Rapport de projet long – Itération 3

Equipe EF-3 : AUGEREAU Robin, BELAHRACH Safae, DO COUTO VIDAL Barbara, DEORA Axel, EL ASRI Fatima Zahra, GRAVIER Amandine, HUANG Julien, TARRADE Antonin.

Département Sciences du Numérique - Première année
2022-2023

Table des matières

Table des figures

1 Introduction

Le projet a pour but de faciliter la création de raffinages en offrant une solution pratique et efficace. La construction d'un raffinage est une étape importante lors de la conception d'un programme, car cela permet de décomposer un problème difficile et abstrait en sous-problèmes plus simples que l'on sait résoudre.

Cependant, il peut être difficile de représenter un raffinage de manière structurée et lisible. La solution proposée par le projet est de créer un logiciel d'assistance qui permettra de générer automatiquement des tableaux structurés et formatés contenant les raffinages. Ce logiciel sera doté d'une interface graphique facile à prendre en main permettant à l'utilisateur d'entrer du texte ainsi que de choisir des options de formatage.

L'interface graphique propose donc des options de mise en page pour faciliter la lecture et la compréhension des raffinages.

L'objectif final du projet est de permettre aux utilisateurs de créer des raffinages de manière plus rapide, plus facile et plus efficace, tout en garantissant leur qualité et leur lisibilité. Le logiciel d'assistance offre également une solution optimisée pour la représentation des raffinages, permettant ainsi une meilleure gestion du flot de données.

2 Première itération

Nous avons réfléchi à comment faire pour avoir une version de l'application utilisable le plus rapidement par l'utilisateur, même si toutes les fonctionnalités ne sont pas implémentées. C'est dans cette optique que nous avons décidé, pour cette première itération, d'avoir une interface graphique en deux colonnes : la première pour l'édition des raffinages et la seconde pour l'arborescence des raffinages, comme vous pourrez le voir sur le manuel utilisateur. Nous avons fait le choix de pouvoir ajouter seulement une structure de contrôle ("si") ainsi que les deux types d'actions possibles : l'action complexe (raffinage) et l'action élémentaire. Ainsi, à la fin de cette première itération, notre application est déjà utilisable.

3 Deuxième itération

Pour cette seconde itération, nous avons décidé d'améliorer notre application en complétant les fonctionnalités implémentées. Nous avons donc ajouté des options de présentation du texte et nous avons créé de nouveaux boutons permettant de faciliter l'utilisation de l'application. Nous avons également ajouté de nouvelles structures de contrôle utiles au raffinage.

4 Troisième itération

Pour cette troisième et dernière itération de notre projet, nous avons finalisé certaines fonctionnalités qui nous paraissaient primordiales pour le bon fonctionnement de notre application. Nous n'avons donc rien rajouter de nouveau, mais nous avons amélioré et fini certains aspects de notre application.

5 Les principales fonctionnalités

— Options de formatage de texte : Le logiciel doit proposer des options de formatage de texte, telles que la police, la taille et le style de texte pour améliorer la présentation des raffinages.

Avancement à l'itération 1 :
→ La taille du texte peut-être augmentée ou diminuée d'un clic sur le bouton ZoomOut ou ZoomIn. La police a été changée pour être en monospace, un format plus agréable pour le

code, et la coloration du raffinage est en cours de mise en place.

Avancement à l'itération 2 :
→ La police du texte peut-être modifiée à l'aide du bouton Police. Celle-ci sera choisie parmi une bibliothèque de polices en monospace, et change l'intégralité du texte présent sur le panel. La coloration du raffinage est en cours de mise en place.

Avancement à l'itération 3 :
→ La possibilité de voir si une actions complexe a été complété ou non.

— Couleur pour chaque niveau de raffinage : Le logiciel doit permettre à l'utilisateur de visualiser l'avancement de son raffinage :
Rouge : L'action complexe courante n'est pas raffinée
Orange : Les actions complexes ne sont pas toutes raffinées
Vert : Toutes les actions complexes sont raffinées *ie* les derniers niveaux de raffinage ne présentent que des action élémentaires.

Avancement à l'itération 1 :
→ Création d'un arbre de raffinage (Partie gauche) permettant à l'utilisateur d'ordonner sa conception. En outre, celui-ci peut ajouter ou supprimer des raffinages directement depuis l'arborescence. Les couleurs ne sont pas encore implémentées.

Avancement à l'itération 2 :
→Lors d'un clic sur le bouton 'ActionComplexe', l'action est non seulement ajoutée au raffinage courant, mais un nouveau niveau de raffinage et créé automatiquement

Avancement à l'itération 3 :
→ Tout fonctionne correctement, y compris les couleurs.

— Options pour les structures de contrôle : Des options spéciales pour les structures de contrôle telles que les boucles, les conditions et les instructions de contrôle de flux doivent être incluses pour faciliter la création des raffinages.

Avancement à l'itération 1 :
→ L'utilisateur peut ajouter à sa guise des structures de contrôles les unes après les autres ; Il peut en ajouter plusieurs parmi celles implémentées.

Avancement à l'itération 2 :
→ De nouvelles structures de contrôles ont été implémentées (Pour, Tantque et Répéter). Les structures plus complexe ont été démarré également (SwitchCase, SiSinon).

Avancement à l'itération 3 :
→ Possibilité d'ajouter des éléments a l'intérieur même d'une structure de contrôle. A noter : la structure SiSinon n'est toujours pas fonctionnelle.

— Options pour les actions élémentaires et complexes : Des options pour la création et l'affichage des actions sur la vue édition et l'ajout de nouveaux niveaux de raffinage à l'arbre des raffinages.

Avancement à l'itération 1 :
→ L'utilisateur peut ajouter les niveaux de raffinage manuellement à l'arbre en saisissant les

titres de chaque sous-fils (noeud).
→ La création des interfaces publiques Action, Element, des classes ActionComplexe et ActionElementaire dont les méthodes implémentées et documentées ont été utiliser lors de la deuxième itération.

