une collection Arraylist() de Case dans la classe fonction, une Arraylist est particulière utile parce qu'elle peut contenir des valeur nulle et est ordonnée, son coût de mémoire est faible peut aussi contenir des doublons à contrario de la collection HashSet qui ne peut contenir de doublons sachant que dans un tableur il est très fréquent d'utiliser des doublons. À cela s'ajoute le fait que dans les méthodes toString on a besoin de l'ordre d'insertion des noms des cases surtout pour les classes qui hérite de fonction et des opérations binaires non commutatives telles que le la division ou soustraction.

1.4 Principaux traitements

1.4.1 Evaluation des formules

La méthode eval() est abstraite dans les classes Formule, Operationbinaire, Fonction et développée dans les classes addition, soustraction, multiplication, division, somme et moyenne. Elle renvoie le résultat en double de l'opération binaire ou la fonction utilisée.
Dans les classes addition soustraction et multiplication, la méthode eval est triviale.
Notons alors ɑ={+,*,-} pour éviter de faire trois fois la même fonction. On a alors :

```
eval() : double
        retourne (droite.valeur() ɑ gauche.valeur() )
```

La méthode valeur() se trouve dans la classe Case elle permet d'avoir la valeur d'une case

La fonction eval dans la classe division est similaire à celle des autres operations binaires à l'exception qu'il faut vérifier si l'opération est possible c'est à dire qu'on ne divise pas par zéro

```
eval() : double
        si (droite.valeur() = 0)
                afficher (operation impossible)
        sinon
                retourne (gauche.valeur() / droite.valeur() )
        finsi
finfonction
```

Dans la classe somme eval() se présente comme suit :

```
eval() : double
        retourne (les cases de setcase.valeur().sum())
```

On utilise setcase qui est une list de case et aussi les méthodes de stream() comme tel `stream().mapToDouble(m->m.valeur()).sum()`

Dans la classe moyenne on procède de la même façon que dans somme mais il faudra prendre en compte le cas où la list de case est vide.

```
eval() : double
        Si (setcase est vide)
                afficher( opération impossible)
        Sinon
                retourne (les cases de setcase.valeur().sum()/ setcase.size() )
        fisin
finfonction
```

On utilise Stream aussi comme suit :

```
stream().mapToDouble(m>m.valeur()).sum())/(super.setCase.size()
```

1.4.2 Affectation/modification d'une Case

Méthode *fixerValeur(double x)* de la classe Case, une action qui associe le double *x* à la case en modifiant son attribut valeur

Algo :
fixerValeur(double x)
si (this.valeur() ≠ x) alors
    x=this.valeur()
    setChanged();
    notifyObservers(this);
FinSi
FinAction
À noter que *this* réfère à la case dans laquelle on se trouve.

Méthode *setFormule(Formule f)* de la classe Case, une action qui exécute une formule f qui peut être une *opérationbinaire* ou une *fonction*
Si on se trouve dans un cycle on l'affiche saison on applique la formule souhaitée à la case dans laquelle on se trouve, si la valeur de la case où on est positionné est différente de la valeur qu'est sensé renvoyer la formule en question on appelle la méthodes fixervaleur(), et on informe la classe Observer du changement à l'aide de notifyObservers() qui fera la suite.

```
setFormule(Formule f)
        Si (F.danscycle(this)) alors
                afficher(on est dans un cycle)
        Sinon
                this.formule → F
                double x → F.eval()
                Si (this.valeur() ≠  x) alors
                        this.fixervaleur(x)
                        setChanged()
                        notifyObservers(this)
                FINSI
        FINSI
FINAction
```

1.4.3 Propagation

On peut essayer de comprendre la propagation comme une réaction en chaîne, par exemple la case B6 dépend des valeurs des cases B5 et C3, donc si on modifie une de ces deux cases il faut être en mesure de la notifier et de mettre à jour la cases B6.

Pour cela on utilisera l'interface Observer qu'on implementera à la classe grille et notifiera des changement apportés à l'objet "observable", ici une case des cases appartenant au HashMap de cases dont est constitué la classe Grille à l'aide de la méthode `notifyObservers()` ,propre à l'interface Observer, qu'on utilise dans les methodes `Setformule( Formule F )` et `fixervaleur (double x)` qui sont dans la classe Case.

On peut aussi ajouter autant d'objet "observable" à l'aide de `addObserver()`. Cette même interface nous permet de faire les mises à jour nécessaires avec la méthode suivante :

```
public void update (Observable obs, Object arg) {
    this.cases.values().stream()
            .filter(c -> c.formule != null) // on garde les cases ou il ya une formule
            .forEach(c -> { //pour chaque case c contenat une formule
                try {
                    c.fixervaleur(c.formule.eval()); // on fixe la valeur
                } catch (OperationImpossibleException ex) { // sinon on renvoie un message d'erreur
                    System.out.println("ERREUR FORMULE DANS " + c.colonne + c.ligne);
                }
    });
}
```

1.4.4 Affichage utilisation de toString()

La méthodes to string est utilisée dans les classes héritées de la classe Operationbinaires et les classes héritées de la classe Fonction, elle permet l'affichage de la manière dont est calculée la valeur de la case affichée. On remarque une forte similitude entre les methodes ToString() des sous-classes opérations binaires et une autre ressemblance entre celle des sous-classes fonctions.

