|  |
| --- |
|  |
| Projet de PPO : Réalisation d’un tableur en JAVA |
| TOURE Serigne Saliou & GRIMAUD Nicolas  **Tuteur** : RUDAMETKIN Walter |



Sommaire

INTRODUCTION 1

1. SCHEMA UML 3
2. STRUCTURES DE DONNEES 4
3. METHODES ESSENTIELLES 5

Accès Valeur 5

Accès Contenu 6

Accès Contenu Développé 6

Evaluation Valeur 8

Propagation 9

Vérification Cycle 9

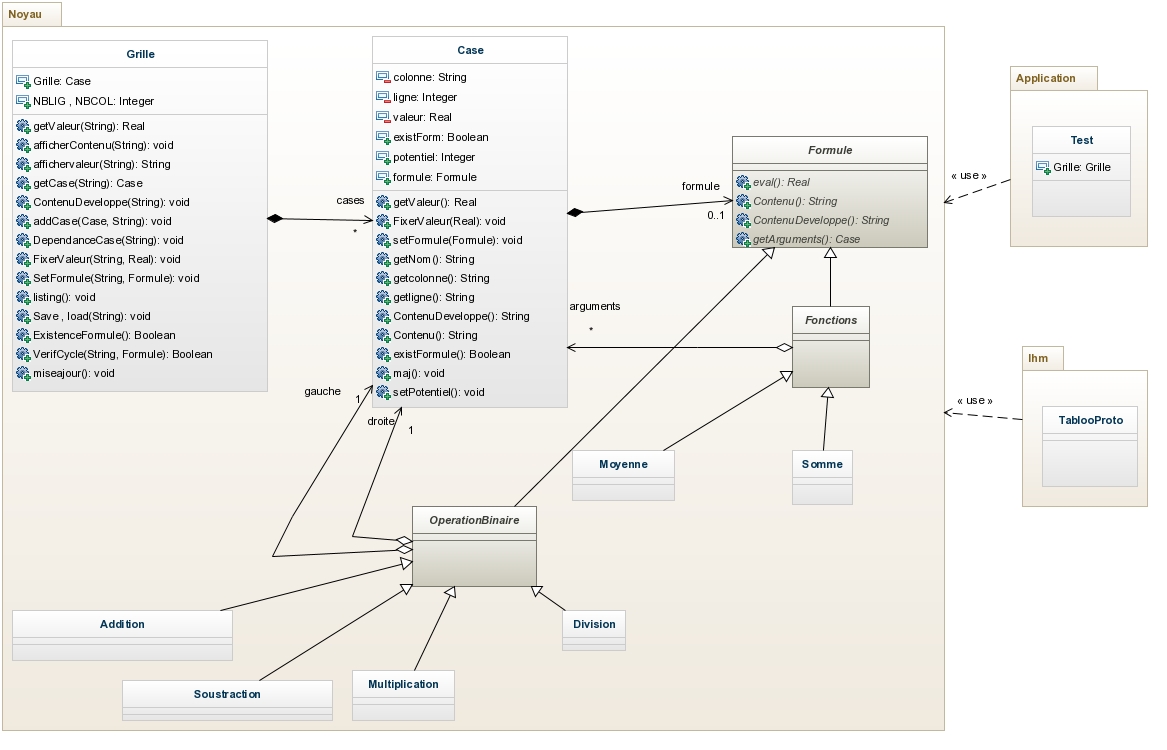
1. BILAN ET AMELIORATION 15
2. LISTING INTEGRAL 16

CONCLUSION 20

Introduction

Après avoir réalisé un premier travail d’analyse et de conception, des modifications ont été apportées quant aux choix de nos structures de données, des méthodes utilisées ainsi qu’au niveau du schéma de conception UML. Dans ce rapport ci nous allons donc détailler les changements apportés ainsi que la démarche adoptée pour mener à bien le projet. Dans une moindre mesure nous évoquerons certaines difficultés rencontrées et les améliorations possibles. Ainsi ce rapport se veut être un complément au rapport d’analyse et de conception

1. SCHEMA DE CONCEPTION UML

Le schéma UML a relativement changé au fur et à mesure qu’on s’est rendu compte des differentes fonctionnalités de notre application. Notons bien que les méthodes présentes dans nos classes d’opérations ne figurent pas dans ce schéma parce qu’elles sont toutes héritées de la classe abstraite Formule.

