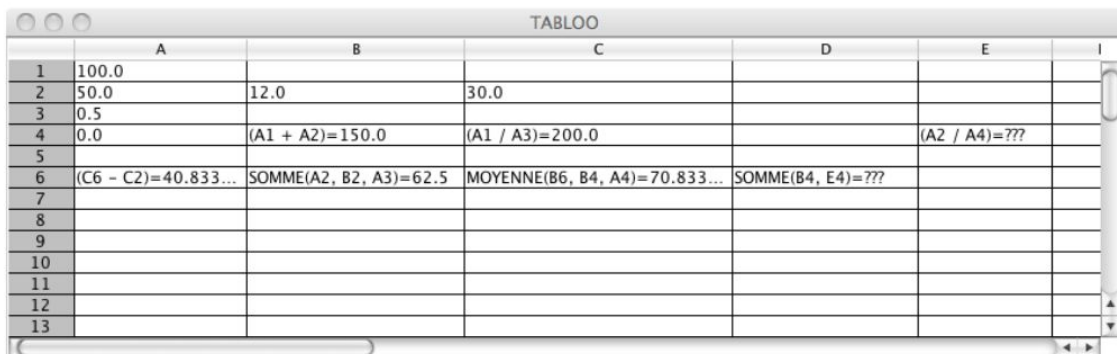


Projet de PPO

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	2
1. Analyse/conception	3
1.1. Analyse/conception générale du cahier des charges du noyau	3
1.1.1. Classe Grille	3
1.1.2. Classe Case	4
1.1.3. Hiérarchie de classes Formule	5
1.1.4. Les extensions de la classe Exception	6
1.1.5. Interface graphique	7
1.2. Schéma UML généré à partir de GenMyModel	8
1.3. Choix des Structures de données	9
1.3.1. Justification du choix des structures de données	9
1.3.2 Déclaration dans les classes correspondantes	9
1.4. Principaux traitements	10
1.4.1. Evaluation des formules	10
1.4.2. Affectation/modification d'une case par une valeur	12
1.4.3. Affectation/modification d'une case par une formule	14
1.4.4. Accès à une case par sa valeur	15
3. Description des tests et résultats	26
3.1. Test de création des cases de la grille	26
3.2. Test de propagation lors d'une affectation d'une nouvelle valeur à une case	26
3.3. Test de détection de cycle	26
3.4. Test des méthodes permettant de faire de demander le contenu et le contenu développé d'une case	26
3.5. Test de sauvegarde, de modification et de chargement de la grille	27
4. Mode d'emploi	28
4.1. Exécution du projet sans argument	28
4.2. Exécution du projet avec argument	28
4.2. Générer un fichier sérialisé	28
5. Bilan	29
5.1. La réalisation du projet	29
5.2. Les fonctionnalités de notre projet	29
5.3. Bilan personnel	30
5.3.1. Bilan personnel Jordan Laplace	30
5.3.2 Bilan personnel Maxime Sauvage	30

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 menée durant les deux premières séances mais il a aussi pour objectif de présenter le projet réalisé durant le reste des séances et pendant notre temps de travail personnel.

Dans ce compte rendu, nous présentons l’Analyse/Conception générale du projet retravaillée au cours des séances supplémentaires, il contient également une arborescence détaillée des fichiers du projet de l’application et la partie “1.1.5.” parlant des fonctionnalités supplémentaires que nous avons développées. Par la suite nous expliquons les tests réalisés pour valider le bon fonctionnement de nos méthodes. Nous disposons dans une quatrième partie, le mode d’emploi de notre application. Enfin nous avons effectué un bilan de notre projet sur nos réalisations ainsi qu’une réflexion personnelle sur notre travail.

1. Analyse/conception

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

1.1.1. Classe Grille

La grille de calcul sera un objet de classe *Grille*, cette classe sera en relation avec les cases à l'aide de la structure de données que nous avons choisie. Le choix de la structure de données est détaillé dans la partie 1.3. Choix des structures de données.

La gestion de la langue se fera à l'aide d'un attribut propre à la classe : *langue*, celle-ci pourra être modifiée directement depuis l'application Tabloo.

Dans cette classe, nous retrouvons la méthode *Valeur()* permettant de retourner la valeur numérique de la case sauf si elle n'existe pas ou si l'évaluation de la valeur de la case a déclenché une erreur.

Il y a également les méthodes de modification de la grille pour l'interface graphique : *modifCreaCase(String nomCase, double d)* permet de modifier une case dans la listes des cases de la grille, si elle n'existe pas elle est ajoutée dans la liste avec comme nom le paramètre *nomCase*. Ces méthodes prennent aussi bien un double qu'une fonction en paramètre pour créer ou modifier la case.

Nous avons aussi les méthodes permettant de sauvegarder et charger une grille via les méthodes *sauvegarderGrille(String path)* et *chargerGrille(String Path)*.

