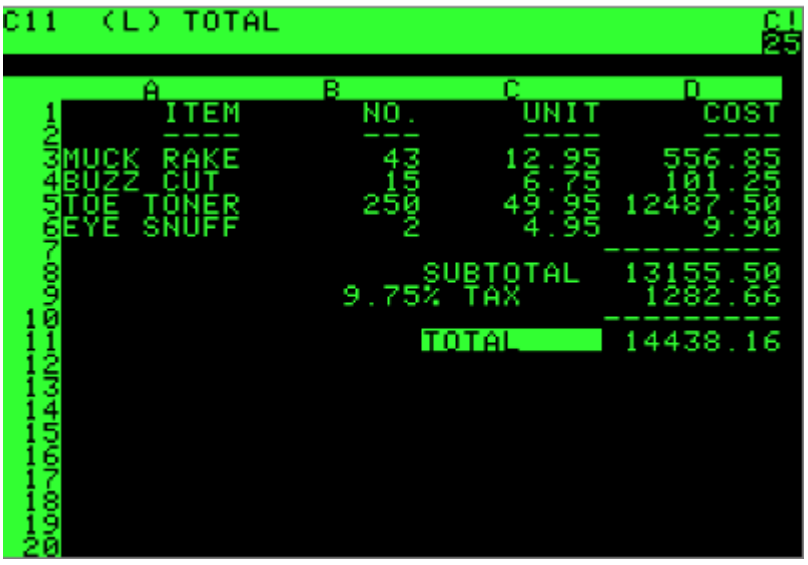
# Situation d’Apprentissage et d’évaluation

Alexis Wamster

Djabrail Khapizov

Mikhail Anani



# Sommaire

# Introduction

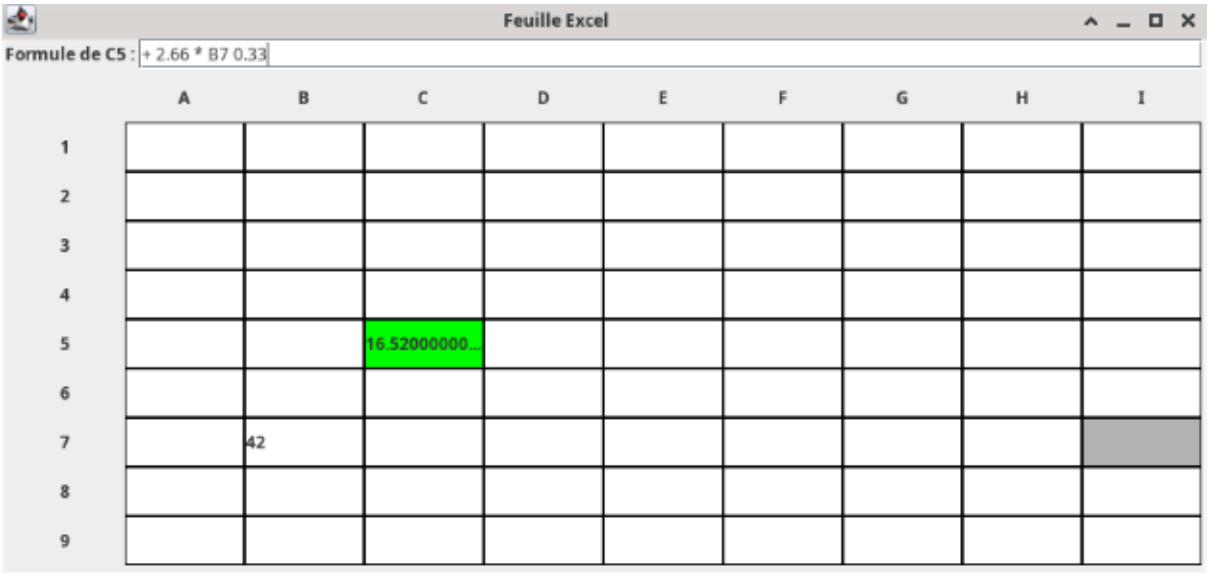
Un tableur est un outil informatique fascinant qui permet d'organiser et de manipuler des données, généralement numériques, dans une grille bidimensionnelle. Chaque cellule de cette grille peut contenir soit une donnée brute, soit une valeur calculée dynamiquement en fonction du contenu d'autres cellules. Cette simplicité conceptuelle, combinée avec une puissance de calcul impressionnante, a révolutionné la manière dont les données sont traitées et présentées dans le monde professionnel et éducatif.

Nous nous penchons sur un jalon historique dans l'évolution des tableurs : VisiCalc. Lancé comme le premier tableur, VisiCalc a ouvert la voie à un tout nouveau genre d'outils informatiques, changeant radicalement la gestion des données.

Dans cet esprit d'innovation, notre projet consiste à développer une version simplifiée d'un tableur. Notre objectif est de créer une feuille de calcul comportant 9 lignes et 9 colonnes, capable de manipuler uniquement des nombres réels à l'aide d'une sélection restreinte d'opérateurs.