1. Structures de données

Dans cette phase on redéfinit les structures et les méthodes essentielles qui ont changé par rapport au rapport d’analyse

1. Stockage de la grille

**Map<String,Case> cases = new TreeMap<String,Case>()**

La justification de ce choix est relative au tri automatique qu’opère un TreeMap dans le stockage des éléments. Ce tri pourrait nous être utile lors de la propagation en changeant le paramètre trié en l’occurrence les clés.

1. Stockage d’arguments pour les formules

**List<Case> getArguments() = new ArrayList<Case>()**

On utilise une Arraylist pour stocker les arguments que prend chaque formule. Ce choix est motivé par le fait que le nombre de ces arguments n’est pas connu à l’avance. De plus cette liste permettra de bien implémenter la méthode vérifiant l’existence des cycles.

1. Méthodes Essentielles
   1. Accès à une valeur

Pour accéder à une valeur donnée par ses coordonnées on implémente la méthode ***getValeur()*** qui prend en entrée une chaine de caractère représentant les coordonnées de la case dans la classe **Grille**.

**public** **double** getValeur(String nomCase) **throws** EvaluationException {

**try** {

**return** cases.get(nomCase).valeur();

} **catch** (ArithmeticException e) {

**throw** **new** EvaluationException();

} **catch** (NullPointerException ex) {

**throw** **new** EvaluationException();

}

}

Cette méthode utilise la méthode implémentée dans la classe **Case** retournant la valeur stockée dans une case. Elle se préoccupe aussi des erreurs d’évaluation saisies dans la classe formule.

* 1. Accès au contenu d’une Case

Le contenu d’une case est la valeur d’une case si celle-ci ne contient pas de formule ou la formule sinon. Pour ce faire on implémente la méthode ***getContenu()*** dans la classe **Grille** qui prend en entrée les coordonnées d’une case et fait appelle à la méthode ***Contenu()*** dans la classe **Case**. Enfin la méthode ***Contenu()*** de **Case** fait utilise la méthode ***Contenu()*** présente dans la classe **Formule**.

**CLASSE GRILLE**

**public** String getContenu(String nomCase) {

**return** cases.get(nomCase).Contenu();

}

**CLASSE CASE**

**public** String Contenu() {

**if** (existForm) {

**return** formule.Contenu();

} **else** {

**return** String.*valueOf*(getvaleur());

}

}

Dans la classe **Case** on fait appelle à la méthode ***Contenu()*** de la classe **Formule** s’il existe une formule dans la case ou sa valeur si il n’y a pas de formule. La fonction s’implémente alors dans la classe **Formule** de manière abstraite et donc différente suivant les opérations**.**

**CLASSE FORMULE**

* **ADDITION : OPERATION BINAIRE**

**public** String Contenu() {

**return** "(" + gauche.getNom() + "+" + droite.getNom() + ")";

}

Quand il s’agit d’opération binaire on retourne juste le nom des deux case Gauche et Droite par leur méthodes ***getNom().***

* **MOYENNE : FONCTIONS**

**public** String Contenu() {

String listeC = arguments.get(0).getNom();

**for** (**int** i = 1; i < arguments.size(); i++)

{

listeC = listeC + "," + arguments.get(i).getNom();

}

**return** "MOYENNE" + "(" + listeC + ")";

}

}

Quand il s’agit d’une fonction on affiche, par les mêmes méthodes que précédemment, les noms des arguments de la fonction.

* 1. Accès au contenu développé d’une Case

Le contenu développé d’une case est sa valeur quand celle-ci ne contient pas de formule ou son contenu c’est-à-dire les cases qui sont en arguments ou bien récursivement si les cases en arguments sont elle-même instanciées avec des formules.

