Compte-Rendu

SAE 3.01 Développement d'Applications

ANTZORN Hugo - BOUAOUKEL Walid - BOUDOUAH Ilias - CARNET Alexander

Sommaire

Fonctionnalités réalisées

1. Création de colonnes (type Kanban) avec limite du nombre de colonnes et gestion d'erreur en cas de dépassement.

2. Renommage d'une colonne (passage en édition + validation au clavier).

3. Ajout d'une tâche dans une colonne via une fenêtre dédiée (titre, description, durée, date de début, dépendances, sous-tâches).

4. Déplacement d'une tâche d'une colonne à une autre par glisser-déposer (drag & drop).

5. Modification d'une tâche via une fenêtre dédiée (édition des attributs et mise à jour des dépendances/sous-tâches).

6. Consultation des détails d'une tâche via une fenêtre (titre, description, durée, date, liste des dépendances, liste des sous-tâches).

Relations entre tâches

7. Ajout de dépendances entre tâches.

8. Ajout de sous-tâches à une tâche.

9. Détection et prévention des cycles de dépendances : refus des dépendances invalides lors de l'ajout/modification (parcours des dépendances).

10. Nettoyage des références lors de l'archivage : suppression des liens vers la tâche archivée dans les autres tâches et retrait de la sélection Gantt si nécessaire.

Archivage

11. Archivage d'une tâche (retrait de sa colonne puis ajout à l'archive).

12. Consultation de l'archive via une fenêtre dédiée.

13. Désarchivage d'une tâche (réintégration dans la première colonne) et suppression définitive d'une tâche archivée.

Vues et interface

14. Vue Bureau (Kanban) : colonnes et tâches sous forme de cartes, ajout de tâches, renommage, drag & drop, ouverture des fenêtres de détails/modification.

15. Vue Liste : tableau récapitulatif des tâches (titre, colonne, date de début, durée, dépendances) avec possibilité de déplacer une tâche en modifiant sa colonne via la table.

16. Menu de navigation (VueAccueil) permettant le changement de vue : Bureau / Liste / Archive / Gantt.

Diagramme de Gantt

17. Sélection des tâches à afficher dans le Gantt via une fenêtre dédiée (cases à cocher).

18. Génération du diagramme de Gantt à partir des tâches sélectionnées.

19. Prise en compte des dépendances dans le calcul des dates : une tâche ne démarre pas avant la fin de ses dépendances.

20. Affichage d'une grille journalière avec repères de dates en en-tête.

21. Gestion du dépassement écran : rendu dans un conteneur scrollable permettant de naviguer dans le Gantt.

Persistance et exécution

22. Sauvegarde/chargement du modèle complet (Repository : saveAll/loadAll) : colonnes, tâches, archive, sélection Gantt, etc.

23. Lancement de l'application via une classe JavaFX de test (TestVue) qui charge les données (ou crée un modèle vide) puis affiche l'interface principale.

Structure logicielle

24. Mise en place d'une architecture MVC avec Observateur : le modèle notifie les vues qui se rafraîchissent automatiquement après une action.

25. Utilisation d'une stratégie de rendu (StrategieModeAffichage) pour changer dynamiquement de vue (Bureau/Liste/Gantt) sans modifier la logique métier.

Diagramme de classe final

Répartition du travail entre étudiants

Hugo:

- Déplacement d'une tâche entre colonnes (drag & drop).
- Affichage des détails d'une tâche (popup de consultation).
- Gestion des dépendances entre tâches (ajout / modification).
- Prévention des cycles dans les dépendances (détection + blocage).
- Conception des classes de base : Tache et TacheComposite.
- Mise en place du système de sous-tâches.
- Gestion du calcul du diagramme de Gantt (dates effectives selon dépendances).
- Affichage du diagramme de Gantt (barres, échelle par jour, dates, scroll si dépassement).
- Rédaction des tests liés aux tâches.