Ensuite, l'affichage du contenu des cases se fera à l'aide de plusieurs méthodes en fonctions de ce que l'on souhaite voir. Il y a donc *affichageValeur(String nomCase)* qui affiche la valeur, *affichageContenu(String nomCase)*, *affichageContenuS(String nomCase)* qui affiche le contenu de la case tel qu'une formule, *affichageContenuD(String nomCase)* et *affichageContenuDS(String nomCase)* qui affiche le contenu développé de la case. Il y a deux méthodes pour l'affichage de contenu des cases, la différence se fait par le résultat qui comprend ou non "= + le résultat".

Enfin, nous avons ajouté un programme principal main qui permet de générer le fichier *grilleSujet.dat* qui utilise la méthode de création de case afin de reproduire l'exemple du sujet et qui sauvegarde la grille générée avec la méthode *sauvegarderGrille(String path)*.

1.1.2. Classe Case

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

- *nom* de type *String*
- *valeur* de type double
- *formule* de type *Formule*, il s'agit de la relation entre *Case* et *Formule*.
- *successeur* de type *List<Case>*
- *predecesseur* de type *List<Case>*
- *isCaseErreur* de type boolean, qui détermine si une case contient une erreur.

Ainsi *successeur* et *predecesseur* justifient l'auto-association sur *Case*.

Notre classe admet deux constructeurs pour créer une instance de notre objet. L'un prend une formule et l'autre prend un double. Ainsi nos constructeurs requièrent un double ou une formule pour que la case créée ait une valeur. Ceci évite de donner à une case une valeur par défaut ou de la considérer comme inutilisable avant la première utilisation de *fixerValeur(double d)* ou *setFormule(Formule f)*.

Lors de l'utilisation de la méthode *setFormule(Formule f)* qui permet d'attribuer une formule à une *Case*, 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, donc nous avons pensé à en créer une permettant de lier une case aux cases qui ont besoin d'elle, c'est là que *successeur* et *predecesseur* entrent en jeu, ces attributs permettent pour chaque *Case* de stocker les cases qui dépendent de nous via la méthode *addPreSucc*

Ainsi, grâce à cette relation de dépendance entre les cases nous pouvons détecter les cycles direct et indirect via les méthodes *rechercheCycle(Formule f)* et *rechercheCycleIndirect(Case racine)* quand une formule est attribuée à une *Case*. *rechercheCycleIndirect(Case racine)* détecte si la racine (case que nous utilisons pour détecter le cycle) est présente dans la liste de ses prédécesseurs, si dans une des récursions racine correspond à un des prédécesseurs alors cela implique que nous bouclons. Nous déclencherons alors des erreurs que nous aurons créées pour indiquer le problème 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 provenant de la hiérarchie de classes *Formule*, ces méthodes sont *contenu*, *contenuS()*, *contenu_developpe()* et *contenu_developpeS()*. De façon analogue à la classe *Grille*, on trouve deux méthodes par type de contenu qui se différencient par la présence du "= + résultat".

Lors de l'utilisation de la méthode *FixerValeur(double d)*, l'attribut *isCaseErreur* est remis à faux par défaut, la valeur de la case vaut alors le paramètre de la méthode, et si la case contenait une formule elle est écrasée. Ses prédécesseurs sont retirés car la case ne dépend plus d'autre case, ensuite les cases qui dépendent de cette dernière qui est modifiée sont alors actualisées grâce à la méthode *actualiserValeur()*.

Enfin, nous devons gérer la modification de la valeur numérique d'une case par la méthode *actualiservaleur()* qui permet de recalculer les cases qui dépendent de nous dans l'ordre lors d'un changement de valeur.

1.1.3. Hiérarchie de classes Formule

Les formules sont représentées par une hiérarchie de classes de racine abstraite *Formule*, le lien entre la classe *Case* et la classe *Formule* se fait par l'attribut *formule*.

Elle contient les méthodes suivantes : *eval()* qui calcule la valeur de la case selon le type d'opération, *toString()* qui permet de renvoyer le contenu d'une formule avec le terme "=" et le résultat de la formule, *toStringSimple()* qui renvoie le contenu simple de la formule, *toStringDev()* qui renvoie le contenu développé de la formule avec le terme "=" et le résultat de la formule, *toStringDevSimple()* qui renvoie le contenu développé simple de la formule, *getCasesFormule* retourne la liste des cases qui sont concernées par la formule.

La classe *Formule* possède deux sous-classes : *OperationBinaire* et *Fonction*. *OperationBinaire* implémente *getCasesFormule()* en créant une liste de cases avec l'ajout de *gauche* et *droite* dans la liste de cases. *Fonction* implémente *getCasesFormule* en retournant la liste *cases_ope* cet attribut étant le lien entre *Fonction* et *Case*.

La classe *OperationBinaire* possède quatre sous-classes : *Addition*, *Soustraction*, *Multiplication* et *Division*. Chacune de ces classes implémente toutes les méthodes de *Formule* en fonction du type d'opération.