# Description des fonctionnalités

Notre feuille de calcul sera divisée en 81 cellules, réparties en 9 colonnes (indexées de A à I) et 9 lignes (indexées de 1 à 9). Chaque cellule contiendra une formule initialement vide, que l'utilisateur pourra modifier. Ces formules peuvent inclure des constantes réelles, des références à d'autres cellules, et des opérateurs arithmétiques, avec une syntaxe en notation préfixe. Par exemple, "+ 2.66 * B7 0.33" est une formule syntaxiquement correcte.



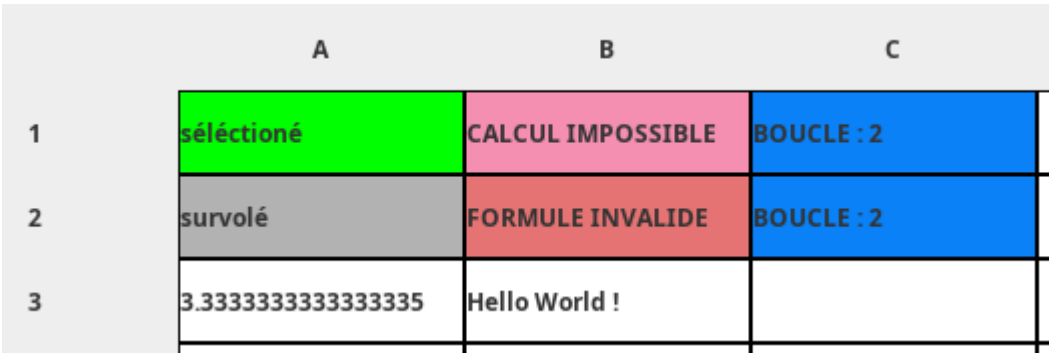

Nous distinguerons visuellement les quatre états possibles pour chaque cellule :

- Vide
- Contenant une formule correcte et calculable
- Contenant une formule correcte mais incalculable
- Contenant une formule incorrecte.

Une gestion intelligente des formules évitera les références circulaires et les divisions par zéro, garantissant ainsi la validité et la calculabilité des données.

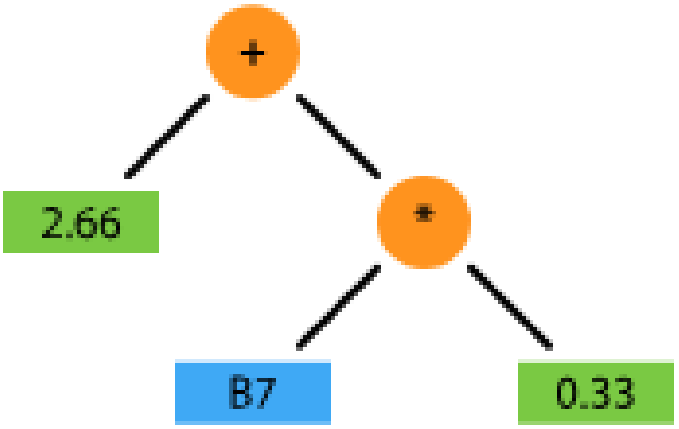
Nous avons également ajouté des visuels supplémentaire:

- La cellule sélectionnée sera verte
- Celle que l'on survole sera grisée
- Les cellules qui comportent un problème circulaire seront visuellements différentes de celles qui ont une formule correcte mais incalculable (une couleur aléatoire avec un numéro de boucle)
- Enfin, il y a la possibilité d'écrire du texte. (Le visuel d'un texte est identique à celui d'une formule)