Avancement à l'itération 2 :
→ L'utilisateur peut ajouter à son raffinage une action en cliquant sur les boutons dédiés et les afficher sur l'interface (Des observateurs ont été implémentées pour ce but (ActionComplexeListener et ActionElementaireListener)). Le lien entre l'arbre et le clic sur les boutons était établi -> Ajout automatique des actions complexes sous forme de sous-raffinages (nouveaux niveaux) du raffinage courant .

Avancement à l'itération 3 :
→ Le curseur est pris en compte, l'utilisateur peut donc ajouter des élements dans une structure (pour cela il faut cliquer sur le titre de la structure), ou a la suite d'un element (Cliquer sur l'élément, ou le Fin.. pour une structure).

— Possibilité d'ajouter des commentaires : Le logiciel doit permettre à l'utilisateur d'ajouter des notes supplémentaires à chaque étape du raffinage pour aider à la compréhension et à la maintenance du code.

Avancement à l'itération 1 : **non implémenté.**

Avancement à l'itération 2 : **non implémenté.**

Avancement à l'itération 3 : **non implémenté.**

— Option de sauvegarde et de chargement : L'utilisateur doit pouvoir sauvegarder et charger des raffinages précédemment créés pour une utilisation ultérieure. Les raffinages doivent être enregistrés dans un format facilement accessible.

Avancement à l'itération 1 : **non implémenté.**

Avancement à l'itération 2 :
→ Il est possible de choisir l'emplacement de la sauvegarde à l'aide du bouton Enregistrer-Sous. Cependant il n'est pas encore possible de sauvegarder son travail en format .JSON.

Avancement à l'itération 3 :
→ Tentative (spike) avec la librairie GSON et le nouveau format des raffinages (Balises), qui fut un échec.

— Exportation des raffinages : L'utilisateur doit pouvoir exporter les raffinages dans différents formats, tels que PDF, texte brut, HTML ou autre, pour une utilisation ultérieure. Les exportations doivent être de qualité et facilement partageables, incluant des formats répandus tels que Latex.

Avancement à l'itération 1 : **non implémenté.**

Avancement à l'itération 2 : **non implémenté.**

Avancement à l'itération 3 :

→ Il est maintenant possible d'exporter son raffinage au format .tex.

— Génération des flots de données : Le logiciel doit être en mesure de générer des flots de données automatiquement à partir des raffinages écrits par l'utilisateur.

Avancement à l'itération 1 : **non implémenté**

Avancement à l'itération 2 : **non implémenté.**

Avancement à l'itération 3 : **non implémenté.**

6 Les diagrammes de classe

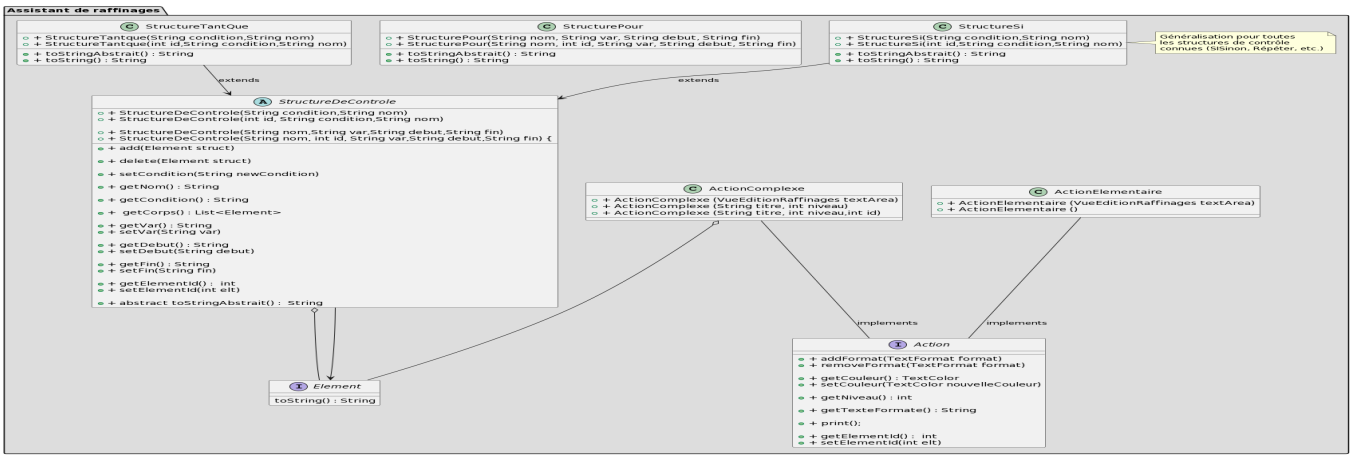

FIGURE 1 – Diagramme UML

6.1 Découpage en sous-systèmes

Dans notre projet, on a utilisé le patron de conception (design pattern) Composite vu qu'il propose une structure arborescente permettant d'implémenter avec la même interface Element sur les feuilles et les composites afin qu'ils soient manipulés de la même manière c'est à dire de manière uniforme.
Explication des choix mentionnés dans le diagramme :

1. Interface structure Element avec la méthode toString() :
L'interface "Element" représente une structure de base pour les éléments d'un raffinage (Soit Action soit StructureDeControle) . Elle définit une méthode "toString()" qui permet selon le choix entre les structures de contrôle et le choix entre action élémentaire ou complexe affichera une représentation textuelle de l'élément sur notre interface .

2. Relation d'agrégation entre Element et l'interface Action :
La relation d'agrégation permet le partage : le même objet (Element dans notre cas) peut appartenir à plusieurs liens d'agrégation (soit action soit structure) , cette relation indique ainsi qu'un élément peut contenir une référence vers un objet qui implémente l'interface "Action » et que cet élément peut effectuer une ou plusieurs actions définies dans l'interface "Action".