La classe *Fonction* possède deux sous-classes : *Somme* et *Moyenne*. Chacune de ces classes implémente toutes les méthodes de *Formule* en fonction du type de fonction.

1.1.4. Les extensions de la classe *Exception*

Nous avons créé cinq classes qui étendent la classe *Exception* permettant de gérer les erreurs liées aux traitements de l'application Tabloo.

ArithmeticException est lancée lorsque qu'une division par zéro est effectuée, elle est manipulée notamment dans la méthode *eval()* de la classe *Division*.

CaseErreurException est lancée lorsqu'une *Case* n'a pas de valeur, c'est une erreur qui peut se transmettre d'une case à une autre à l'aide de l'attribut *isCaseErreur*.

CaseNotFoundException est lancée lorsqu'on essaye d'accéder à une case qui n'existe pas.

ListeVideException est lancée lorsqu'il n'y a pas de cases dans la liste de cases en paramètres de la méthode *eval()* des classes *Somme* et *Moyenne*.

Enfin, *CycleException* est lancée lorsqu'un cycle est créé lors d'un appel de *setFormule(Formule f)*.

1.1.5. Interface graphique

En plus de réaliser l'interface graphique qui nous a été demandé. Nous avons pris l'initiative d'ajouter des fonctionnalités pour exploiter l'ensemble de notre noyau. Nous allons passer en revue les différentes modification et ajout que nous avons réalisé à l'interface graphique de base pour en arriver à ce résultat.

Tout d'abord, Nous avons séparé la classe `MyTableModel` de la classe `TablooProto`. Ceci permet d'avoir moins de code dans la classe `TablooProto` pour la rendre plus lisible.

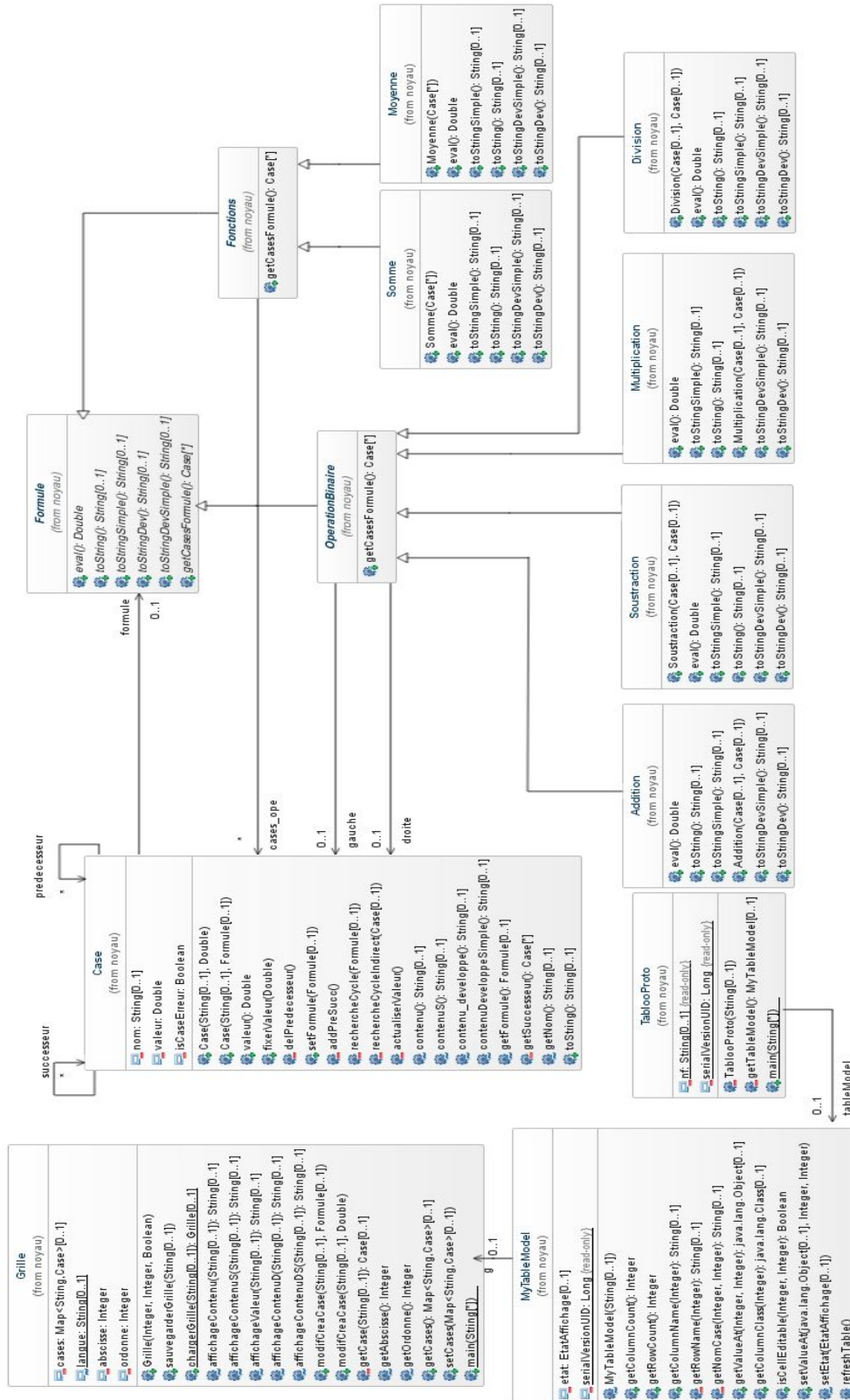
Ensuite, Nous avons ajouté les méthodes *`setEtat`* et *`refreshTable`* suivante dans `MyTableModel` ainsi qu'un enum *`EtatAffichage`* et une variable *`EtatAffichage`* nommée *`etat`*.