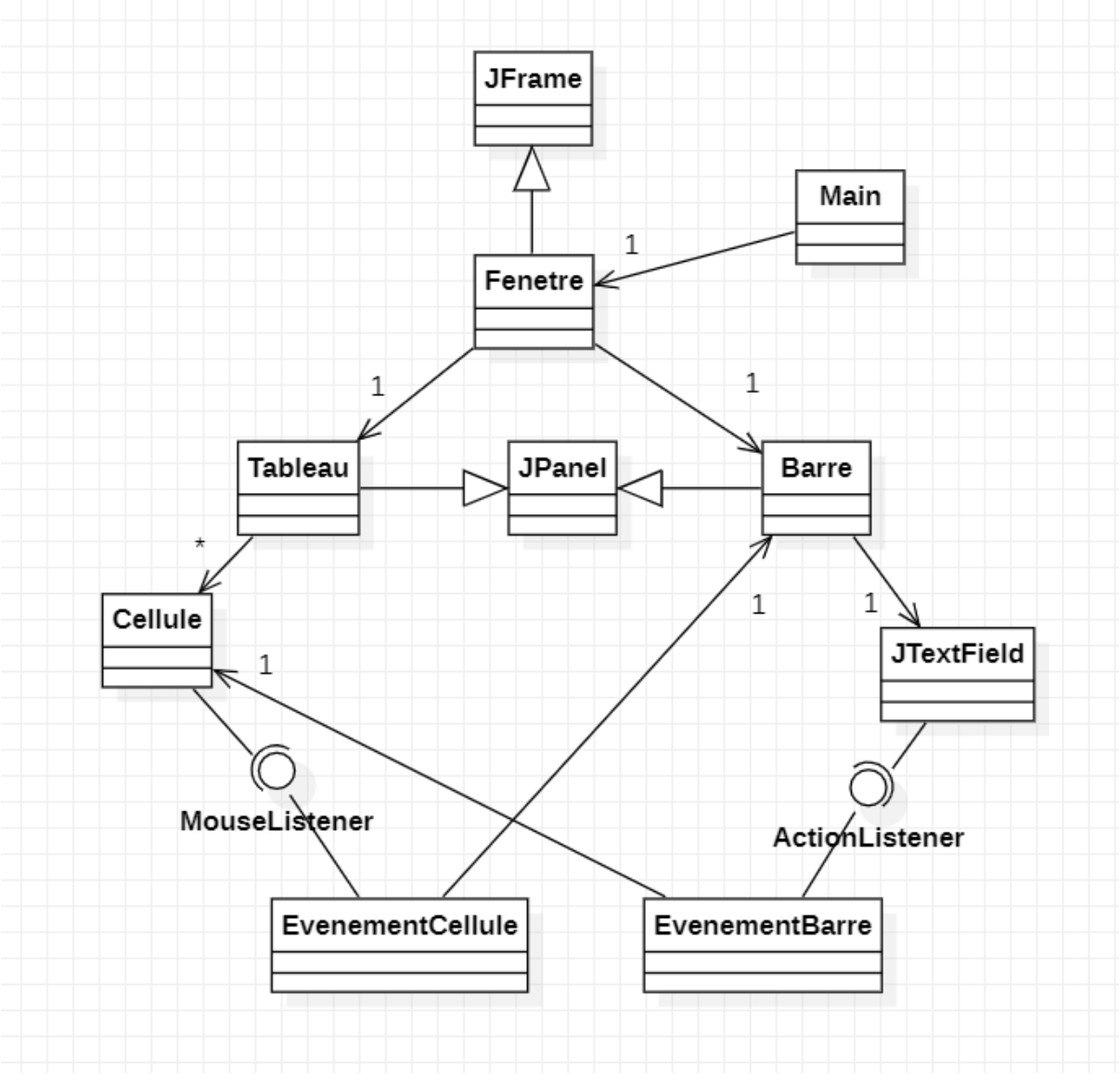

Les formules, potentiellement longues et complexes, ne seront pas affichées directement dans la feuille de calcul. Seuls les résultats des calculs seront visibles pour les cellules avec des formules correctes et calculables. Une zone d'édition dédiée permettra la modification des formules, avec un système de mise à jour en temps réel pour la cellule en cours d'édition et pour les cellules dépendantes.


Enfin, nous adopterons une approche de programmation avancée, en conservant une représentation secondaire des formules sous la forme d'un arbre de syntaxe abstraite. Cette méthode optimise non seulement les performances en évitant des réinterprétations répétées mais permet également une vérification efficace de la syntaxe


+
2.66
*
B7
0.33

# Présentation de la structure du programme

## Construction d'une fenêtre




La classe principale qui lance le programme (**Main**) crée et affiche une Fenêtre.