Contrôleurs développés

- ControleurAjouterDependances
- ControleurAjouterSousTache
- ControleurModifierTache
- ControleurAjouterTache
- ControleurDragAndDrop

Vues / Popups développées

- PopupDetailsTache
- VueGantt

Walid:

- Classes principales pour les Vues:
    - Observateur;
    - StrategieModeAffichage;
    - VueTaches;
- VueListe;
- Possibilité de changer la colonne d'une tâche depuis la VueListe;
- (Popup) Fenetre dediee a' la création d'une nouvelle tâche (titre, description, durée, date de début, dépendances, sous-tâches);
- Possibilité de créer une tâche depuis la VueListe;

- (Popup) Fenetre dediee a' la modification d'une nouvelle tâche (titre, description, durée, date de début, dépendances, sous-tâches);
- (Popup) Fenetre dediee a la liste des tâches archivées;
- Possibilité d'archiver ou de modifier une tâche grâce à un contextmenu depuis la VueListe;
- Possibilité de désarchiver ou de supprimer une tâche de l'archive;
- Dans le cas où il n'existerait pas de colonnes, annule la possibilité' désarchiver les tâches archivées;

Ilias:

- Classe Colonne
- Classe Modèle
- Vue Accueil
- Supprimer une Colonne (donc le bouton et contrôleur associé, ainsi que sa fonction dans le Modèle)
- Archiver une Tâche (donc le bouton et contrôleur associé, ainsi que sa fonction dans le Modèle)
- Tests unitaires pour le Modele et la Colonne

Alexander:

- Vue Bureau (Kanban) : affichage des tâches et des colonnes
- Sélection de tâches pour le diagramme de gantt (Popup)
- Sélection de toutes les tâches d'un coup pour le diagramme de Gantt
- Sauvegarde et persistance des données de l'application : les colonnes, les tâches dans les colonnes, les tâches archivées.
- Ajout d'une nouvelle tâche par la vue bureau
- Ajout d'une nouvelle colonne dans la vue bureau
- Renommer une colonne sur la vue bureau
- Modification d'une tâche par la vue bureau
- Ecriture de test unitaire pour le repository

Présentation d’un élément original dont vous êtes fiers

Hugo:

J’ai choisi de développer le diagramme de Gantt car c’est une fonctionnalité qui m’attire beaucoup. C’était motivant car cela demandait à la fois de réfléchir à un calcul cohérent des dates et de construire un affichage graphique clair. Le point le plus intéressant a été la prise en compte des dépendances entre tâches. Une tâche ne peut pas commencer tant que toutes ses dépendances ne sont pas terminées. Pour résoudre ça, j’ai séparé le problème en deux parties :

- Un calculateur (GanttCalculator) qui génère pour chaque tâche une date de début “effective” et une date de fin.
- Une vue (VueGantt) qui transforme ces dates en coordonnées à l’écran (décalage en jours depuis une date minimale) pour dessiner les barres.

Le calcul repose sur un parcours récursif des dépendances, pour déterminer le début d’une tâche, on calcule la fin de chaque dépendance, puis on garde la date la plus tardive comme début effectif. Une map permet de mémoriser les résultats déjà calculés ce qui évite de recalculer plusieurs fois la même tâche lorsqu’elle sert de dépendance à plusieurs autres tâches.
Problèmes rencontrés et solutions apportées :

- Problème 1 : dépendances en chaîne (dépendance de dépendance). Sans traitement récursif, certaines tâches démarraient trop tôt. Solution : calcul récursif du début effectif en parcourant toutes les dépendances jusqu’aux tâches “racines”, avec mise en cache des dates calculées.

- Problème 2 : cycles de dépendances. Un cycle (A dépend de B, B dépend de A ou cycles plus longs) rend le planning incohérent et peut provoquer une récursion infinie. Solution : vérification au niveau du modèle lors de l’ajout/remplacement de dépendances (refus si cela crée une boucle) avec retour utilisateur via un message d’erreur.

- Problème 3 : lisibilité et dépassement de l’écran. Un Gantt devient vite large (axe du temps) et haut (nombre de tâches) ce qui bloque l’affichage. Solution, ajout d’une grille journalière + dates affichées en haut et intégration du diagramme dans un conteneur scrollable pour naviguer horizontalement et verticalement.

Au final, cette fonctionnalité est celle dont je suis le plus fier car elle combine plusieurs aspects du projet, respect de l’architecture MVC, gestion de contrainte et rendu graphique clair

Walid:

Les éléments dont je suis fier dans ce travail de groupe sont les fenêtres modales (popup) pour la création et la modification d'une tâche, ainsi que l'intégration des menus contextuels, d'autant plus que les popups ont également servi pour les autres Vues.

L'objectif principal était de ne pas trop encombrer la fenêtre principale. Pour cela, j'ai pensé à utiliser:

- Le menu contextuel: qui permet d'accéder aux actions sans remplir l'écran de boutons;
- Les popups: ils isolent la création ou la modification d'une activite, permettant à l'utilisateur de se concentrer sur la saisie des données.