3. Les classes ActionComplexe et ActionElementaire héritent de l'interface Action, ils sont des

implémentations concrètes de l'interface "Action".

4. La classe "StructureDeControle" implémente l'interface "Element" vu qu'un élément peut être une structure de contrôle.

5. Utilisation du patron de conception stratégie : « StructurePour », « StructureSi », « StructureSiSinon », « StructureRepeat », « StructureTantque » étendent « StructureDeControle », sont des sous-classes de la classe et peuvent ajouter des fonctionnalités supplémentaires spécifiques à chaque type de structure de contrôle.

En conclusion, ce diagramme UML montre une hiérarchie de classes et d'interfaces liées à la structure et au comportement des éléments de notre Assistant aux raffinages.

7 Principaux choix de conception et de réalisation

7.1 L'interface graphique

Menu et barre d'outil : mis en place la barre de menu, la barre d'outil et les raccourcis clavier. Avancement : seulement les boutons "classique" tel que save, open, save as... (voir 2) Rien ne se passe encore lorsque l'on clique sur les boutons.

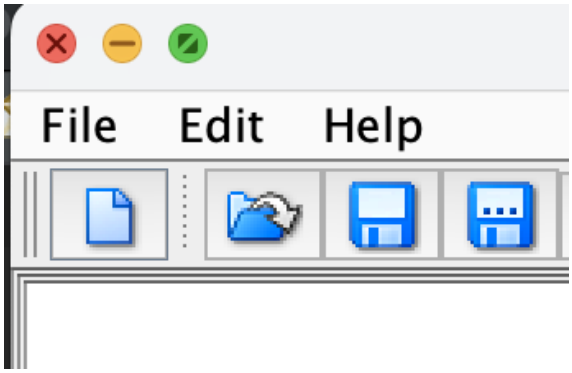

FIGURE 2 – Menu

Boutons : Tant Que, Répéter, Pour : dans une phase de test.

Pour les boutons "actions élémentaire", "action complexe" : elles ne fonctionnent pas encore. D'autre boutons comme supprimer des actions ou structure de contrôle sont à mettre. Organisation de l'interface utilisateur : le "splitage" des différentes fenêtre comme on peut le voir dans la figure 3

7.2 L'édition de raffinages

La section d'édition des raffinages prendra la forme d'un IDE modulaire, hybride "clic-texte", dans le sens où ajouter une structure de contrôle se fait par un clic sur un des boutons (en bas à gauche sur la figure 3).

Pour l'instant, l'objectif est que les mots-clés de structure s'affichent en rouge, et que les mots-clés de condition soient en bleus, ainsi que le reste des actions simples ou complexes.

Exemple :

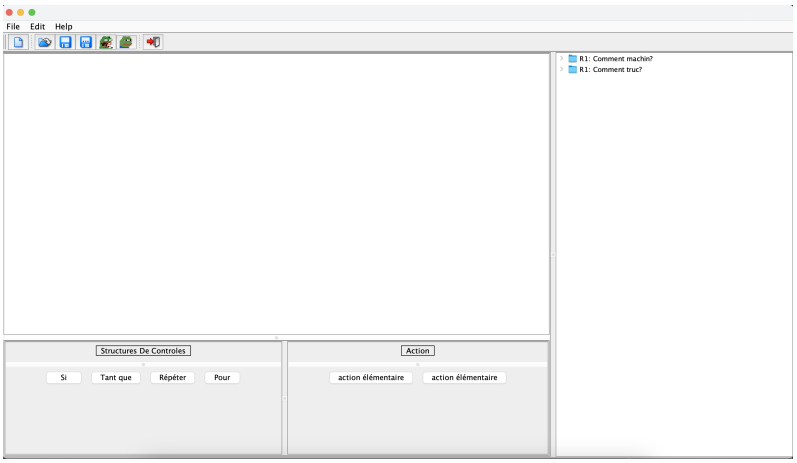

FIGURE 3 – interface graphique

```
Si(1==2)Alors:
        ActionComplexe
FinSi
```

FIGURE 4 – Structure de contrôle, avec une action complexe à réaliser.

Dans ce cas, le mot-clé FinSi n'est, pour l'instant, pas encore correctement reconnu, mais la condition et **Si** ainsi que **Alors** sont en rouge.

De plus, le logiciel devra reconnaître totalement les mots-clés, et à quelle structure ils appartiennent, ou la position dans le code à laquelle ils sont. En l'occurence, cliquer sur un mot-clé permet de découvrir son identifiant, son type (condition ou structure). A l'avenir, ce sera une manière efficace de distinguer les structures et leur hiérarchie dans le code, dans un contexte de gestion de la mémoire et des objets.

```
Vous avez clické sur 'Alors',
 qui est un mot-clé de type 'structure',
 avec comme id 2.
```

FIGURE 5 – Cliquer sur un mot-clé de structure de contrôle donne son type, son nom et son ID.

Ceci est réalisé au moyen d'un **JTextPane**, qui permet d'exploiter des **Element** (*classe* ***javax.swing.text.Element***), et ainsi de leur ajouter des **Listener**, ainsi qu'un set d'attributs de type **SimpleAttributeSet**.

7.3 Les structures de contrôle

Pour gérer les différentes structures de contrôle dont on peut avoir besoin dans la construction des raffinages, nous avons décidé de créer une classe abstraite "StructureDeControle". De cette façon, nous pourrons y ajouter autant de structures de contrôle que l'on souhaite. Pour cette première itération, nous avions décidé de ne mettre que la structure de contrôle "si".