- *`EtatAffichage`* est un enum permettant de connaitre dans quel état l'utilisateur à décider d'afficher les données (`contenu` , `contenu_simple` , `contenu_développé` , `contenu_développé_simple` , `valeur`).
- *`setEtat(EtatAffichage e)`* permet de modifier la variable *`etat`*.
- *`refreshTable()`* permet de rafraîchir l'affichage.

Aussi, Nous avons aussi modifié la méthode `getValueAt` dans `MyTableModel`. Nous avons ajouté un switch permettant ,en fonction de la variable état, de choisir la chaîne de caractère à afficher.

Enfin nous avons ajouté un menu à notre interface graphique. Ce menu permet de modifier la langue et le contenu à afficher. L'initialisation de ce menu se situe dans le *`main`* de la classe *`tablooProto`*.

1.2. Schéma UML généré à partir de GenMyModel



1.3. Choix des Structures de données

1.3.1. Justification du choix des structures de données

Lors de la conception de l'application, nous avons dû choisir des structure de données afin de stocker les données des cases selon la classe qui les utilise. Ainsi dans la classe Grille nous utilisons une Hashmap car on ne soucie pas de l'ordre des cases. L'association clé-valeur permet d'accélérer le temps de recherche d'une case dans la Map.

Les alternatives de cette Hashmap étaient une List qui est peu pratique lorsqu'il y a un grand nombre de données à stocker ainsi qu'une TreeMap qui propose un ordre interne dans le stockage qui ne nous intéresse pas lors du traitement de la grille.

Pour les relations de dépendances entre les cases et les paramètres des méthodes de fonctions nous utilisons des List, en effet cette structure nous permet un parcours simple, des ajouts et suppressions faciles lorsque le nombre de données à stocker dedans est petit, ce qui correspond bien à ce qu'on avait besoin.

1.3.2 Déclaration dans les classes correspondantes

Dans la classe *Grille* :

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

Dans la classe *Case* :

```
private List<Case> predecesseur;
```

```
private List<Case> successeur;
```

Dans la classe *Fonction* :

```
protected List<Case> cases_ope ;
```

1.4. Principaux traitements

1.4.1. Evaluation des formules

```
public double eval() throws CaseErreurException {  
    Calcul de l'Addition  
    try { return gauche.valeur() + droite.valeur() ;}  
    Si une des cases contient une erreur, une CaseErreurException est lancée  
    catch (CaseErreurException e) {  
        throw new CaseErreurException() ;  
    }  
}
```

Méthode eval() dans la classe Addition

```
public double eval() throws CaseErreurException {  
    Calcul de la Soustraction  
    try {return gauche.valeur() - droite.valeur() ;}  
    Si une des cases contient une erreur, une CaseErreurException est lancée  
    catch (CaseErreurException e) {  
        throw new CaseErreurException() ;  
    }  
}
```

Méthode eval() dans la classe Soustraction

```
public double eval() throws CaseErreurException {  
    Calcul de la Multiplication  
    try {return gauche.valeur() * droite.valeur() ;}  
    Si une des cases contient une erreur, une CaseErreurException est lancée  
    catch (CaseErreurException e) {  
        throw new CaseErreurException() ;  
    }  
}
```

Méthode eval() dans la classe Multiplication

```
public double eval()  
    throws ArithmeticException, CaseErreurException {  
    double d, g;  
    Si une des cases contient une erreur, une CaseErreurException est lancée  
    try {  
        d = droite.valeur();  
        g = gauche.valeur();  
    } catch (CaseErreurException e) {  
        throw new CaseErreurException();  
    }  
    Si le dénominateur est nul alors une autre ArithmeticException est lancée  
    if (d == 0.0) throw new ArithmeticException();  
  
    return g / d;  
}
```