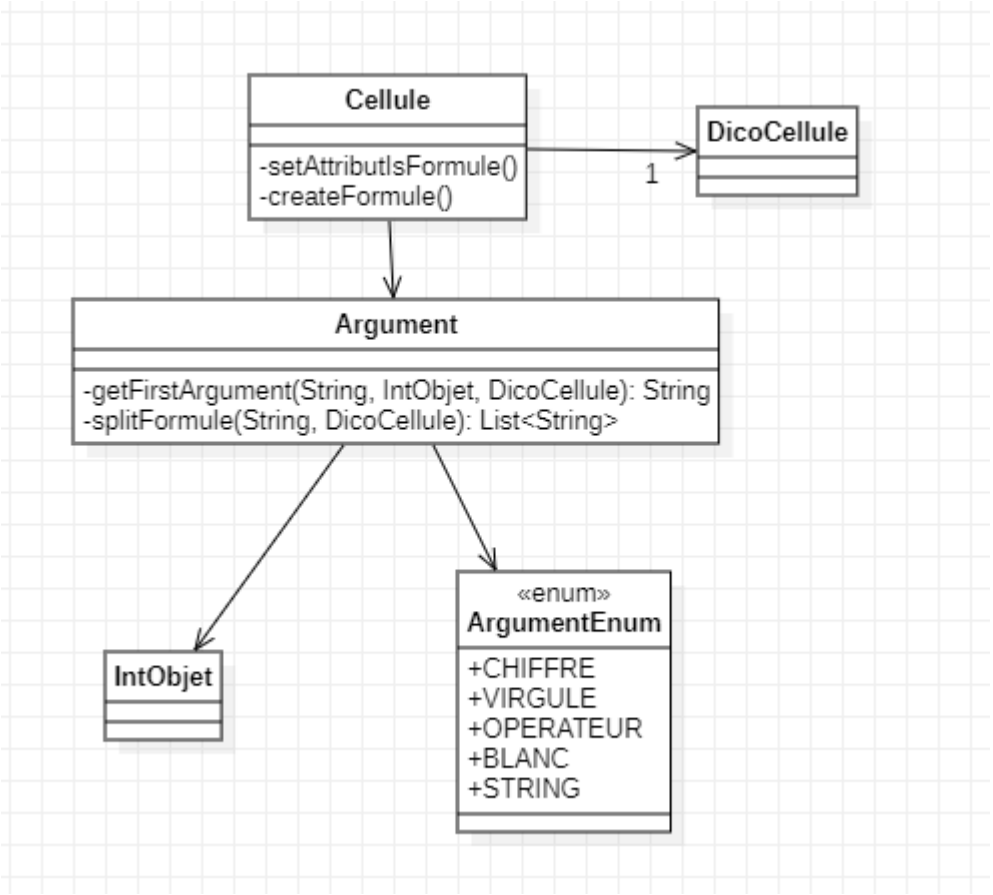
Ainsi, la classe **Fenêtre** représente une fenêtre dans l'interface utilisateur en héritant de **JFrame**.
Dans ce programme une fenêtre est composée de deux éléments majeurs : un espace pour saisir une formule et un tableau de cellules.

Ces deux éléments, **Tableau** et **Barre**, sont des classes qui héritent de **JPanel**.

La classe **Barre** contient un **JTextField** dans laquelle l'utilisateur va saisir une formule. Ce **JTextField** a un écouteur d'événements **EvenementBarre** qui implémente l'interface **ActionListener**. Cet écouteur d'évènement à pour but de mettre à jour le contenu d'une cellule et de toutes celles qui en dépendent dès que l'utilisateur saisit une formule dans le **JTextField**. **EvenementBarre** doit donc connaître la cellule qu'il met à jour.

La classe **Tableau** contient un certain nombre de cellules (normalement 81). Chacune de ces cellules est un objet de la classe **Cellule** qui gère ses propriétés et affiche le résultat d'une formule. Pour cela elle hérite de la classe **JLabel**. Elles ont toutes le même écouteur d'événement : **EvenementCellule** qui implémente l'interface **MouseListener**. Le but de cet écouteur est de griser les cellules survolées, et de changer la couleur de la cellule sélectionnée pour plus d'ergonomie. Cela met aussi à jour la cellule sélectionnée dans **Barre** afin que le changement de formule affecte la bonne cellule.

## Détection d'une formule et de ces composants




La classe **cellule** contient de nombreuses méthodes pour gérer ses données et certaines permettent de vérifier si une chaîne de caractère est une formule (**setAttributIsFormule**) et de créer un arbre avec les composants de la formule à partir de cette chaîne (**createFormule**).

Il y a 3 cas dans lesquels l'algorithme détecte une formule dans une chaine de caractere:

- La chaîne commence par un nombre (ex: "66.3* 2" , ".5", "7C+3")
- La chaîne commence par un opérateur (ex: "**3", "-8", "/A3B2")
- La chaîne commence par un nom de Cellule (ex:"B5", "C2-C8", "C4cds4f56s")

*Les exemples ci-dessus ne sont pas tous des formules valides. Il sont simplement considéré comme une formule par l'algorithme