Tout d'abord, parlons de la classe abstraite "StructureDeControle". Cette classe contient plusieurs attributs qui seront communs à toutes les structures de contrôle que l'on voudra implémenter par la suite. Nous avons parmi ces attributs la condition ou le corps par exemple. La condition est une String contenant la condition principale qui compose une structure de contrôle. Le corps quant à lui, est une List d'élements de type Structure, c'est-à-dire de structures de contrôle, d'action élémentaire ou de raffinages.
Nous avons fait le choix ici de ne rien interdire à l'utilisateur. Il n'a pas de condition sur le nombre de structures de contrôle que l'utilisateur peut ajouter, ni sur le nombre d'actions simples ou complexes qu'il peut ajouter dans un même raffinage. Cela sera l'objet d'une amélioration future.
Cette classe comporte également d'autres attributs, qui servent pour l'interface graphique et l'affichage.

Ensuite, nous allons détailler la classe "StructureSi". Cette classe étend de la classe abstraite "StructureDeControle". Elle contient un champ "aireTexte" de type "VueEditionRaffinages" ainsi qu'un constructeur qui prend en paramètre une chaîne de caractères représentant la condition, une chaîne de caractères représentant le nom, et une instance de la classe "VueEditionRaffinages". La méthode "afficher()" de la classe "StructureSi" est implémentée pour afficher une chaîne de caractères représentant une structure de contrôle "Si" dans l'objet "VueEditionRaffinages" associé à cette instance. Cette chaîne est construite en utilisant la condition et le corps de la structure "Si" et est ensuite ajoutée à l'objet "aireTexte" à l'aide de la méthode "append()". Enfin, un message "Condition SI ajoutée !" est affiché dans la console.

Les autres classes "Structure" qui représentent les autres structures de contrôle simples (pour, tant que, répéter) ont un fonctionnement identique à la classe StructureSi, à la différence que l'affichage a été modifié en conséquence.

La classe "StructurePour" hérite de la classe abstraite "StructureDeControle" et implémente la méthode toString(). Cette méthode permet de retourner une représentation sous forme de chaîne de caractères de la structure "Pour" et de son contenu. La structure "Pour" prend en compte plusieurs paramètres :

+nom : le nom de la structure, utilisé pour l'identification ou le suivi dans le programme.

+var : la variable de contrôle utilisée dans la boucle "Pour" pour itérer sur la séquence.

+debut : la valeur initiale de la variable de contrôle.

+fin : la condition de fin de la boucle "Pour", indiquant jusqu'où itérer.

La méthode toString() parcourt les éléments du corps de la boucle "Pour" et les ajoute à la représentation sous forme de chaîne de caractères. Chaque élément est précédé d'une indentation pour une meilleure lisibilité. Enfin, la chaîne de caractères résultante est retournée avec l'ajout des mots clés "Pour", "De", "À" et "FinPour" pour indiquer le début et la fin de la boucle "Pour". Cela permet d'utiliser la structure "Pour" dans le programme en créant une instance de la classe "StructurePour" avec les paramètres appropriés et en ajoutant les éléments à exécuter dans la boucle à l'aide de la méthode ajouterElement() de la classe abstraite "StructureDeControle".

La classe "StructureTantque" hérite de la classe abstraite "StructureDeControle" et implémente la méthode toString(). Cette méthode retourne une représentation sous forme de chaîne de caractères de la structure "TantQue" et de son contenu. La structure "TantQue" prend en compte deux paramètres :

+condition : la condition qui doit être vérifiée pour continuer à exécuter le bloc d'instructions. Tant que cette condition est vraie, le bloc d'instructions est répété.

+nom : le nom de la structure, utilisé pour l'identification ou le suivi dans le programme.

La méthode toString() crée une chaîne de caractères en ajoutant les mots clés "TantQue" et "Faire" suivis de la condition spécifiée. Ensuite, elle parcourt les éléments du corps de la structure "TantQue" et les ajoute à la représentation sous forme de chaîne de caractères. Chaque élément est précédé d'une indentation pour une meilleure lisibilité. Enfin, la chaîne de caractères résultante est retournée avec l'ajout du mot clé "FinTQ" pour indiquer la fin de la structure "TantQue".

La classe "StructureRepeat" hérite de la classe abstraite "StructureDeControle" et implémente la méthode toString(). Cette méthode retourne une représentation sous forme de chaîne de caractères de la structure "Repeat" et de son contenu. La structure "Repeat" prend en compte deux paramètres :

+condition : la condition qui doit être vérifiée pour terminer la répétition du bloc d'instructions. Une fois cette condition satisfaite, la répétition s'arrête.

+nom : le nom de la structure, utilisé pour l'identification ou le suivi dans le programme.

La méthode toString() crée une chaîne de caractères en ajoutant le mot clé "Faire" pour indiquer le début du bloc d'instructions. Ensuite, elle parcourt les éléments du corps de la structure "Repeat" et les ajoute à la représentation sous forme de chaîne de caractères. Chaque élément est précédé d'une indentation pour une meilleure lisibilité. Enfin, la chaîne de caractères résultante est retournée avec l'ajout du mot clé "Jusqu'À" suivi de la condition spécifiée pour indiquer la fin de la structure "Repeat".

Nous avons également commencé à travailler sur l'implémentation de structures de contrôle plus complexes. En effet, les structures dites "simples" sont celle qui n'ont qu'un seul corps à gérer. Les structures plus complexes comme le SiSinon, ou le SwitchCase, ont 2 corps différents, voir un nombre indéfini de corps différents.

Nous avons décidé de réfléchir en se disant que l'utilisateur devrait rentrer un nombre n de cas qu'il voudrait pour le SwitchCase au moment de la création du raffinage. Ensuite, nous avons également décidé de démarrer par réaliser uniquement le SiSinon pour regarder les différents problèmes que l'on pourrait avoir et comment faire fonctionner la structure.

La structure de contrôle "SiSinon" permet de conditionner l'exécution de certaines instructions en fonction d'une condition, en les regroupant dans les parties "Alors" et "Sinon" de la structure "Si-Sinon". La méthode toString() génère une représentation lisible de cette structure avec les instructions correspondantes.