De la même manière que pour l’implémentation de la fonction getContenu , on implémente la méthode ***getContenuDeveloppé()*** dans la classe **Grille** qui prend en entrée les coordonnées d’une case et fait appelle à la méthode ***ContenuDeveloppé()*** dans la classe **Case**. Cette dernière utilisera la méthode ***ContenuDeveloppé()*** présente dans la classe **Formule** pour afficher le contenu développé d’une case. Alors ce qui nous intéresse ici c’est la méthode ***ContenuDeveloppé()*** de la classe **Formule** car les autres méthodes qui appellent cette fonction sont identiques que dans l’utilisation de Contenu ( décrite précédemment )

**CLASSE FORMULE**

* **ADDITION : OPERATION BINAIRE**

**public** String ContenuDeveloppe() {

**if** (!gauche.existFormule() && !droite.existFormule()) {

**return** "(" + gauche.getNom() + "+" + droite.getNom() + ")";

}

**if** (!gauche.existFormule()) {

**return** "(" + gauche.getNom() + "+"

+ droite.formule.ContenuDeveloppe() + ")";

} **else** **if** (!droite.existFormule()) {

**return** "(" + gauche.formule.ContenuDeveloppe() + "+"

+ droite.getNom() + ")";

} **else**

**return** "(" + gauche.formule.ContenuDeveloppe() + "+"

+ droite.formule.ContenuDeveloppe() + ")";

}

}

Quand il s’agit d’opération binaire, on différencie les cas à savoir : s’il n’y a aucune formule dans les deux cases gauche et droite, on affiche leur nom comme dans Contenu. S’il existe une formule dans l’une des deux cases gauche ou droite, on affiche le nom de celui qui ne contient pas de formule concaténé avec l’appel récursif de cette même méthode ***ContenuDeveloppé***() sur l’autre case contenant une formule. De ce fait l’appel récursif permettra d’énumérer l’arborescence de la case. Par contre quand toutes les deux cases contiennent une formule, on effectue l’appel récursif sur toutes les deux. On utilisera la même méthode pour toutes les formules de type opération binaire.

* **MOYENNE : FONCTIONS**

La même méthodologie est utilisée pour ce qui est le contenu développé des fonctions. La seule différence ici est qu’on doit parcourir une liste d’arguments donc on doit plus s’intéresser qu’à deux cases mais une liste de cases. La technique reste la même.

**public** String ContenuDeveloppe() {

**if** (!arguments.get(0).existFormule()) {

listeCD = arguments.get(0).getNom();

} **else** {

listeCD = arguments.get(0).ContenuDeveloppe();

}

**for** (**int** i = 1; i < arguments.size(); i++)

{

**if** (!arguments.get(i).existFormule()) {

listeCD = listeCD + "," + arguments.get(i).getNom();

} **else** {

listeCD = listeCD + "," + arguments.get(i).ContenuDeveloppe();

}

}

**return** "MOYENNE" + "(" + listeCD + ")";

}

On initialise une liste, qu’on devra retourner, avec le nom de la première case de la liste arguments si celle-ci ne contient pas de formule sinon on fait un appel récursif à la fonction ***ContenuDeveloppe()*** sur cet élément. Une fois cette liste initialisée on parcourt l’ensemble des cases de la liste d’arguments de cette fonction et on effectue le même traitement pour chaque case de la liste d’arguments. A la fin de la méthode on affiche une liste de contenu développé avec une arborescence très exhaustive. On notera que la même méthode sera utilisée pour toutes les fonctions.

* 1. Evaluation d’une Case

L’évaluation d’une valeur dans une formule se fait suivant chaque formule, cela ne change pas de notre rapport de départ, moyennant la vérification des exceptions d’ évaluation ExceptionEvaluation lorsqu’on divise par zéro ou lorsqu’on effectue une formule avec une liste vide. En l’occurrence :

**CLASSE DIVISION**

**public** **double** eval() **throws** EvaluationException {

**if** (droite.valeur() != 0.0) {

**return** gauche.valeur() / droite.valeur();

} **else** {

**throw** **new** EvaluationException();

}}

Un exception est capturée lorsque le diviseur est nulle.

**CLASSE MOYENNE**

De la même manière lorsqu’il s’agit d’une fonction l’exception est capturée lorsque la liste d’arguments est vide.