Problèmes rencontrés :

- Après l'insertion des données dans le popup de création et la confirmation, je rencontrais des problèmes de conflits entre les types de variables utilisés par les autres classes et ceux que j'utilisais pour créer une nouvelle tâche.

Solution : J'ai essayé de modifier certains types de variables sans créer d'erreurs pour les autres, ou de convertir les informations passées au popup pour qu'elles correspondent au type utilisé par les classes.

- Lorsque je modifiais les informations d'une tâche, les informations sur la fenêtre principale ne se mettaient pas à jour automatiquement. Pour voir les changements, je devais cliquer deux fois sur chaque colonne de la tâche ou changer de Vue.

Solution : Le problème venait du fait que le tableau ne se rafraîchissait pas, même si je mettais à jour l'observateur. J'ai donc dû ajouter une ligne de code pour forcer la mise à jour du tableau.

Ilias:

Personnellement je choisis de présenter toute la partie Accueil de l'interface de l'application, étant ce que je considère la partie la plus intéressante que j'ai pu toucher (vu que c'est ce qui permet de se déplacer facilement entre les vues, donc une fonctionnalité qu'on pourrait juger comme capitale). Il fallait modifier "l'architecture", et implémenter une classe intermédiaire entre StratégieModeAffichage et l'Interface VueTache. Cette conception permets de laisser la partie Accueil de l'écran "fixe", tandis que la partie basse affichant les vues est gérée par le Modèle et la VueAccueil, et change basé sur le bouton appuyé par l'utilisateur qui est présent du coup dans l'Accueil.
Mon problème majeur que j'ai pu rencontrer était justement de trouver la conception derrière pour garder une partie de l'écran "fixe", que le programme ne devait pas s'occuper de tout

reafficher à chaque fois. Les 2-3h d'analyse en début d'itération m'ont donc été fortement utiles. J'ai également eu beaucoup de problème pour ce qui est au niveau des handler, étant donné que cela faisait un certain temps que je n'ai pas pu en pratiquer. Mais au final, j'ai pu réussir à implémenter un Accueil qui marche.

Alexander :

J'ai choisi de présenter la sélection de tâches pour Gantt car c'est un élément central dans l'application qui sert à l'utilisateur pour personnaliser le diagramme de Gantt. L'interface doit donc être facile et rapide d'utilisation. Pour cela j'ai décidé d'utiliser des checkbox et d'ajouter un bouton pour sélectionner toutes les tâches d'un seul coup. Ce qui peut faire gagner beaucoup de temps si on à beaucoup de tâches à sélectionner.

J'ai rencontré 2 problèmes lors du développement de la sélection des tâches :

1. Les Checkbox de javaFX ont un fonctionnement un peu spécial, lorsque l'utilisateur coche ou décoche la case, celle-ci se met à jour graphiquement toute seule et avant le handler que j'ai associé à la checkbox, je récupère donc la donnée inverse dans mon code. La résolution de ce problème à donc été assez simple, j'ai juste inverser le résultat que je récupérait.

2. Le second problème était une erreur de ma part concernant le fonctionnement de javaFX, Les checks box ne se mettait pas à jour lors du click sur le bouton pour tout sélectionner .Dans la méthode actualiser de mon popup, qui est appelé lors du click sur le bouton pour tout sélectionner, je recréer une VBox contenant les checkBox et les nom des tâche et je remplaçais celle qui existait déjà par la nouvelle . Mais en procédant de cette manière l'UI ne se mettait pas à jour, il fallait clear la VBox déjà existante et la remplir à nouveau.

Les éléments qui ont été modifiés par rapport à l'étude préalable

Comme attendu, le plan d'Itérations a été beaucoup modifié comparé à ce qui a été fait.

Lors de la grande partie d'analyse servant à étudier le sujet, nous avions décidé de partir sur une architecture ActiveRecord en ce qui concerne la sauvegarde des Tâches et Colonnes (des données plus généralement) entre chaque lancement de l'application. Mais au final,

nous avons à la place implémenté un Singleton Repository, qui avec ses méthodes enregistre et charge la sauvegarde par le biais d'un fichier texte.