Méthode eval() dans la classe Division

```
public double eval()  
    throws ListeVideException, CaseErreurException {  
    double res = 0;  
    Si la liste entrée en paramètre est vide alors une exception est lancée  
    if (cases_ope.size() == 0)  
        throw new ListeVideException();  
    Parcours des cases de la liste entrée en paramètre et calcul de la Somme des valeurs de la liste des cases entrée en paramètre  
    for (Case c : cases_ope) {  
        try {  
            res += c.valeur();  
            Si une des cases contient une erreur, une exception est lancée  
        } catch (CaseErreurException e) {  
            throw new CaseErreurException();  
        }  
    }  
    Division de la Somme par la taille de la liste entrée en paramètre  
    return res / cases_ope.size();  
}
```

Méthode eval() dans la classe Moyenne

```
public double eval()
    throws CaseErreurException, ListeVideException {
    double res = 0;

    if (cases_ope.size() == 0) throw new ListeVideException();
    Parcours des cases de la liste entrée en paramètre et calcul de la Somme des valeurs de la liste des cases entrée en paramètre
    for (Case c : cases_ope) {
        try {
            res += c.valeur();
            Si une des cases contient une erreur, une exception est lancée
        } catch (CaseErreurException e) {
            throw new CaseErreurException();
        }
    }
    return res;
}
```

Méthode eval() dans la classe Somme

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

```
public void fixerValeur(double d) {
    La case va contenir maintenant une valeur double d, elle ne contient plus d'erreur
    isCaseErreur = false;
    Affectation de la valeur d
    valeur = d;
    La case ne contient donc pas de formule
    formule = null;
    Elle ne dépend plus d'autres cases et ses prédécesseurs ne l'ont plus en tant que successeur
    delPredecesseur();
    predecesseur.clear();
    On doit alors actualiser les cases qui dépendent de moi
    actualiserValeur();
}
```

Méthode fixerValeur() dans la classe Case

```
private void delPredecesseur() {
    Cette méthode ne s'applique que sur les cases qui contiennent une formule
    if (formule != null) {
        On récupère les cases concernées par la formule
        List<Case> cs = formule.getCasesFormule();
        int i;
        for (Case c : cs) {
            i = 0;
            Parcours des successeurs
            for (Case cc : c.getSuccesseur()) {
                Recherche de la
                if (cc == this) {
                    c.getSuccesseur().remove(i);
                    break;
                }
                i++;
            }
        }
    }
}
```

Méthode delPredecesseur() dans la classe Case

```
private void actualiserValeur() {
    Si la case contient une formule, on doit la recalculer
    if (formule != null) {
        try {
            Calcul de la formule
            valeur = formule.eval();
            isCaseErreur = false;
        }
        Si il y a une erreur lors de l'évaluation de la case, le booléen est mis à jour pour indiquer que cette case est désactiver
        catch (ArithmeticException | CaseErreurException |
        ListeVideException e) {
            isCaseErreur = true;
        }
    }

    Actualisation des successeurs
    for (Case c : successeur) c.actualiserValeur();
}
```

Méthode actualiserValeur dans la classe Case

```
public List<Case> getCasesFormule() {  
    return cases_ope ;  
};
```

Méthode getCasesFormule() dans la classe Fonction

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

```
public void setFormule(Formule f)  
    throws CycleException {
```

S'il y a un cycle, alors il y propagation d'une CycleException

```
    rechercheCycle(f);
```

Suppression de l'objet courant de la liste des successeurs de nos prédécesseurs.

```
    delPredecesseur();
```

Nouvelle formule

```
    formule = f;  
    isCaseErreur = false;
```

Nous vidons la liste des prédécesseurs

```
    predecesseur.clear();
```

ajout des successeurs et prédécesseurs dans les listes correspondantes

```
    addPreSucc();
```

Actualisation des valeurs

```
    actualiserValeur();
```

```
}
```

Méthode setFormule(Formule f) dans la classe Case

```
private void rechercheCycle(Formule f) throws CycleException {  
    List<Case> cs;  
    cs = f.getCasesFormule();  
    for (Case c : cs) {  
        Si l'une des cases vaut l'objet Case courant, on déclenche une CycleException  
        if (c == this) throw new CycleException();  
        Recherche d'un cycle indirect  
        c.rechercheCycleIndirect(this);  
    }  
}
```

Méthode rechercheCycle(Formule F)

```
private void rechercheCycleIndirect(Case racine)
    throws CycleException {
    Parcours des predecesseurs
    for (Case c : predecesseur) {
        Si il contient la racine, lance CycleException
        if (predecesseur.contains(racine))
            throw new CycleException();
        La fonction est réursive
        c.rechercheCycleIndirect(racine);
    }
}
```

Méthode rechercheCycleIndirect(Case racine)

1.4.4. Accès à une case par sa valeur

```
public double valeur() throws CaseErreurException {
    Si la case contient une erreur, une CaseErreurException est lancée car la valeur n'a pas pu être évaluée
    if (isCaseErreur)
        throw new CaseErreurException();
    return valeur;
}
```

Méthode valeur() dans la classe Case

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

Méthode toStringSimple() dans la classe Addition

```
public String toStringSimple() {
    return "(" + gauche.getNom() + " - " + droite.getNom() +
    ")";
}
```