**public** **double** eval() **throws** EvaluationException {

**double** somme = 0;

**if** (!arguments.isEmpty()) {

**for** (Case s : arguments) {

somme = s.valeur() + somme;

}

**return** somme / arguments.size();

} **else** {

**throw** **new** EvaluationException();

}

}

Le principal changement qu’on a opéré par rapport au rapport de départ est la propagation et la vérification des cycles avant l’affection d’une formule à une case

* **PROPAGATION**

La propagation sera effectué par le biais d’un regroupement de trois fonctions essentielles. D’abord dans la **classe Case :**

* public void ***setPotentiel*(**) { //fonction setPotentiel() d'une case

if (existForm==false) { //Si la case n'a pas de formule

potentiel=0 ; //potentiel de la case= 0

} //Fin si

else { //Sinon (c'est-à-dire que la case possède une formule)

potentiel=0 ; //potentiel de la case= 0

int p=0 ; //variable locale : p:entier

for(int i = 0; i < formule.getArguments().size(); i++) { //Pour i parcourant la liste (de cases) getArguments() de la formule de la case

if (p < formule.getArguments().get(i).potentiel) { //Si p < potentiel(case parcourue)

p=formule.getArguments().get(i).potentiel ; //p=potentiel(case parcourue)

} //Fin si

} //Fin boucle For

potentiel=p+1 ; //potentiel de la case = p+1

} //Fin sinon

} //Fin fonction

La fonction ***setPotentiel()*** sert à paramétrer le potentiel d'une case.

Le potentiel correspond en fait à la taille maximale de la descendance de sa formule.

Exemple:

A1=4 potentiel(A1)=0

A2=6 potentiel(A1)=0

B1=A1+A2 potentiel(A1)=1

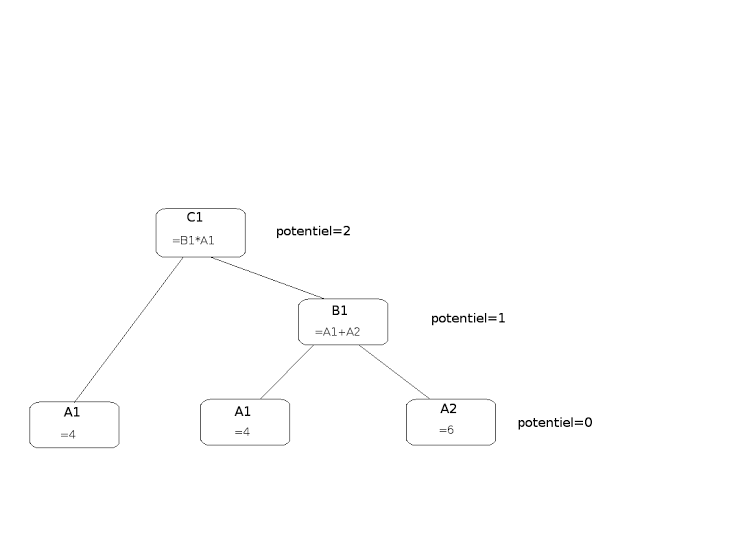
B2=SOMME(A1,A2,A1,A2) potentiel(A1)=1

C1=B1\*A1 potentiel(A1)=2

C2=MOYENNE(B1,B2,A2) potentiel(A1)=2

Ainsi, si une case n'a pas de formule, son potentiel sera de 0.

Si la case possède une formule, on va chercher par le biais d'un parcours dans la liste d'arguments de la case, la case qui a le plus grand potentiel. Le potentiel attribué à la case concernée sera le plus grand potentiel de sa descendance + 1. On illustre assez bien par le dessin ci-dessous.



* ***void maj()*** throws EvaluationException { //fonction maj() d'une case

try {

valeur = valeur(); //valeur de la case mise à jour (par la formule ou par valeur)

} catch (EvaluationException e) { //si EvaluationException est trouvée

ecrire("Erreur lors de l'évaluation de la formule")

}

setPotentiel(); //mise à jour du potentiel

} //Fin fonction

La fonction ***maj()*** d'une case sert à mettre à jour la valeur et le potentiel de cette case qui possède une formule.