Pour les opération binaire notons **β = {+,-,*,/}** ainsi on à le pseudo-code suivant:

gauche de type Case
droite de type Case
ToString() : string
    retourne ("(" + gauche.colonne + gauche.ligne + " **β** " + droite.colonne + droite.ligne + ")")
finAction

Pour les fonction notons ***ƒ*={MOYENNE, SOMME}** ainsi on a le pseudo-code suivant:

s de type string
setcase une list de Case
i entier de parcours
ToString()
    s = "***ƒ***("
    Pour (i parcourant set case)
        s= s + i.colonne + i.ligne
    Finpour
retourner s + ")"
FinAction

1.4.5 Traitement des cycles

dans la classe fonction

danscycle(Case c) action qui prend en paramètre c de type case et qui renvoie un Boolean .

Algo : si cette case contient une des cases de setCase alors on retourne vrai, on est dans un cycle direct sinon on vérifie toutes les cases de setcase et s'il n'y a pas de formule alors on retourne faux car on trouve pas de cycle. Sinon on vérifie que dans toutes les cases de setcase il ya au moins une formule utilisé dans ce cas-là on applique la même action danscycle(c) avec vérification des conditions précédentes alors on est susceptible de trouver un cycle indirect.

```
danscycle ( Case c) : boolean
        Si ( setcase contient c)
                retourne vrai
        Sinon si (toute les cases de setcase ne contiennent pas une
formule)
                retourne faux
        Sinon
                Parcours de setcase et verification si il ya une formule, puis utilisation
                de danscycle(c) sur chaque sous-formule et filtrer celles qui renvoie
                vrai;
                retourne faux si la liste retenue est vide
                retourne vrai sinon
        FinSi
Finaction
```

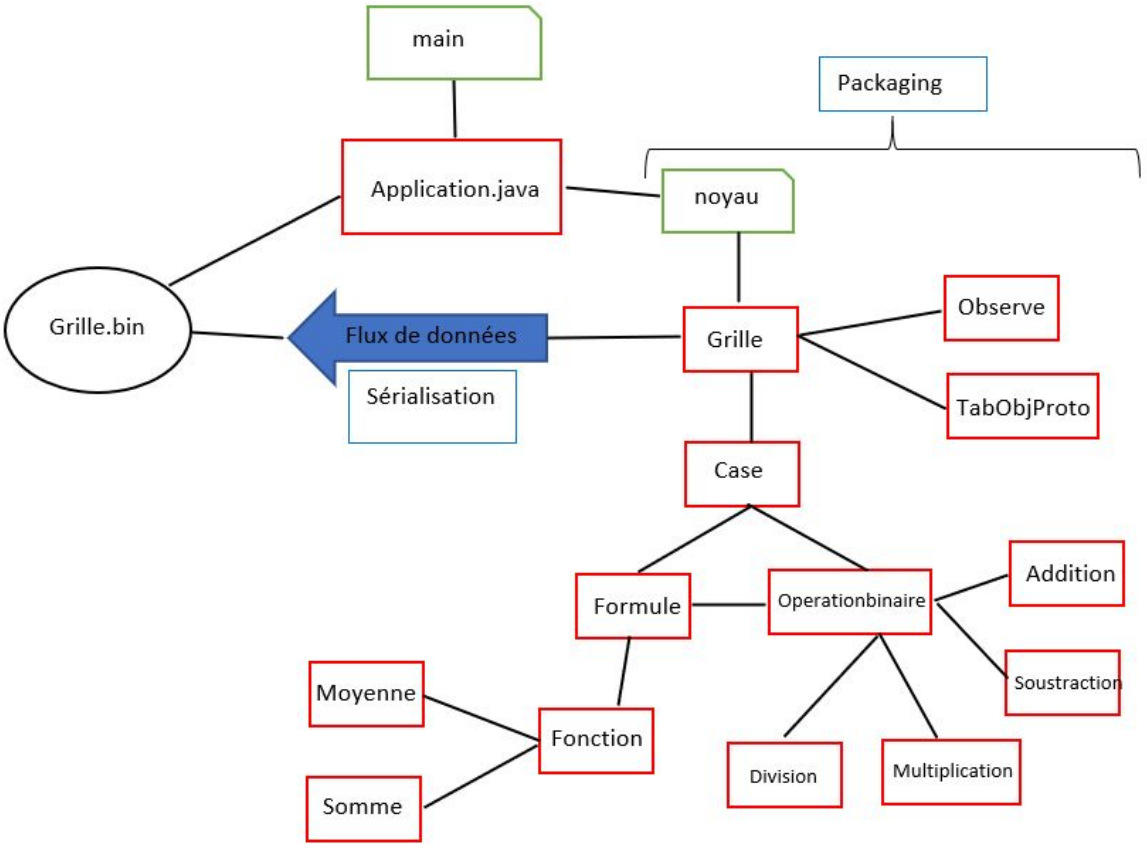
Dans la classe operationbinaire

danscycle(Case c) action qui prend en parametre c de type case et qui renvoie un boolean .