Un second point qui a différé depuis est l'organisation direct de notre travail. Lors de l'étude préalable, nous avons supposé que nous allions implémenter la Vue Liste qu'à l'itération 5. Cependant, nous avions conclu qu'il fallait mieux la faire dès la première itération, étant donné qu'il nous fallait de quoi faire une démonstration dés la fin de la première itération, et qu'elle était la partie de l'application la plus aisée à faire.

Un troisième point était au vis à vis des tâches en elle même : à la toute base, on souhaitait mettre un système d'état pour chaque tâche, où l'utilisateur pouvait choisir un état entre "En cours", "Non commencé", "Fini" et "En pause", et ce pour chaque tâche. Mais nous avons jugé mieux que l'utilisateur se serve des différentes colonnes pour modéliser un état, et qu'il était donc inutile d'implémenter ce système d'état.

Un autre point de différence que l'on pourrait tout de même noter : lorsque l'utilisateur tente de créer une nouvelle colonne, l'application lui affichait un popup lui permettant de saisir les informations. Cependant, nous avons opté pour une solution plus simple : l'application génère une colonne avec un nom par défaut, que l'utilisateur peut modifier par la suite si souhaité.

Patrons de conception et d'architecture mis en œuvre dans le programme

Notre application suit une architecture MVC (Modèle / Vues / Contrôleurs). Le modèle centralise l'état de l'application (colonnes, tâches, archive, sélection des tâches pour le Gantt) et expose les opérations métier (ajouter/modifier une tâche, déplacer, archiver/désarchiver, gérer les dépendances et sous-tâches, etc.). Les vues (VueBureau, VueListes, VueGantt et les différentes popups) sont responsables de l'affichage des informations et de la récupération des actions utilisateur. Enfin, les contrôleurs font le lien, ils interprètent les événements (clic, drag & drop, validation clavier…) et déclenchent les méthodes du modèle.

Pour synchroniser automatiquement l'interface avec les données, nous avons mis en place le patron Observer. Le modèle maintient une liste d'observateurs et appelle notifier() dès qu'une action modifie l'état (ajout, modification, déplacement, archivage…). Chaque vue observatrice implémente actualiser() et se met ainsi à jour.

La persistance s'appuie sur un Singleton pour la classe Repository. L'objectif est de garantir une instance unique responsable de la sauvegarde et du chargement du modèle complet. Nous utilisons aussi le principe d'instance unique pour certaines vues (par exemple VueBureau et VueListes) afin d'éviter de recréer l'interface à chaque changement et d'assurer un comportement stable.