Méthode toStringSimple() dans la classe Soustraction

```
public String toStringSimple() {
    return "(" + gauche.getNom() + " * " + droite.getNom() +
    ")";
}
```

Méthode toStringSimple() dans la classe Multiplication

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


Méthode toStringSimple() dans la classe Division

```
public String toStringSimple() {
    String str = "";

    Si la langue de la grille est française alors le contenu est en français, sinon il sera en anglais
    if (Grille.langue.equals("Francais")) {
        str += "MOYENNE(";
    } else {
        str += "MEAN(";
    }

    if (cases_ope.size() == 0)
        return str + ")";

    Ecriture de tous les noms de cases dans la parenthèses séparés par une virgule
    for (int i = 0; i < cases_ope.size() - 1; i++) {
        str += cases_ope.get(i).getNom() + ", ";
    }
    str += cases_ope.get(cases_ope.size() - 1).getNom();

    return str + ")";
}
```

Méthode toStringSimple() dans la classe Moyenne

```
public String toStringSimple() {

    String str = "";

    Si la langue de la grille est française alors le contenu est en français, sinon il sera en anglais
    if (Grille.langue.equals("Francais")) {
        str += "SOMME(";
    } else {
        str += "SUM(";
    }

    if (cases_ope.size() == 0)
        return str + ")";

    Ecriture de tous les noms de cases dans la parenthèses séparés par une virgule
    for (int i = 0; i < cases_ope.size() - 1; i++) {
        str = str + cases_ope.get(i).getNom() + ", ";
    }
    str += cases_ope.get(cases_ope.size() - 1).getNom();

    return str + ")";
}
```

Méthode toStringSimple() dans la classe Somme

```
public String toString() {
    String str = toStringSimple() ;

    try {
        str += " = " + eval() ;
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException e) {
        str += " = ????" ;
    }
    return str ;
}
```

Méthode toString() dans la classe Addition

```
public String toString() {
    String str = toStringSimple() ;

    try {
        str += " = " + eval() ;
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException e) {
        str += " = ????" ;
    }

    return str ;
}
```

Méthode toString() dans la classe Soustraction

```
public String toString() {
    String str = toStringSimple() ;

    try {
        str += " = " + eval() ;
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException e) {
        str += " = ????" ;
    }
    return str ;
}
```

Méthode toString() dans la classe Multiplication

```
public String toString() {
    String str = "(" + gauche.getNom() + " / " +
droite.getNom() + ")";

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException | ArithmeticException e) {
        str += " = ????" ;
    }
    return str;
}
```

Méthode toString() dans la classe Division

```
public String toString() {

    String str = toStringSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException | ListeVideException e) {
        str += " = ????" ;
    }
    return str;
}
```

Méthode toString() dans la classe Moyenne

```
public String toString() throws IndexOutOfBoundsException {

    String str = toStringSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException | ListeVideException e) {
        str += " = ????" ;
    }
    return str;
}
```

Méthode toString() dans la classe Somme

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

Méthode toStringDevSimple() dans la classe Addition

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

Méthode toStringDevSimple() dans la classe Soustraction

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

Méthode toStringDevSimple() dans la classe Multiplication

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

Méthode toStringDevSimple() dans la classe Division

```

public String toStringDevSimple() {
    String str;
    str = "";
    Gestion de la langue
    if (Grille.langue.equals("Francais")) {
        str += "MOYENNE(";
    } else {
        str += "MEAN(";
    }

    if (cases_ope.size() == 0)
        return str + ")";

    for (int i = 0; i < cases_ope.size() - 1; i++) {
        Si la case utilise une formule, nous opérons la récursion
        if (cases_ope.get(i).getFormule() != null) {
            str +=
cases_ope.get(i).getFormule().toStringDevSimple() + ", ";
        }
        Sinon nous affichons le nom de la case
        else {
            str += cases_ope.get(i).getNom() + ", ";
        }
    }

    Permetts de ne pas finir avec une virgule
    if (cases_ope.get(cases_ope.size() - 1).getFormule() !=
null) {
        str += cases_ope.get(cases_ope.size() -
1).getFormule().toStringDevSimple();
    } else {
        str += cases_ope.get(cases_ope.size() - 1).getNom();
    }
    return str + ")";
}