S’il existe une formule pour la case, alors elle va donner à sa valeur le résultat de l'évaluation de la formule.

Il sera généré toutefois une exception dans le cas où la formule est une division par zéro, où dans ce cas la valeur de la case prendra la valeur "NaN".

La fonction met ensuite à jour le potentiel de la case.

**DANS LE CLASSE GRILLE :**

* public ***void miseajour()*** throws EvaluationException { //fonction miseajour() d'une grille

int potentiel\_max = 0; //variable locale : potentiel\_max:entier

for (String c : cases.keySet()) { //Parcours des cases de la grille

if (potentiel\_max < cases.get(c).getpotentiel()) { //si potentiel\_max<potentiel(case parcourue)

potentiel\_max = cases.get(c).getpotentiel(); //potentiel\_max=potentiel(case parcourue)}}}

//Fin si

//Fin parcours des cases de la grille

//potentiel\_max est égal au plus grand potentiel, toutes cases confondues

for (int i = 0; i <= potentiel\_max; i++) { //Pour i allant de 0 à potentiel\_max

for (String c : cases.keySet()) { //Parcours des cases de la grille

if (cases.get(c).getpotentiel() == i) { //si potentiel(case parcourue)=i

cases.get(c).maj(); //faire la mise à jour de la case parcourue

} //Fin si

} //Fin parcours des cases de la grille

} //Fin boucle For

} //Fin fonction

La fonction ***miseajour()*** de la grille sert à mettre à jour toutes les cases de la grille.

Elle va tout d'abord détecter le plus grand potentiel d'une case présente dans la grille, en effectuant un parcours.

Ensuite, elle va appliquer la fonction ***maj()*** de la classe **Case** à chaque case une par une et par potentiel croissant, afin d'effectuer une bonne propagation.

Il est à noter que si nous avions réussi à trier la grille par potentiel croissant, alors la boucle utilisant l'indice i pourrait être supprimée. On parcourt (potentiel\_max+1) fois la grille au lieu d'une seule fois. C'est pour cette raison que nous avions choisi une structure de TreeMap pour la grille.

* **VERIFICATION DE CYCLES**

L’objectif est de verifier qu’une formule ne genere pas de cycle avant de pouvoir l’appliquer à une case. Ceci revient donc à verifier si une case n’est pas argument de la formule qu’on souhaite lui affecter directement. Inderectement aussi elle ne doit pas etre dépendants des arguments qui forment cette formule. Nous avons donc mis en place 2 principales fonctions avant de verifier ceci.

Dans la **Classe Formule** : on déclare une méthode abstraite qui retourne une liste d’arguments ***getArguments()*** . Le but de cette liste est de retourner l’ensemble des arguments qui forment une formule directement et indirectement. Dans cette liste on retrouve l’arborescence totale des arguments . Ainsi suivant les types de formule cette méthode s’implement de manière un peu différente.

* **ADDITION : OPERATION BINAIRE**

**public** List<Case> getArguments() {

List<Case> listArg = **new** ArrayList<Case>();

listArg.add(gauche);

listArg.add(droite);

**if** (!gauche.existFormule() && droite.existFormule()) {

listArg.addAll(droite.formule.getArguments());

} **else** **if** (!droite.existFormule() && gauche.existFormule()) {

listArg.addAll(gauche.formule.getArguments());

} **else** **if** (droite.existFormule() && gauche.existFormule()) {

listArg.addAll(gauche.formule.getArguments());

listArg.addAll(droite.formule.getArguments());

}

HashSet<Case> hs = **new** HashSet<Case>();

hs.addAll(listArg);

listArg.clear();

listArg.addAll(hs);

**return** (listArg);}

Dans une opération Binaire, dès le début de la méthode on stocke dans la liste les deux cases Gauche et droite. On vérifie ensuite si ces cases contiennent des formules ou non. Si c’est le cas on effectue l’appel récursif de la méthode sur la case qui contient une formule ou sur les deux si elles contiennent toutes des formules, ensuite on concatène les listes obtenues par la méthode ***addAll()*** des listes. A la fin de la méthode, puisqu’on a fait une récursivité on peut obtenir des doublons pour une liste d’arguments de formule. Par souci d’optimalité on supprime ces doublons en transférant la liste dans une ***Hashset*** qui elle n’accepte pas les doublons puis on récupère la liste nettoyée et on la renvoie.