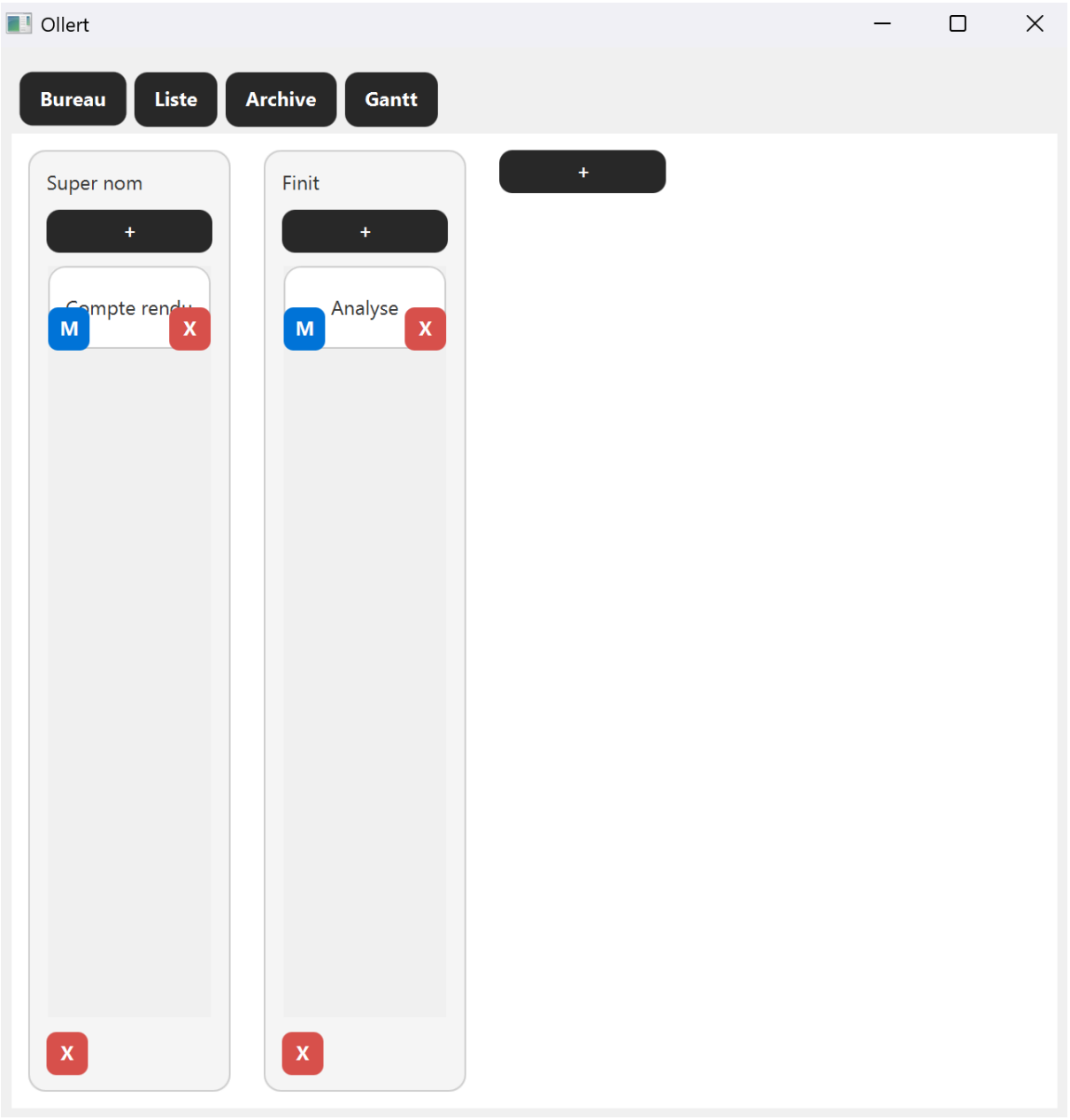
La gestion des tâches repose sur une approche Composite avec TacheComposite et Tache. TacheComposite regroupe les attributs communs (titre, description, durée, date de début) ainsi que les relations (listes de dépendances et de sous-tâches). La classe Tache hérite de TacheComposite pour représenter une tâche concrète. Cette structure simplifie l'extension du modèle et permet de manipuler les tâches de manière uniforme, tout en intégrant facilement les relations entre tâches.

Enfin, l'application utilise le patron Strategy pour le rendu des différentes vues. L'interface StrategieModeAffichage permet de choisir dynamiquement le mode d'affichage (Bureau, Liste, Gantt) en changeant simplement la stratégie utilisée, sans modifier le modèle. Cela isole l'affichage de la logique métier et facilite l'ajout d'un nouveau mode de visualisation si besoin.