```

Méthode toStringDevSimple() dans la classe Moyenne

```

public String toStringDevSimple() throws
IndexOutOfBoundsException {

    String str;
    str = "";
    Gestion de la langue
    if (Grille.langue.equals("Francais")) {
        str += "SOMME(";
    } else {
        str += "SUM(";
    }

    if (cases_ope.size() == 0)
        return str + ")";

    for (int i = 0; i < cases_ope.size() - 1; i++) {
        Si la case utilise une formule, nous opérons la récursion
        if (cases_ope.get(i).getFormule() != null) {
            str +=
cases_ope.get(i).getFormule().toStringDevSimple() + ", ";
        }
        Sinon, nous affichons le nom de la classe
        else {

            str += cases_ope.get(i).getNom() + ", ";

        }
    }

    Permetts de ne pas finir avec une virgule
    if (cases_ope.get(cases_ope.size() - 1).getFormule() !=
null) {
        // Si la case utilise une formule, nous oppérons la
récursion
        str += cases_ope.get(cases_ope.size() -
1).getFormule().toStringDevSimple();
    } else {
        // Si non nous afficheons le nom de la classe
        str += cases_ope.get(cases_ope.size() - 1).getNom();
    }

    return str + ")";
}

```

Méthode toStringDevSimple() dans la classe Somme

```
public String toStringDev() {
    String str = toStringDevSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException e) {
        str += " = ????" ;
    }

    return str ;
}
```

Méthode toStringDev() dans la classe Addition

```
public String toStringDev() {
    String str = toStringDevSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException e) {
        str += " = ????" ;
    }

    return str ;
}
```

Méthode toStringDev() dans la classe Soustraction

```
public String toStringDev() {
    String str = toStringDevSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException e) {
        str += " = ????" ;
    }

    return str ;
}
```

Méthode toStringDev() dans la classe Multiplication


```
public String toStringDev() {
    String str = toStringDevSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException | ArithmeticException e) {
        str += " = ????" ;
    }

    return str;
}
```

Méthode toStringDev() dans la classe Division

```
public String toStringDev() {
    String str = toStringDevSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException | ListeVideException e) {
        str += " = ????" ;
    }

    return str;
}
```

Méthode toStringDev() dans la classe Moyenne

```
public String toStringDev() {
    String str = toStringDevSimple();

    try {
        str += " = " + eval();
        Affichage de point d'interrogation en cas d'erreur sur la valeur de la case
    } catch (CaseErreurException | ListeVideException e) {
        str += " = ????" ;
    }

    return str;
}
```

Méthode toStringDev() dans la classe Somme

2. Arborescence du projet

- ❑ ProjetPPO_Laplace_Sauvage
 - ❑ src
 - ❑ noyau
 - ❑ Addition
 - ❑ ArithmeticException
 - ❑ Case
 - ❑ CaseNotFoundException
 - ❑ CycleException
 - ❑ Division
 - ❑ Fonctions
 - ❑ Formule
 - ❑ Grille
 - ❑ ListeVideException
 - ❑ Moyenne
 - ❑ Multiplication
 - ❑ MyTableModel
 - ❑ OperationBinaire
 - ❑ Somme
 - ❑ Soustraction
 - ❑ TablooProto
 - ❑ test
 - ❑ testAddition
 - ❑ testCycle
 - ❑ testDivision
 - ❑ testMoyenne
 - ❑ testMultiplication
 - ❑ testPropagation
 - ❑ testSerialisation
 - ❑ testSomme
 - ❑ testSoustraction
 - ❑ testValeurGrille
- ❑ Laplace_Sauvage_projetPPO_CR1
- ❑ Laplace_Sauvage_projetPPO_CR2
- ❑ Readme.md
- ❑ grilleSujet.dat
- ❑ UML_projetPPO.png

3. Description des tests et résultats

3.1. Test de création des cases de la grille

Lors de ce test (classe *testValeurGrille*), nous avons déclaré chacune des cases correspondantes à celle de l'exemple présenté dans l'énoncé. Ensuite dans une méthode *exempleProjet*, nous avons appelé la méthode permettant d'attribuer une valeur pour les cases contenant des valeurs numériques (A1, A2, A3, A4, B2, C2). Puis nous avons utilisé la méthode permettant de fixer les formules dans le bon ordre pour le reste des cases, pour les fonctions Somme et Moyenne, il fallait faire attention à bien créer des listes de cases pour les rentrer ensuite en paramètres des méthodes.

Nous avons obtenu les mêmes résultats que l'exemple, cela signifie que nos méthodes d'affectation de valeurs et de d'affectation de formule sont correctes.

3.2. Test de propagation lors d'une affectation d'une nouvelle valeur à une case

Lors de ce test (classe *testPropagation*), nous avons repris le code de la classe *testValeurGrille* pour récupérer les données de la grille. Dans une méthode *modificationValeur* nous faisons appel à *fixerValeur* sur une case qui contenait déjà une valeur et dont beaucoup de cases dépendent de cette dernière. Enfin nous vérifions que les cases qui dépendent de la case modifiée ont bien une valeur mise à jour.

3.3. Test de détection de cycle

Lors de ce test (classe *testCycle*), nous également repris le code de la classe *testValeurGrille* pour récupérer les données de la grille, puis dans une méthode *creationCycleDirect* nous modifions la case A1 avec la méthode *setFormule* afin qu'elle soit égale à "A1 + A2", ceci lance bien l'exception *CycleException*. Dans une autre méthode *creationCycleIndirect* nous modifions cette fois-ci la case B2 avec la méthode *setFormule* afin qu'elle soit égale à "A1 + C6", créant alors un cycle indirect car C6 dépend de B6 qui dépend de B2, une erreur est créée et c'est bien celle attendu : *CycleException*.