Algo : si cette case contient une des cases utilisées dans les opérations binaires soit la case de gauche ou la case de droite alors on retourne vrai, on est dans un cycle direct sinon on vérifie la case droite ou la case gauche si une des deux utilise une formule alors on retourne faux car on ne trouve pas de cycle. Sinon on applique la même action danscycle(c) avec vérification des conditions précédentes sur la case gauche ou la case droite alors on est susceptible de trouver un cycle indirect.

```
danscycle ( Case c) : boolean
        Si ( c = la case droite ou c = la case gauche )
                retourne vrai
        Sinon si (la case droite.formule= null et la case gauche.formule= null )
                retourne faux
        Sinon
                retourne (gauche.formule.danscycle(c) ou droite.formule.danscycle(c))
        finsi
FinAction
```

2 Arborescence et Sérialisation



L'arborescences du projet est a peu près similaire au schéma UML, la notion de sérialisation est ici visible. On utilise cette notion pour sauvegarder les objets de la classe Grille dans le fichier Grille.bin en les transformant au préalable en flux de données binaires. On a besoin des aussi des méthodes *save() et load(filename)* pour assurer le bon déroulement de la sauvegarde des données.

La classe Grille est sérialisé et donc toutes ses sous-classes le sont aussi par héritage de l'interface Serializable c'est-à-dire les classes Case et Formule.

La classe application qui constitue notre main est relation avec le fichier de sauvegarde Grille.bin pour pouvoir charger les données sérialisées.

3 Mode d'emploi

```java
public static void main(String[] args) throws IOException, ClassNotFoundException{
    Grille g = new Grille(); // on cree un objet de type Grille
    g.add("A", 1, 100.0);
    g.add("A", 2, 50.0);
    g.add("B", 2, 12.0);// utilisation de la fonction add pour ajoueter des elements
    System.out.println("Hello this is B2 " + g.getCase("B2").valeur());// test rapide

    Formule f_add = new Addition(g.getCase("A2"),g.getCase("B2")); // verifcation de la formule addition

    g.add("C", 1, f_add);
    System.out.println("Hello this is C1 " + g.getCase("C1").valeur()); // varification de valeur C1

    g.add("B", 2, 50);
    System.out.println("Hello this is B2 " + g.getCase("B2").valeur());// test rapide
    Formule f_addcycle = new Addition(g.getCase("A1"),g.getCase("C1"));// verifcation de cycle

    g.add("C", 1, f_addcycle);
    System.out.println("Hello this is C1 " + g.getCase("C1").valeur());// test rapide

    System.out.println(g.GetContenu("C1"));
    g.save();// sauvegarde dans le fichier Grille.bin
```

A cela s'ajoute une vidéo explicative que nous vous avons partagé.

4 Bilan et modifications possible

Le problème d'héritages des toString() n'a pas été traité voici une idée à développer

getContenuDev(String nomCase) dans la classe Grille
    Si la case n'a pas de formule alors
        return this.getContenu(nomCase)
    Sinon
        Pour chaque case utilisé dans la formule
            return this.getContenu(nomCase) : getContenuDev(case) +
            case.formule.toString()

En travaillant sur ce projet, surtout dans une période très stressante avec les examens qui s'enchaînent jour pour jour, j'ai dû m'adapter à cet emploi du temps et dû être productif toute la journée. J'ai consacré tout mon effort à la réussite de ce projet qui malgré tout reste incomplet, car la programmation orientée objet utilisée dans la plupart des applications logicielles, des traitements BigData (Hadoop, reduceMap...) est bien meilleure et intéressante que la programmation procédurale qui représentait pour moi une zone de confort où tout s'exprimait simplement en ligne de code contenant des if..else statements. Se jeter dans la gueule du loup qu'est la programmation par objet était la première étape à franchir, et c'était bien le cas avec ce projet. Ce projet m'a aussi aidé à mieux concevoir les projets en général, à découvrir les différentes classes/interfaces de java et à mieux utiliser les interfaces graphiques, le seul moyen dont un non-développeur peut communiquer avec l'application.

Mehdi

Tous les problèmes que, pendant 8 mois, j'évitais me sont tomber dessus en 2 semaines

Ibrahim