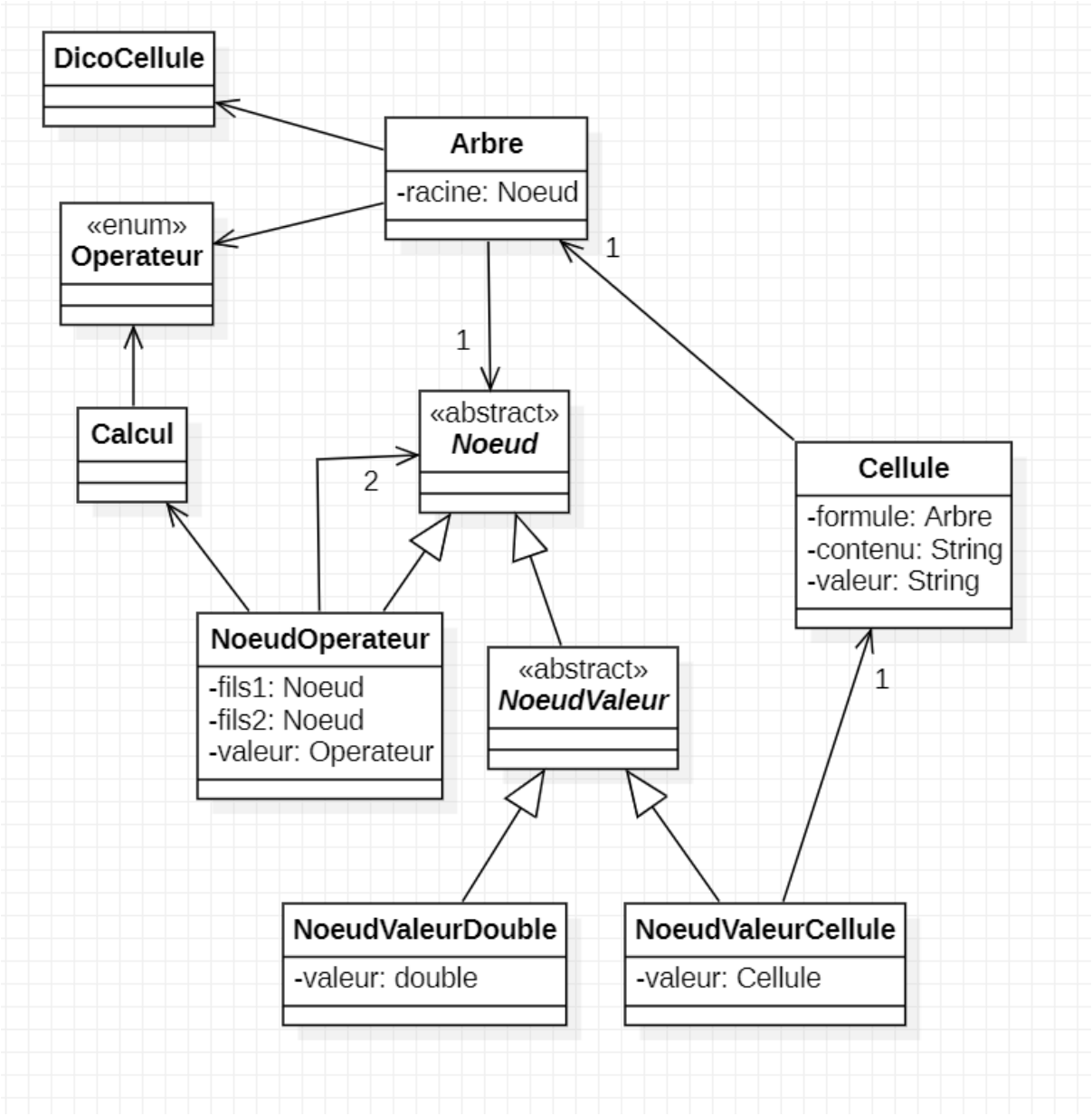
Pour détecter si un nom de cellule est valide, chaque cellule contient un objet de la classe **DicoCellule** qui permet de retrouver un objet Cellule à partir de son nom.

Pour trouver le premier composant d'une formule, une cellule utilise une méthode statique de la classe **Argument** (**getFirstArgument**).
Pour obtenir la liste des composants de la formule, une cellule utilise une autre méthode statique dans Argument (**splitFormule**).

La classe **Argument** est donc utilisé pour détecter et séparer les composants d'une formule écrite dans une chaîne de caractère.
Elle utilise un type énuméré **ArgumentEnum** qui permet d'indiquer si un caractère est plutôt: un chiffre, une virgule, un opérateur, un caractere blanc ou autre.
Elle utilise également la classe **IntObjet**. qui est un entier capable de changer à l'intérieur d'une méthode (comme les pointeurs en C). Par exemple, la méthode getFirstArgument renvoie deux choses le premier composant de la formule mais également le nombre de caractères qui ont été utilisés dans ce premier composant. (Avec la formule "  A4sfd", le premier composant est "A4" et nombre de caractère utilisé est 6 (4 caractere blanc et les deux de "A4"))