7.4 Les actions complexes et élémentaires

L'interface Action étend l'interface Structure, elle ajoute des méthodes spécifiques aux actions, notamment addFormat() et removeFormat() pour ajouter et supprimer des formats de texte, getCouleur() et setCouleur() pour obtenir et définir la couleur du texte, et getTexteFormate() pour obtenir le texte formaté de l'action. Enfin, la méthode print() permet d'afficher le contenu de l'action sur la console. La méthode addFormat() prend un paramètre TextFormat, qui peut être utilisé pour formater le texte de l'action. La méthode removeFormat() prend également un paramètre TextFormat pour supprimer un format existant.
La méthode getCouleur() renvoie la couleur actuelle de l'action, tandis que setCouleur() permet de définir la couleur du texte. Les couleurs peuvent être représentées par des objets TextColor qui définissent des couleurs prédéfinies.

La méthode getTexteFormate() renvoie le contenu de l'action avec tous les formats de texte appliqués. Cette méthode peut être utilisée pour afficher le contenu formaté dans l'interface utilisateur.
Nous avons ajouté une méthode toString() pour afficher une représentation textuelle de l'action élémentaire, qui inclut le titre de l'action et pour une action complexe, cette méthode parcourt les éléments du corps de l'action et les ajoute à la représentation sous forme de chaîne de caractères. Chaque élément est précédé d'une indentation pour une meilleure lisibilité.

Pour différencier entre une action qui nécessite d'être raffinée et une action simple, nous avons implémenté deux classes ActionComplexe et ActionElementaire qui réalisent l'interface Action.

Pour implémenter les constructeurs des deux classes, nous avons opté à choisir les attributs suivantes :
- int niveau ; pour modéliser le niveau de raffinage correspondant à l'action complexe
- LinkedList<Element> Elements ; pour modéliser les blocs qui structurent l'action complexe (Raffinage)
- String contenu ; pour modéliser le contenu du bloc de texte
- LinkedList<TextFormat> formats ; pour modéliser les formats de texte à appliquer (par exemple gras, italique, souligné)
- TextColor couleur ; pour modéliser les couleurs de texte à appliquer (par exemple BLACK, RED, GREEN, YELLOW, BLUE, PURPLE, WHITE)

Enfin, nous avons effectué des tests de Junit pour s'assurer du bon fonctionnement des classes et la robustesse des méthodes implémentées.

7.5 Les boutons d'édition de raffinage

Pour pouvoir agrémenter un raffinage, l'utilisateur doit se servir des différents boutons mis a dispositions. Ce dernier a donc la possibilité d'ajouter des structures de contrôles ainsi que des actions, complexe ou élémentaires.

7.5.1 Boutons d'actions

L'utilisateur peut ajouter a son raffinage une action en cliquant sur les boutons dédiés. Ces derniers ont donc chacun un observateur qui va tout simplement, instancier une nouvelle Action en récupérant son contenu via un input utilisateur, et l'ajouter au éléments (liste chaînée) du raffinage courant (celui choisi sur l'arbre). Néanmoins, dans le cas d'une action complexe, le traitement est différent. En effet, si une action complexe est présente dans un raffinage, c'est qu'il faut la raffiner ie ajouter un nouveau niveau de raffinage. C'est pour cela que lorsque l'on clique sur le bouton action complexe, non seulement celle-ci est ajoutée au raffinage courant, mais un nouveau niveau de raffinage va être automatiquement ajouter à l'arbre. Ceci de manière totalement analogue à l'ajout manuel d'un nouveau raffinage.

7.5.2 Boutons de structures de contrôles

Comme pour les boutons d'actions, ceux dédiés aux structures de contrôles vont instancier une nouvelle structure de contrôle avec les informations rentrées par l'utilisateur, puis le rajouter aux éléments du raffinage courant. Cependant, il n'est pas envisageable de créer un observateur diffèrent pour chaque type de structure de contrôle. Cela remettrait en cause totalement l'utilisation du patron de conception 'Strategie' rigoureusement choisi au préalable. La solution est alors de passer en paramètre d'instance de l'observateur la classe de la structure de contrôle :

```
Class<? extends StructureDeControle> class1
```

Listing 1 – Parametre d'instance de l'observateur

Grace à cela, on peut travailler en initialisant une variable de poigné apparente StructureDeControle mais qui va ensuite avoir comme poignée réelle celle de la classe récupérée en paramètre. Pour cela, on utilise la méthode *getConstructor()* qui prend en argument les types des paramètres du constructeur recherché, puis *newInstance()* pour créer une instance de la classe :

```
sdc = classe.getConstructor(new Class<?>[] { String.class, String.class
    }).newInstance(condition, "");
```

Listing 2 – Instanciation générique d'une structure de controle

7.6 L'arbre des raffinages

Une fonctionnalité phare de l'application est la possibilité d'accéder aux raffinages sous la forme d'un arbre. Celui-ci se trouve sur la partie droite de l'interface graphique et est construit au fur à mesure par l'utilisateur. Pour cela, lors de l'ouverture de l'application, un pop-up apparaît demandant le raffinage 0, celui qui servira comme racine de l'arbre (voir figure 6).