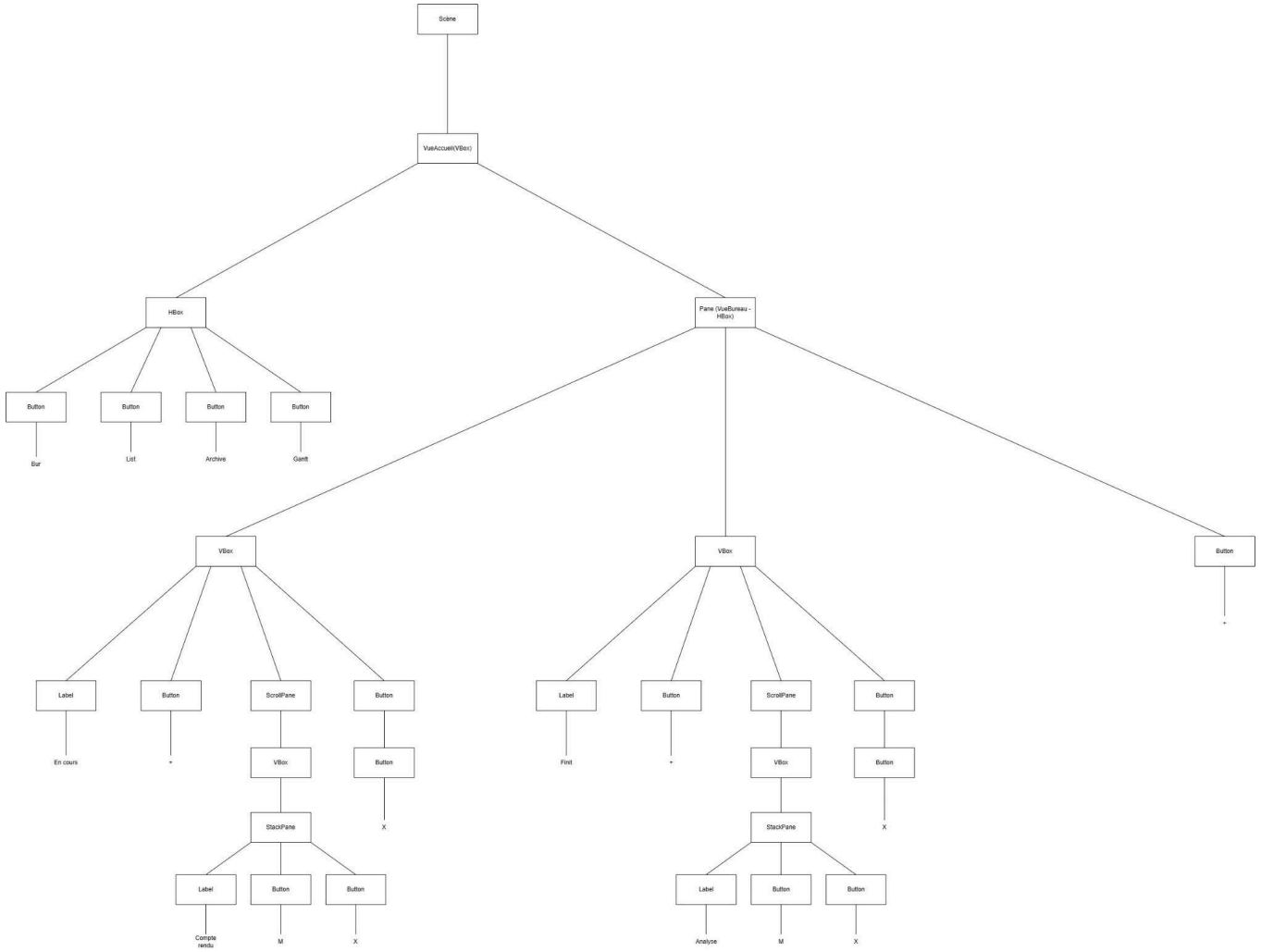
L'interface graphique

Le Graphe de scène

Le graphe de scène est celui lorsque l'on est sur la vue Accueil avec la vue bureau sélectionnée. On prendra un exemple avec 2 colonnes avec chacune 1 tâche. Comme ci dessous :



Le graphe correspondant :



L’organisation graphique

Pour changer entre la vue bureau et la vue liste, nous utilisons un patron stratégie dans la classe VueAccueil de type StrategieModeAffichage (une interface) qui est implémenté soit par un objet de type VueBureau qui permet comme son nom l’indique d’afficher la vue bureau, soit par un objet de type VueListes qui permet comme son nom l’indique d’afficher la vue liste.
Pour la vue Gantt et l’archive, nous avons juste une vue dans un pop up.

Comment fonctionne le changement de vue ?

Nous avons un contrôleur bouton permettant de basculer en mode vue liste et un autre pour basculer en mode vue bureau. Par défaut quand l’application se lance, c’est la vue bureau qui est affichée. Pour afficher les tâches archivées et le diagramme de gantt, nous utilisons pour chacun un pop-up.
Les boutons pour naviguer entre les vues sont toujours présents en haut de la vue Accueil.

Mode d’emploi pour le lancement de l’application

Il faut ouvrir le dossier Ollert_QDEV comme racine du projet intelliJ.

La version de Java

- La version de java utilisée est le jdk-24.

Les modules à installer

- Les librairies du projet sont :
    - JUnit4
    - JUnit5.8.1
    - openjfx.javafx.controls

L’organisation des fichiers

A la racine on retrouve un dossier avec les diagrammes de toutes les itérations (nommé “diagrammes”), un dossier avec les classes de tests nommé “test” et enfin un dossier nommé “src” qui contient le code de l’application. Dans celui-ci se trouve 4 packages et le fichier module-info.java. Le premier package “controlleur” contient tous les handler de de l’application, le package ”donnees” contient la structure de données, c'est-à- dire toutes les classes comme Tache, Colonne, le modèle, … Le troisième package “exception” contient une exception. Et le dernier package “vues” contient toutes les vue (incluant les popup) javaFX.

Fichier main

Le fichier main à lancer pour avoir l’application est TestVue qui se trouve dans : src/vues.