La méthode reste la même pour toutes les opérations binaires.

* **MOYENNE : FONCTION**

S’agissant des fonctions, on prend l’exemple de la classe moyenne.

**public** List<Case> getArguments() {

List<Case> listArg = **new** ArrayList<Case>();

List<Case> listArg0 = **new** ArrayList<Case>();

listArg0 = arguments;

**int** i;

**for** (i = 0; i < arguments.size(); i++) {

**if** (arguments.get(i).existFormule()) {

listArg.addAll(arguments.get(i).formule.getArguments());

}

}

listArg.addAll(listArg0);

HashSet<Case> hs = **new** HashSet<Case>();

hs.addAll(listArg);

listArg.clear();

listArg.addAll(hs);

**return** (listArg);

}

L’idée reste la même que pour les opérations binaires. Ici dès le début on rentre dans une liste de nom listArg0, toute la liste des arguments de la fonction. Ensuite on parcourt toute la liste des arguments déjà existante pour voir les arguments possédant une formule. Si on trouve un argument qui possède une formule on fait un appel récursif sur cet argument et on concatène la liste d’arborescence obtenue récursivement après avoir parcouru tous les arguments avec la liste d’arguments de départ. On effectue le même tri que précédemment pour éliminer les doublons.

IL faut alors utiliser ces méthodes dans la grille. Dans la **Classe GRILLE :** On implémente une fonction ***VerifCycle(string,formule)*** qui prend en paramètre les coordonnées d’une case et une formule donnée et qui renvoie un booléen.

**public** **boolean** verifCycle(String nomCase, Formule f) {

**boolean** d = **false**;

**for** (**int** i = 0; i < f.getArguments().size(); i++) {

**if** (nomCase.equals(f.getArguments().get(i).getNom())) {

d = **true**;

}

}

**return** d;

}

On parcourt la liste des arguments de la formule donnée en paramètre et on regarde si la case donnée ne figure pas dans cette liste. Si oui on renvoie **true** sinon **false**.

De ce fait dans la ***fonction SetFormule(string,formule)*** qui affecte une formule à une case on effectue la vérification grâce à ***VerifCycle*** avant de procéder à l’affection et à la mise à jour par propagation.

**public** **void** setFormule(String nomCase, Formule f)

**throws** EvaluationException {

**try** {

**if** (!verifCycle(nomCase, f)) {

cases.get(nomCase).setFormule(f);

miseajour();

} **else** {

System.***out***.println("Impossible evaluation de "

+ getCase(nomCase).getNom() + ':' + "Cycle Existant");

}

} **catch** (NullPointerException ex) {

System.***out***.println("La case " + nomCase + " est inexistante.");

} **catch** (EvaluationException e) {

System.***out***.println("FO.");

}

1. BILAN & AMELIORATION

* **FONCTIONS EN ANGLAIS :**

Nous nous sommes rappelés assez tard de l’extension de la langue pour l’utilisation et l’affichage de la grille. Cependant il n’aura été pas très compliqué de le faire vu qu’il s’agit uniquement d’un affichage. Les méthodes d’affichage abstraites des formules seraient mis alors en Anglais avec seul changement les noms des fonctions (Moyenne devient Mean) et on les utiliserait dans l’interface graphique en donnant la main à l’utilisateur pour qu’il fasse le choix de la langue à afficher.

* **Problème concernant les listes vides**

Nous n'avons pas eu le temps de traiter entièrement l'exception qui devrait être générer dans le cas de somme ou moyenne d'une liste vide. Certes nous avons implémentation la saisie de cette exception lors de l’évaluation mais elle ne se lève pas lors des tests.

Cette exception aurait été comprise dans EvaluationException (avec la division par zéro), donc l'affichage aurait été "???"

Par ailleurs, notons que la grille devrait pouvoir être extensible pour se rapprocher au mieux d’un tableur classique. De même, l’édition des formules devrait pouvoir se faire et par la même occasion l’analyse syntaxique. Le jeu de données devrait pouvoir être plus large.