# Explication des classes participant à l'arbre de syntaxe abstraite et un diagramme d'objets

**Diagramme de classe**



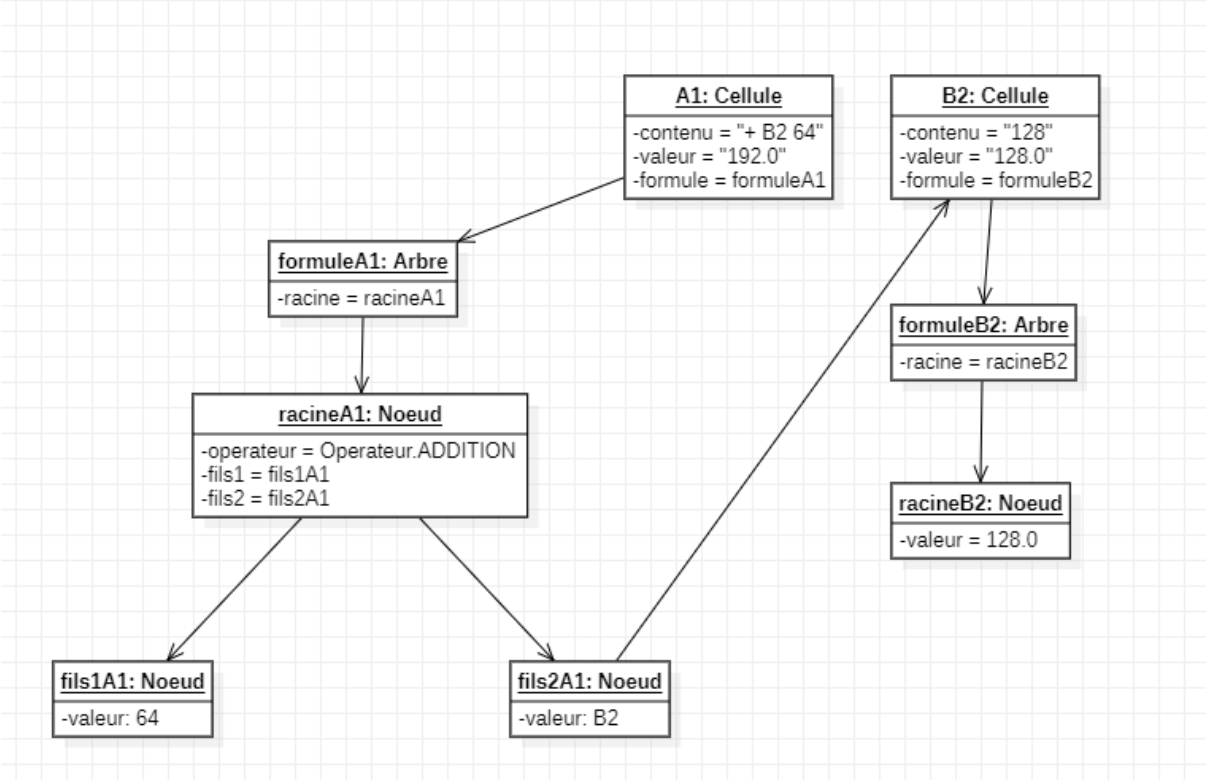

Chaque objet de type **Cellule** contient sa formule sous forme d'arbre. **Arbre** est une classe qui connait un nœud racine interconnecté à d'autres nœuds. Elle est capable de créer un nœud et de l'ajouter à ses branches.

Un nœud peut être un opérateur. donc la classe **Arbre** utilise le type énuméré **Operateur** ainsi que la classe **Calcul** qui contient une méthode pour savoir si un String est un opérateur.
Un nœud peut aussi être une référence vers une autre cellule. **Arbre** utilise donc aussi la classe **DicoCellule** pour convertir un String qui contient le nom d'une cellule en une référence vers un véritable objet de type **Cellule**.

- La classe **Noeud** est abstraite. En effet, un **Noeud** peut contenir soit un opérateur, soit une valeur finale. D'où les deux classes **NoeudOperateur** et **NoeudValeur** qui héritent de Noeud. Un opérateur doit effectuer un calcul entre deux nombres. Ainsi, **NoeudOperateur** a deux références vers deux autres nœuds.
- Une valeur, elle, est plutôt une feuille de l'arbre et ne contient donc aucune référence vers un autre nœud. Il existe deux types de valeur dans notre programme : les réels et les références vers une cellule. Ainsi, la classe abstraite **NoeudValeur** est aussi divisée en deux : **NoeudValeurDouble** et **NoeudValeurCellule**.

## Diagramme d'objet

Voici un diagramme objet qui illustre le diagramme de classe précédent



# Exposition de l'algorithme qui détecte les références circulaires

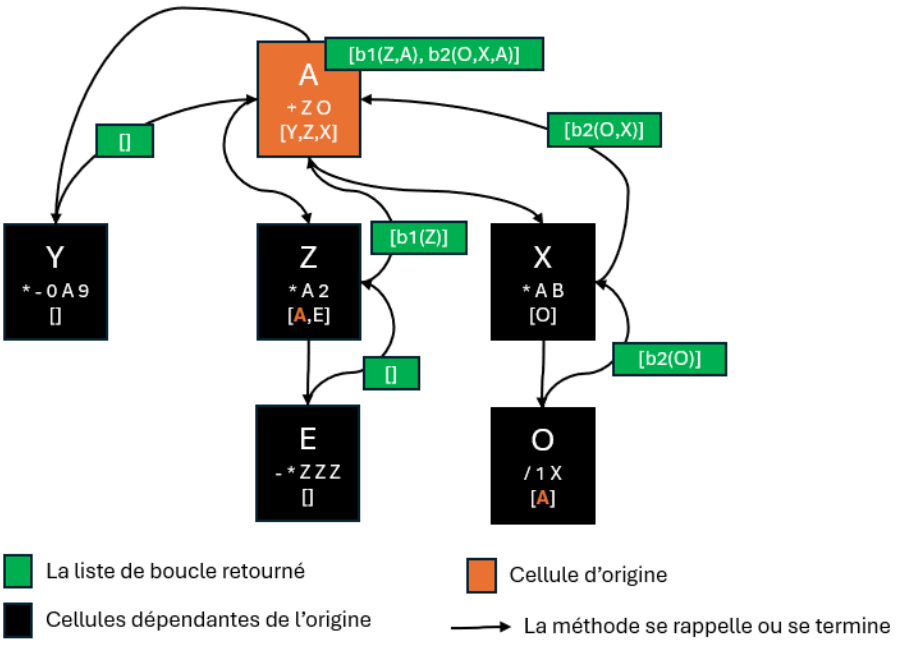
Pour détecter une référence circulaire, il est d'abord nécessaire de comprendre comment fonctionnent les références.