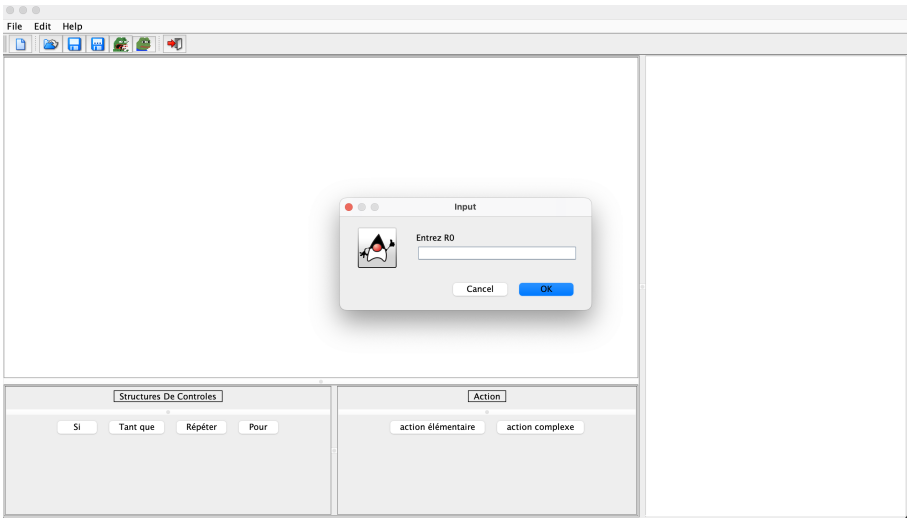

FIGURE 6 – Pop-up a l'ouverture de l'application

Pour pouvoir construire cet arbre, nous avons choisi de travailler avec la Classe **JTree** de la librairie *java.swing.tree*, ainsi que créer une classe à par entière, *VueListeRaffinage*. L'arbre va alors être composé d'actions complexes, dont leur titre va être affiché à chaque noeud (voir figure 7). En effet, il y a une arborescence directe lors de la création de raffinage. Un R_1 peut avoir plusieurs R_2 qui eux même peuvent avoir plusieurs R_3, *et cetera*.

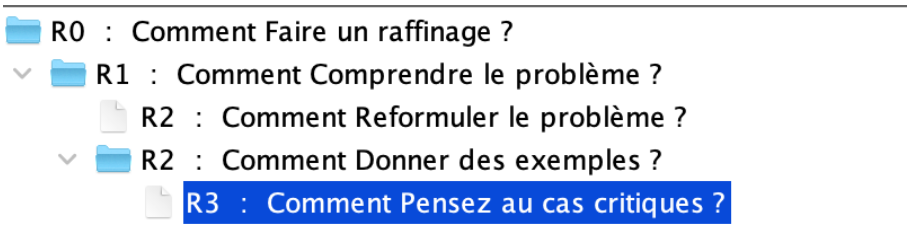

FIGURE 7 – Exemple d'arbre de raffinage

De plus, cet arbre doit être interactif avec l'utilisateur. Pour cela nous avons choisi d'introduire un **PopupMenu**, de la librairie *java.swing.JPopupMenu*. On ajoute alors à celui-ci deux **MenuItem**, qui seront les actions 'Ajouter' et 'Supprimer'(Voir figure 10). Il est alors nécessaire de créer un observateur, qui comme pour le cas de boutons, va une action lorsque l'utilisateur clique sur 'Ajouter' ou 'Supprimer'. Pour cela nous avons créer une classe interne, *PopupActionListener* qui réalise l'interface *ActionListener*. Celle-ci va, en fonction du clic utilisateur, traiter la commande associé (ajouter ou supprimer un raffinage).

Pour pouvoir créer un nouveau niveau de raffinage, il suffit alors de faire un "clic droit" sur un raffinage existant. Un pop-up apparaîtra alors demandant, comme pour le R_0 un nom pour le raffinage. Il suffit alors de rentrer une action et un nouveau niveau de raffinage sera créer, et aura pour parent le raffinage sur lequel le clic à été effectué. Il en va ainsi de même pour supprimer un raffinage. Cependant il faut noter que supprimer un raffinage va alors supprimer tout les raffinages plus profonds liés a celui-ci. De plus, il n'est, dans cette version, pas possible de supprimer le R_0. Cela correspondrait en effet à tout recommencer de zéro et sera implémenté dans les versions futures.

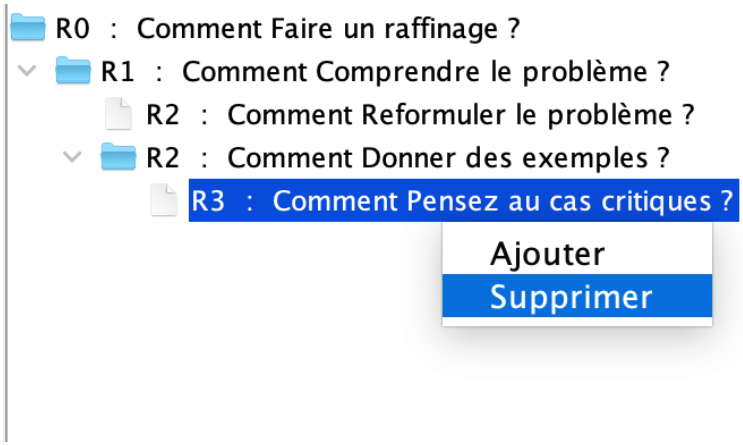

FIGURE 8 – Exemple d'arbre de raffinage avec un PopupMenu

8 Problèmes rencontrés et solutions apportées

8.1 1ère itération

Lors de cette première itération, nous avons rencontrés les problèmes suivants :

→ Problème de conception : Comment faire pour gérer l'identification (le "parsing") du texte du raffinage en cours dans la section d'édition ? Il a fallu changer la JTextArea en JTextPane, car la première est très basique et inexploitable sur le long terme.

→ Difficultés lors de la conception de l'interface graphique : En effet, pour pouvoir implémenter la fonctionnalité de "clic droit" sur l'arbre, il est nécessaire de créer une classe *PopupActionListener*. Néanmoins, celle-ci n'a pas accès au *JTree* et ne peux donc pas le modifier.
Solution : Définir la classe de manière *interne*, ie dans la classe *VueEditionRaffinage*.