3.4. Test des méthodes permettant de faire de demander le contenu et le contenu développé d'une case

Lors de ces tests (classes *testAddition*, *testSoustraction*, *testMultiplication*, *testDivision*, *testSomme*, *testMoyenne*), nous vérifions que le contenu et le contenu développé correspondent bien au résultat de l'appel de *toStringSimple* et *toStringDevSimple* pour chacune des classes *Addition*, *Soustraction*, *Multiplication*, *Division*, *Somme* et *Moyenne*.

3.5. Test de sauvegarde, de modification et de chargement de la grille

Lors de ce test, dans un premier temps nous créons une grille que nous sauvegardons. Ensuite nous modifions cette grille. Puis nous rechargeons la grille que nous avons sauvegardée. Enfin nous testons les valeurs de la grille chargée. Elles doivent être égale à l'état de la grille juste avant la sauvegarde et ne doivent pas tenir compte des modifications qu'il y a eu entre la sauvegarde et le chargement.

4. Mode d'emploi

4.1. Exécution du projet sans argument

Pour utiliser notre application, il suffit d'exécuter fichier `tablooProto.java`. Ce dernier utilise le fichier `./grilleSujet.dat` pour générer la grille. L'utilisation de l'interface graphique est intuitive. Il vous suffit de cliquer sur les éléments de contenu ou langue pour modifier le contenu ou la langue. Exemple de d'exécution `./a.out`.

4.2. Exécution du projet avec argument

Si vous donnez en argument à l'exécution du fichier `tablooProto.java` un chemin vers un fichier. Alors ça sera le fichier donné en paramètre qui sera utilisée pour générer la grille. Exemple de d'exécution `./a.out "chemin vers un fichier contenant une grille sérialisée"`.

4.2. Générer un fichier sérialisé

Vous pouvez utiliser la méthode `main` de la classe grille pour générer un fichier d'une grille sérialisée.

5. Bilan

5.1. La réalisation du projet

Nous avons réalisé l'application en respectant la totalité du cahier des charges, nous avons créé un projet Eclipse du nom de `ProjetPPO_Laplace_Sauvage`, puis nous avons programmé chacune des classes présentes dans le schéma UML en prenant soin de créer un package noyau et un package de test qui vérifie les résultats de la figure de l'énoncé et des tests supplémentaires demandés par le tuteur. Nous avons travaillé également avec IntelliJ durant la période de vacances scolaire afin de pouvoir poursuivre notre travail.

De plus, nous avons modifié nos classes pour donner la possibilité de sauvegarder l'état de la grille par sérialisation et nous avons fait en sorte qu'après que toutes les procédures de tests soient OK, l'état du tableur est sauvegardé.

Enfin, nous avons créé une interface graphique sous forme d'un tableau graphique simplifié de manipulation d'une grille chargée à partir d'un fichier sérialisé dont le rendu correspond à la figure 1 de l'énoncé. De plus nous avons ajouté des fonctionnalités en plus à notre interface pour pouvoir utiliser toutes les fonctionnalités que nous avons développées dans le noyau.

5.2. Les fonctionnalités de notre projet

Notre application peut détecter les cycles lorsqu'une formule d'une case en créer un, qu'il soit direct ou indirect. Elle peut également propager les modifications d'une valeur d'une case même si elle génère une erreur. Cependant le code du traitement de la propagation n'est pas optimal, en effet, il se peut que nous repassions plusieurs fois par une même case lorsqu'on actualise les valeurs durant cette propagation ce qui constitue une piste d'amélioration de notre application.

Nous avons réussi à traiter toutes les erreurs possibles qui peuvent arriver volontairement ou non, il en résulte que l'application ne peut s'arrêter de fonctionner lors d'une manipulation qui génère une erreur.

5.3. Bilan personnel

5.3.1. Bilan personnel Jordan Laplace

Lors de ce projet j'ai vraiment pu compléter les compétences que nous avons vues en cours, surtout concernant l'interface graphique. En effet ce projet m'a permis de mieux comprendre le fonctionnement d'une interface graphique et des méthodes qui gravite autour. C'est pour en découvrir toujours plus que nous avons ajouté un menu permettant de changer la langue et la forme du contenu affiché.

5.3.2 Bilan personnel Maxime Sauvage

Le projet était complet, il a repris tout ce qu'on a vu en cours et m'a permis de mettre en application ce qui n'avait pas été fait lors des TP, notamment l'interface graphique, la sérialisation et les tests unitaires qui s'avèrent nécessaires afin d'être sûr que nos méthodes fonctionnent bien, le temps donné pour le faire est convenable car nous avons réussi à le terminer.