**Référence simple**

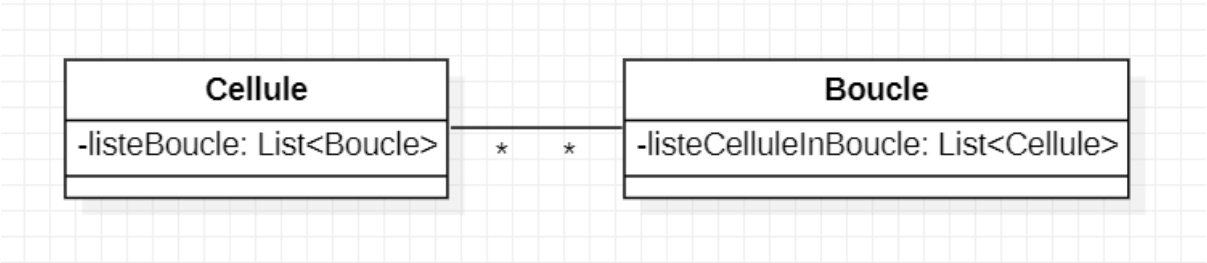
Chaque cellule A contient la liste des cellules qui dépendent directement de A.
Lorsque je modifie A, les cellules qui dépendent de A doivent aussi être mise à jour. Grâce à cette liste, A connais les cellules à mettre à jour après sa modification.

Lorsque A est modifié, cette liste de dépendance est parcourue récursivement de manière à modifier les cellules qui dépendent directement de A mais aussi les cellules qui en dépendent indirectement. (On parcourt la liste des dépendances des cellules inscrites dans la liste de dépendance de A et récursivement)

**Référence circulaire**



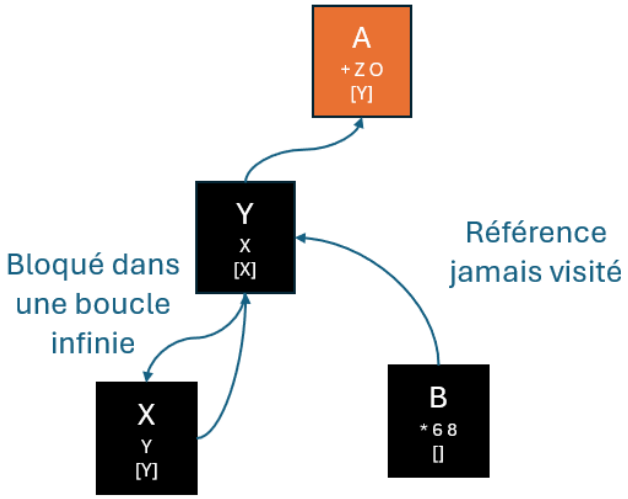
La détection de référence circulaire se fait dès qu'on modifie une cellule A.
C'est la méthode récursive setListBoucle() qui a ce rôle. Elle prend la cellule A qui vient d'être modifiée en argument.
La liste contenu dans A (Y,Z,X) est parcourue récursivement. (Comme un arbre en profondeur voir les flèches du schémas).
Chaque cellule trouvée dans la liste est comparée avec la cellule d'origine A.
Si elles sont identiques (c'est le cas dans la cellule Z et la cellule O), un objet boucle est créé.
On ajoute ensuite à cet objet la cellule qui est en cours de vérification. (Dans la boucle b1, Z est ajouté car Z contient A dans sa liste)
Et on ajoute la boucle à cette Cellule afin que la cellule puisse savoir à quelles boucles de références elle appartient
La liste continue d'être parcourue et lorsqu'on arrive à la fin de la liste, la méthode renvoie tous les objets boucles qui ont été créés.
A chaque étage remontés par la récursion, on ajoute la cellule actuellement visitée aux boucles renvoyées et on ajoute la boucle à la cellule. De cette manière la boucle se remplit de toutes les cellules appartenant à la même référence circulaire.
(ex: la boucle b2 se remplit d'abord de O puis lorsque la récursion remonte elle se complète de X et enfin de A)




Ainsi une cellule connaît les boucles dans lesquelles elle est impliquées, et la boucle connaît les cellules qui en font partie

**<u>complexification</u>**
Mais avec la méthode ci-dessus, on peut rencontrer un problème.
Exemple:On crée une boucle entre X et Y avec X qui dépend d'une cellule A.