1. LISTING INTEGRAL

* **PACKAGE noyau**

**Case.java** : (classe)

-public Case(String col, int lig) : Constructeur de Case prenant en paramètres sa colonne et sa ligne

-public Case(String col, int lig, double d) : Constructeur de Case prenant en paramètres sa colonne, sa ligne et sa valeur

-String getcolonne() : Retourne la colonne de la case

-int getligne() : Retourne la ligne de la case

-public String getNom() : Retourne le nom de la case : la colonne et la ligne

-public double getvaleur() : Retourne la valeur de la case

-double valeur() throws EvaluationException : Retourne la valeur fournie par la formule ou la valeur locale s'il n'y a pas de formule

-int getpotentiel() : Retourne le potentiel de la case

-public void fixerValeur(double x) : Assigne x à la valeur de la case, supprime la formule de la case

-public void setFormule(Formule f) : Assigne f comme formule dans la case

-public boolean existFormule() : Vérifie l'existence d'une formule dans la case

-public void setPotentiel() : Détermine et affecte le potentiel de la case

-void maj() throws EvaluationException : Met à jour la valeur et le potentiel de la case

-public String Contenu() : Retourne le contenu de la formule de la case

-public String ContenuDeveloppe() : Retourne le contenu développé de la formule de la case

**Grille.java :** (classe))

La grille est une TreeMap de cases

-public void save(String backupName) throws IOException : Permet de sauvegarder l'état de la grille

-public void load(String backupName) throws IOException, ClassNotFoundException : Permet de charger la grille

-public Case getCase(String nomCase) : Retourne le nom de la case de clé "nomCase"

-public double getValeur(String nomCase) throws EvaluationException : Retourne la valeur de la case de clé "nomCase"

-public int getColonne() : Retourne le nombre de colonnes de la grille

-public int getLigne() : Retourne le nombre de lignes de la grille

-public boolean ExistenceFormule(String nomCase) : Vérifie si la case "nomCase" contient une formule

-public void addCase(String nomCase, Case c) : Ajoute une case "c" de clé "nomCase" dans la grille

-public void fixerValeur(String nomCase, double d) throws EvaluationException : Assigne à la case "nomCase" la valeur "d"

-public boolean verifCycle(String nomCase, Formule f) : Vérifie si la case "nomCase" est présente parmi les cases de la liste de la formule "f"

-public void setFormule(String nomCase, Formule f) throws EvaluationException : Assigne à la case "nomCase" la formule "f", sauf en cas d'exception (cycle,case inexistante)

-public void miseajour() throws EvaluationException : Met à jour toutes les cases de la grille une par une, la propagation se faisant grâce aux potentiels des cases

-public String getContenu(String nomCase) : Retourne le contenu de la case "nomCase"

-public String getContenuDeveloppe(String nomCase) : Retourne le contenu développé de la case "nomCase"

-public void afficherValeur(String nomCase) : Affiche la valeur de la case "nomCase"

-public void afficherContenu(String nomCase) : Affiche le contenu de la case "nomCase"

-public void afficherContenuDeveloppe(String nomCase) : Affiche le contenu développé de la case "nomCase"

-public void listing() throws EvaluationException : Affiche la valeur, le contenu et le contenu développé de chaque case de la grille

**Fonctions.java** : (sous-classe de Formule)

-Fonctions(List <Case>liste) : Constructeur de Fonctions prenant en paramètre une liste de cases

**Formule.java** : (classe abstraite)

-abstract public double eval() throws EvaluationException ;

-abstract public List<Case> getArguments();

-abstract public String Contenu();

-abstract public String ContenuDeveloppe();

**EvaluationException**.java : (sous-classe de Exception)

Sert à effectuer les évaluations des formules sans erreur

**OperationBinaire.java** : (sous-classe de Formule)

-OperationBinaire(Case left, Case right) : Constructeur de OperationBinaire prenant en paramètres deux cases

**Addition.java** : (sous-classe de OperationBinaire)

-public Addition(Case left, Case right) : Constructeur de Addition prenant les paramètres de OperationBinaire