→ Par rapport aux structures de contrôle : Comment les ajouter ? nous avons décidé de procéder en faisant ajouter à l'utilisateur les structures depuis des boutons, sans avoir à les écrire lui-même à la main. Il n'aurait ensuite qu'à rajouter lui-même les actions élémentaires et les raffinages compris dans ou hors de ces structures dites *Actions complexes*.
→ Par rapport aux actions complexes et élémentaires : Comment modéliser une action et quels sont ses attributs principaux ? Finalement, après discussion, nous avons opté à choisir les attributs suivants :
- int niveau ; pour modéliser le niveau de raffinage correspondant à l'action complexe
- List<Element> sousFils ; pour modéliser les blocs qui structurent l'action complexe (Raffinage)
- String contenu ; pour modéliser le contenu du bloc de texte
- List<TextFormat> formats ; pour modéliser les formats de texte à appliquer (par exemple gras, italique, souligné)
- TextColor couleur ; pour modéliser les couleurs de texte à appliquer (par exemple BLACK, RED, GREEN, YELLOW, BLUE, PURPLE, WHITE)
Ce choix est effectué dans le but d'englober tous les aspects possibles pour la lecture et l'affichage des raffinages dans les prochaines itérations à venir.
Nous avons également confronté le problème de la modélisation des couleurs et formats de texte saisi, et nous avons décidé d'utiliser des énumérations avec des codes hexadécimales pour aboutir aux résultats attendus et pour un meilleur affichage sur la console.

8.2 2ème itération

→ Comment afficher le contenu d'un raffinage ? En effet, un raffinage est composé d'une liste de différents éléments, qui peuvent eux aussi être des raffinages. Pour cela un première approche serait d'utiliser la méthode *toString()* implicitement récursivement en initialisant un string, et en y ajoutant chaque éléments du raffinage en parcourant la liste chaînée. Néanmoins cela engendrerais quelque problèmes. En particulier, prenons par exemple un raffinage R2. Ce que l'on veut voir à l'écran sont les différentes structures de contrôles ainsi qu'actions complexe
élémentaires de **ce niveau de raffinage (2 ici)**. Dans le cas de l'approche précédentes, on aurait affiché le contenu entier de tout les raffinages jusqu'au bout. **Solution :** Créer une nouvelle méthode connaissant le niveau de raffinage actuel. Si l'on passe en paramètre de méthode le niveau actuel de raffinage, il est alors possible de faire une simple disjonction de cas.

8.3 3ème itération

Nous avons rencontré quelques problèmes durant cette itération. Nous avons du modifier pas mal de code pour pouvoir implémenter le fonctionnement de l'ajout via le curseur. Cela nous as demandé bien plus de temps que prévu, mais cela était nécessaire si nous souhaitions avoir une application fonctionnelle pour le rendu final. En effet, cette fonctionnalité faisait partie des fonctionnalités primordiales pour le bon fonctionnement de l'application.

9 Organisation de l'équipe

Pour la première itération, nous avons décidé de nous répartir les rôles pour être plus efficace. Nous avons commencé par organiser une réunion, où nous sommes tous venu pour essayer de répartir équitablement le travail, en se basant sur les objectifs que l'on s'était fixé. Nous en sommes donc arrivés à la répartition suivante :

— Robin Augereau : mettre au propre le diagramme UML

— Safae Belahrach : la partie action (action élémentaire et raffinages) + arbres

— Barbara Do Couto Vidal : la partie action (action élémentaire et raffinages) + arbres

— Amandine Gravier : la partie structure de contrôle (avec le « si »)

— Fatima zahra EL ASRI : la partie structure de contrôle (avec le « si »)

— Axel Deora : interface graphique et gestion de l'édition des raffinages

— Antonin Tarrade : interface graphique et gestion de l'arbre

— Julien HUANG : interface graphique et boutons d'action

Nous avions, grâce à cette répartition, des groupes de travail plus réduit, ce qui est plus simple pour travailler à distance. Nous avons décidé de se laisser la première semaine de travail en petit groupe pour implémenter les différentes classes. Suite à cela, nous avons réaliser un seconde réunion au début de la seconde semaine. Le but de cette réunion était de faire part des remarques et problèmes que nous avions pu rencontrer. Suite à cette seconde réunion, nous avons tous dû communiquer via Discord, car toutes les classes sont grandement inter-connectées. Nous avons ensuite tous participé à la rédaction du rapport commun, pour pouvoir expliquer plus spécifiquement ce que chacun a fait.

Pour la seconde itération, nous avons gardé la même méthode d'organisation qu'à la première itération. Nous avons quand même effectuer quelques modifications. En effet, nous avons eu une réunion de rétrospective sur la première itération, et une idée est revenue de la part de tout le groupe : il nous manquait une plateforme pour avoir une liste claire des différentes tâches que nous devons faire. Nous avons donc décider d'utiliser Trello, une plateforme permettant de gérer les différentes tâches de notre projet. Nous y avons séparé les tâches selon sur quoi elles portent, pour chaque tâche, la personnes travaillant dessus peut écrire un commentaire pour communiquer son avancée aux autres. Nous avons également ajouté les réunions avec leur date, un emplacement pour écrire les points à aborder, et un compte rendu pour les réunions passées. Voici un apperçu de notre Trello :