Lorsqu’on met une nouvelle formule dans A, toutes ces dépendances vont être visitées en profondeur.




D’après cet exemple, la vérification se fera de manière infinie avec certaines dépendances jamais visitées.

Pour pallier ce problème, dès que l’algorithme tombe sur une cellule qui est incluse dans au moins une boucle, on empêche l’algorithme de la revisiter. (On passe donc à la méthode récursive les boules déjà visitées)

**Création de la liste des références**

Pour fabriquer la fameuse liste de références, voici comment ça se passe. Lorsqu'un utilisateur saisit une formule dans une cellule A et que celle-ci est syntaxiquement correcte, un arbre est créé. (Dans cette formule il peut y avoir des références vers des cellules B) On parcourt l’arbre, et pour chaque référence vers B, on ajoute à la liste de dépendance de B, la cellule A.

# Structures de données abstraites vues en cours

**Arbre**

Un arbre est utilisé pour stocker les formules valides des différentes cellules.
Dans cette structure, les feuilles sont obligatoires des doubles ou des cellules, et les branches des opérateurs.

Pour effectuer les calculs ou ajouter des éléments, l'arbre est parcouru en profondeur et utilise une **pile**. (On utilise la pile de la mémoire du système grâce à la récursivité)

**Liste**

Les listes sont utilisées à plusieurs reprises dans le programme. En tant que tableau à limite indéterminée, et également pour supprimer des éléments bien spécifiques de cette liste sans forcément connaître son indice.
Voici quelques exemples d'utilisation:

- Pour stocker les cellules qui dépendent directement d'une autre
- Pour diviser le String qui contient la formule en une liste de String correspondant aux différents composants de la formule.
- Pour stocker toutes les cellules participants à une même boucle de référence circulaire
- …

(on ne sait pas à l'avance combien il y en aura)

**Dictionnaire**

Le dictionnaire permet de retrouver un objet Cellule (avec toutes ses infos) juste en connaissant son nom (ex : "B3")
Très pratique pour effectuer les références.

# Conclusion

## Djabrail

Pour ma part, au début, ce projet me semblait plutôt complexe. C'est pourquoi j'ai proposé de nous y mettre dès le début, plutôt que de retarder les choses. Cette approche s'est avérée bénéfique, car lorsque nous pensions avoir fini et après avoir effectué de nombreux tests, nous avons constaté pas mal de bugs. Il a donc fallu revoir la configuration de certaines classes et en créer de nouvelles pour corriger les bugs et éviter d'en créer d'autres. La collaboration avec mes camarades m'a été bénéfique, car ils m'ont expliqué certains aspects sur lesquels j'avais des difficultés. Ainsi, ce travail d'équipe m'a aidé à avoir une approche différente des problèmes et de leurs résolutions. J'ai donc acquis une expérience supplémentaire.

## Mikhail

Pour conclure, ce projet de création d'un petit tableur a été une réussite notable pour notre équipe. Nous avons fait preuve d'une bonne capacité à travailler ensemble, ce qui a été essentiel pour surmonter les problèmes rencontrés. La gestion efficace des problèmes rencontrés témoigne de notre progrès en matière de travail d'équipe et de résolution de problèmes. De plus, notre organisation tout au long du projet a été un facteur clé de notre succès. Cette expérience a renforcé nos compétences collaboratives et notre aptitude à gérer des situations complexes. J'en retiens beaucoup de leçons , d'expériences, ça a été un vrai plaisir de travailler sur ce sujet avec ce groupe. On s'est beaucoup entraidé, il y a eu beaucoup de réflexion que se soit sur les problèmes circulaires ou autres. Ce projet était super intéressant puisque l'on a pu bien appliquer ce qu'on a vu dans les différents cours(abstraction,arbre,etc..).

## Alexis

Ce projet était très intéressant pour moi car pour la première fois j'ai pu utilisé les abstractions en sentant réellement que c'était nécessaire (même si je savais déjà que ce n'était pas inutile).
Le fait d'être dans un groupe qui se lance rapidement dans le projet a renforcé en moi l'esprit de cohésion. Je ne me suis pas senti seul durant l'avancée du projet.
Cette avance que l'on avait à aussi permis de se consacrer un peu plus à la correction des bugs qu'on ne voit pas aux premier abord. Par exemple: pour les références circulaires, nous avons très vite trouvé une solution mais qui a laissé place à quelques bugs.
Comme dans les autres projets, il y a eu des sujets qui ont nécessité beaucoup de réflexion avant de parvenir à une solution optimale, ce qui m'apporte encore un peu d'expérience.