-public List<Case> getArguments() : Retourne la liste toutes les cases contenues dans l'arborescence de l'addition (sans doublon)

-public double eval() throws EvaluationException : Retourne la valeur du calcul de l'addition, sauf en cas d'exception générée (liste vide)

-public String Contenu() : Retourne le contenu de l'addition

-public String ContenuDeveloppe() : Retourne le contenu développé de l'addition

**Soustraction.java** : (sous-classe de OperationBinaire)

-public Soustraction(Case left, Case right) : Constructeur de Soustraction prenant les paramètres de OperationBinaire

-public List<Case> getArguments() : Retourne la liste toutes les cases contenues dans l'arborescence de la soustraction (sans doublon)

-public double eval() throws EvaluationException : Retourne la valeur du calcul de la soustraction, sauf en cas d'exception générée (liste vide)

-public String Contenu() : Retourne le contenu de la soustraction

-public String ContenuDeveloppe() : Retourne le contenu développé de la soustraction

**Multiplication.java** : (sous-classe de OperationBinaire)

-public Soustraction(Case left, Case right) : Constructeur de Multiplication prenant les paramètres de OperationBinaire

-public List<Case> getArguments() : Retourne la liste toutes les cases contenues dans l'arborescence de la multiplication (sans doublon)

-public double eval() throws EvaluationException : Retourne la valeur du calcul de la multiplication, sauf en cas d'exception générée (liste vide)

-public String Contenu() : Retourne le contenu de la multiplication

-public String ContenuDeveloppe() : Retourne le contenu développé de la multiplication

**Division.java :** (sous-classe de OperationBinaire)

-public Soustraction(Case left, Case right) : Constructeur de Division prenant les paramètres de OperationBinaire

-public List<Case> getArguments() : Retourne la liste toutes les cases contenues dans l'arborescence de la division (sans doublon)

-public double eval() throws EvaluationException : Retourne la valeur du calcul de la division, sauf en cas d'exception générée (liste vide, division par zéro)

-public String Contenu() : Retourne le contenu de la division

-public String ContenuDeveloppe() : Retourne le contenu développé de la division

**Somme.java** : (sous-classe de Fonctions)

-public Somme(List<Case> liste) : Constructeur de Somme prenant les paramètres de Fonctions

-public List<Case> getArguments() : Retourne la liste toutes les cases contenues dans l'arborescence de la somme (sans doublon)

-public double eval() throws EvaluationException : Retourne la valeur du calcul de la somme, sauf en cas d'exception générée (liste vide)

-public String Contenu() : Retourne le contenu de la somme

-public String ContenuDeveloppe() : Retourne le contenu développé de la somme

**Moyenne.java** : (sous-classe de Fonctions)

-public Somme(List<Case> liste) : Constructeur de Moyenne prenant les paramètres de Fonctions

-public List<Case> getArguments() : Retourne la liste toutes les cases contenues dans l'arborescence de la moyenne (sans doublon)

-public double eval() throws EvaluationException : Retourne la valeur du calcul de la moyenne, sauf en cas d'exception générée (liste vide)

-public String Contenu() : Retourne le contenu de la moyenne

-public String ContenuDeveloppe() : Retourne le contenu développé de la moyenne

* **PACKAGE ihm**

**TablooProto.java**

Cette classe est utilisée pour l'interface graphique

* **PACKAGE application**

**Test.java (classe)**

C'est ici que l'on crée notre grille.

On crée des listes qui seront utilisée pour les formules, et on y ajoute des cases .

On crée en dur les cases avec les fonctions addCase(), avec des valeurs ou non.

On crée en dur les formules.

On associe les formules aux cases.

On affiche l'état de la grille avec la fonction listing().

On réalise ensuite la sérialisation.

CONCLUSION

En conclusion ce projet fut d’un apport très considérable, il nous a permis de bien évaluer la puissance et l’efficacité du langage de programmation orienté objet Java sur un travail d’une assez importante envergure. Par ailleurs nos compétences ont bien été développées notamment dans l’utilisation de l’IDE ECLIPSE qui est pourra nous servir tout au long de notre cursus et carrière.