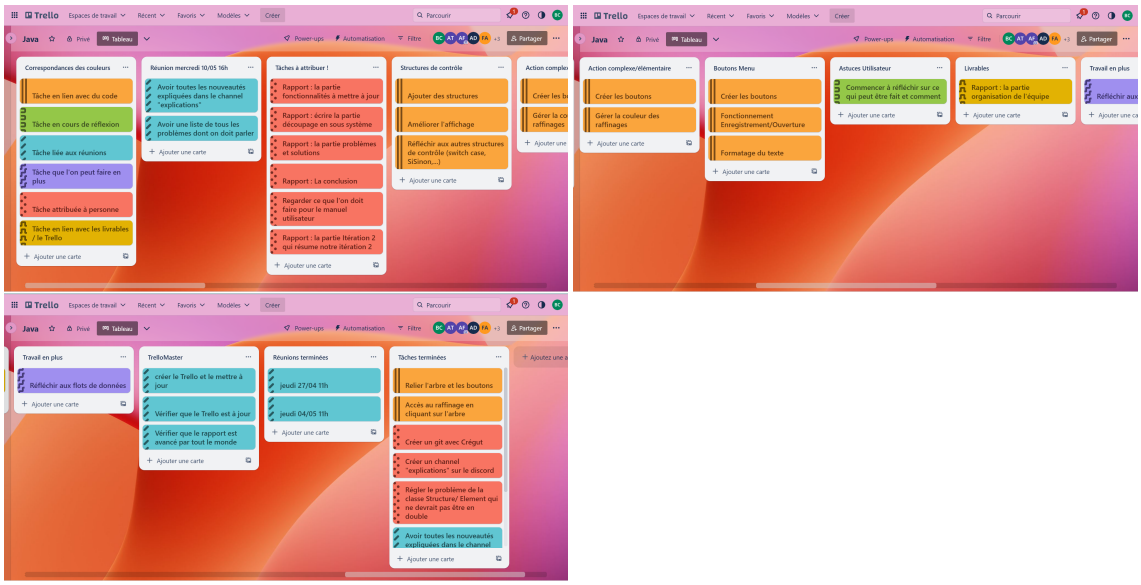

FIGURE 9 – Organisation du Trello

Pour le discord, nous l'avons également amélioré afin d'avoir des channels spécifiques pour chaque partie du projet, ce qui permet d'avoir facilement accès aux informations nous concernant. Nous ajouté un channel "explications" où chaque personne peut mettre une explication sur le fonctionnement de la partie qu'il a implémentée une fois celle-ci opérationnelle.

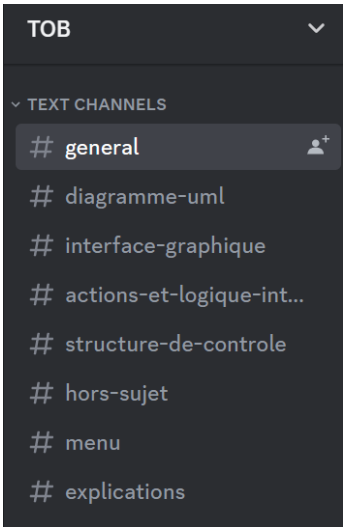

FIGURE 10 – Organisation du groupe Discord

Nous avons ensuite réalisé des réunions toutes les semaines pour échanger sur le travail effectué, en plus de la première réunion permettant de répartir le travail. Nous avions aussi convenu dans un souci d'efficacité, de continuer à échanger via Discord pour certaines demandes urgentes. La répartition que nous avons choisi pour cette itération est la suivante :

— Robin Augereau : Gestion de la fonction d'Export du document en format Json

— Safae Belahrach : Lien entre l'arbre et le bouton d'action complexe + Implémantation des boutons actions et leurs Listeners (ActionComplexeListener et ActionElementaireListener)

— Barbara Do Couto Vidal : Gestion du Trello
— Amandine Gravier : implémentation des nouvelles structures de contrôle + gérer le Trello
— Fatima zahra EL ASRI : implémentation des nouvelles structures de contrôle
— Axel Deora : Lien entre l'arbre et l'édition, coloration et gestion du texte & mots-clés + gestion de la sérialisation / désérialisation
— Antonin Tarrade : Lien entre l'arbre et l'édition + fonctionnement des principaux boutons
— Julien HUANG : Création des boutons nouveau fichier, enregistrer sous, ouvrir et police du texte

Nous avons ici également tous participé à la rédaction du rapport collectif, en plus de nos rapports individuels. Nous avons réussi à avancer correctement le projet selon nos objectifs. Le Trello nous a permis de tous visualiser les différentes tâches qu'il nous restait à faire.

Pour cette troisième et dernière itération, nous avons donc décidé de poursuivre les tâches déjà commencées aux deux premières itérations. Nous avons donc gardé une répartition similaire à celle des deux premières itérations : une réunion pour se répartir le travail et parler de l'organisation de l'équipe. La répartition du travail est la même qu'aux autres itérations car nous avons décidé de finir ce que nous n'avions pas complètement réussi à finir. Chacun a donc travaillé sur les mêmes choses qu'avant. De plus, nous avons également travaillé sur les livrables pour l'oral, afin de s'assurer que chacun soit en mesure de présenter chaque partie de la présentation.

10 Conclusion

10.1 Première itération

Pour conclure, cette première itération consistait à présenter les raffinages sous forme d'arbre, afin de faciliter leur lecture et leur affichage. De ce fait, nous avons pu créer des raffinages structurés via une interface graphique conviviale. L'interface graphique comporte des boutons modélisant le type de structure de contrôle à utiliser ainsi le choix entre la création d'une action élémentaire ou complexe, l'utilisateur a ainsi la possibilité de choisir la structure de contrôle pour la modéliser sur l'écran (coté droit de l'interface). L'interface graphique comprend également des boutons pour saisir les actions complexes et élémentaires que l'on essayera de développer et d'améliorer dans les prochaines itérations.

10.2 Deuxième itération

Dans cette deuxième itération, nous avons lié les différentes représentations du raffinage, et nous avons continué d'implémenter les options de raffinage en rajoutant le bouton action complexe, ainsi que les boucle Pour, TantQue et Répéter. Nous avons également amélioré l'interface graphique en implémentant de nouvelles fonctionnalité qui permettent une meilleure personnalisation de l'espace de travail.

10.3 Troisième itération

Pour cette troisième itération, nous avons consolidé nos acquis en matière de méthodes agiles. Nous avons maintenu notre communication régulière et transparente, en nous concentrant sur l'atteinte de nos objectifs et la réalisation d'une version fonctionnelle pour l'utilisateur. Nous avons continué à utiliser Trello pour suivre nos tâches et à organiser des réunions pour échanger des idées et obtenir des retours sur l'organisation de notre projet.
En résumé, notre projet a été un exemple concret de l'application des méthodes agiles dans la communication et la gestion d'équipe. Nous avons su tirer parti des principes agiles pour améliorer notre efficacité, notre collaboration et notre capacité à livrer une solution de qualité à nos utilisateurs.